

(CN) 1. (CT) Java and Its Promise.....	1
(A-heading) What is Java and where did it come from?.....	2
(A-heading) The Big Idea, WEBOS	4
(A-heading) Java: The Good, the Bad, and the Ugly	7
(B-heading) The Good	7
(C-heading) Java is a strongly-typed language	8
(C-heading) Java is small	8
(C-heading) Java is portable.....	8
(C-heading) Java is object-oriented.....	10
(C-heading) Java has no pointers	10
(C-heading) Java has no multiple inheritance	11
(C-heading) Java has no gotos.....	13
(C-heading) Java has no global variables.....	13
(C-heading) Java has no macros.....	13
(C-heading) Java has only object oriented structures.....	14
(C-heading) Java has garbage collection	14
(C-heading) Java has standard class libraries.....	14
(C-heading) Java has boolean types	15
(C-heading) Java has security.....	16
(C-heading) Java has exceptions	17
(C-heading) Java has threading	18
(C-heading) Java has a uniform floating point specification	18
(C-heading) The compilers are getting fast.....	18
(C-heading) Strings are first-class objects.....	19
(C-heading) Identifiers have unlimited length	19
(B-heading) The Bad.....	20
(C-heading) Sometimes garbage collection is a rotten business	20
(C-heading) Java is not a pure object-oriented language	21
(C-heading) We want our overloaded operators!	22
(C-heading) The API is missing a lot of stuff	23
(C-heading) No native method support for C++	23
(B-heading) The Ugly	23
(C-heading) Arrays can be allocated with two styles.....	24
(C-heading) Java has fragile base classes.....	24
(C-heading) “Appletcations” are confusing everybody	25
(C-heading) File name class name matching	30
(C-heading) No validation system.....	30
(A-heading) The HTML Model vs. the Java Model	31
(B-heading) The HTML Model	32
(B-heading) The Java Model.....	34
(A-heading) The Java Developer Environments.....	35
(B-heading) Getting Started on the Mac with CodeWarrior.....	36
(B-heading) Getting Started on Windows 95/NT with Metrowerks CodeWarrior.....	38
(B-heading) Getting Started on Windows 95 with J++.....	39
(B-heading) Getting Started on the Mac with JDK1.02.....	41
(B-heading) Getting Started on the Mac with Symantec Café.....	43
(A-heading) Summary.....	44
(CN) 2. (CT) Java programming—the basics	49
(A-heading) MBNF Notation.....	51
(A-heading) Simple Syntax.....	59
(B-heading) Comments	59
(B-heading) Identifiers	61

(B-heading) Operators.....	62
(B-heading) Flow of Control.....	66
(C-heading) Expressions	67
(C-heading) If	68
(C-heading) While and do statements	72
(C-heading) Switch.....	73
(C-heading) For	77
(C-heading) Continue	78
(C-heading) Break	80
(C-heading) Return.....	81
(A-heading) Data Types	82
(B-heading) Primitive Types.....	82
(B-heading) Named Constants	86
(B-heading) Classes	87
(C-heading) Overloaded Methods	92
(C-heading) Static Methods.....	94
(C-heading) Null.....	97
(C-heading) Casting	98
(C-heading) Subclassing.....	98
(C-heading) Abstract Classes and Methods	102
(C-heading) Final Classes and Methods.....	103
(C-heading) Packages.....	105
(C-heading) Imports	107
(C-heading) Visibility.....	108
(C-heading) Interfaces	112
(B-heading) Wrapper classes	116
(C-heading) Boolean	117
(C-heading) Character	118
(C-heading) The numeric wrapper classes	121
(B-heading) Strings	123
(B-heading) Arrays	127
(B-heading) Vectors.....	128
(B-heading) Exceptions.....	130
(A-heading) Threads	132
(B-heading) Thread Groups	140
(B-heading) The Thread Manager.....	141
(A-heading) Summary.....	147
(CN) 3.(CT) The Graphic User Interface	148
(A-heading) The Color Class	148
(B-heading) Class Summary	149
(B-heading) Class Usage.....	150
(A-heading) The Graphics Class.....	154
(B-heading) Class Summary	154
(B-heading) Class Usage.....	156
(B-heading) How to draw a grid	162
(A-heading) The FontMetrics Class.....	163
(B-heading) Class Summary	164
(B-heading) Class Usage.....	165
(B-heading) How to Draw a String with a Background.....	167
(B-heading) How to Draw a Vertical String	168
(A-heading) The MenuItem Class.....	168
(B-Heading) Class Summary	169
(B-heading) Class Usage.....	169

(A-heading) The Event Class	170
(B-heading) Class Summary	171
(B-heading) Class Usage	174
(B-heading) Event Handling	176
(B-heading) The Keyboard	176
(B-heading) The Target	177
(B-heading) The Evt Class	179
(B-heading) The Mouse	182
(A-heading) The Component Class	184
(B-heading) Class Summary	185
(B-heading) Class Usage	188
(A-heading) The Container Class	197
(B-heading) Class Summary	197
(B-heading) Class Usage	199
(A-heading) The Frame Class	203
(B-heading) Class Summary	203
(B-heading) Class Usage	204
(B-heading) The ClosableFrame Class	206
(A-heading) The Panel Class	209
(B-heading) Class Summary	209
(B-heading) Class Usage	209
(B-heading) Building a Panel	210
(A-heading) The Checkbox Class	211
(B-heading) Class Summary	211
(B-heading) Class Usage	212
(B-heading) Adding Checkboxes to Frames	213
(A-heading) The Scrollbar Class	215
(B-heading) Class Summary	215
(B-heading) Class Usage	216
(B-heading) Adding Four Border Scrollbars	218
(A-heading) The Label Class	222
(B-heading) Class Summary	222
(B-heading) Class Usage	223
(B-heading) Adding Labels to Frames	224
(A-heading) The Choice Class	224
(B-heading) Class Summary	225
(B-heading) Class Usage	225
(B-heading) Adding Choices to a Frame	226
(A-heading) Summary	231
(CN) 4. Futil	232
(A-heading) The Dialog Class	233
(B-heading) Class Summary	233
(B-heading) Class Usage	234
(A-heading) The FileDialog Class	235
(B-heading) Class Summary	235
(B-heading) Class Usage	236
(B-heading) Futil.getReadFileName	237
(B-heading) Futil.getWriteFileName	238
(A-heading) The File Class	239
(B-heading) Class Summary	239
(B-heading) Class Usage	241
(B-heading) Ls.getDirName	244
(B-heading) Ls.deleteFile	244
(B-heading) Futil.getReadFile	245

(A-heading) The FilenameFilter interface	246
(B-heading) Class Summary	246
(B-heading) Class Usage.....	246
(B-heading) DirFilter	247
(B-heading) The FileFilter Class.....	247
(B-heading) The WildFilter Class.....	248
(B-heading) Ls.getWildNames	248
(B-heading) Ls.wildToConsole.....	249
(B-heading) Ls.deleteWildFile.....	249
(B-heading) Ls.WordPrintMerge	250
(B-Heading) Ls.lowerFileNames	251
(A-heading) The FileOutputStream Class.....	252
(B-heading) Class Summary	252
(B-heading) Class Usage.....	253
(B-heading) Futil.getFileOutputStream	254
(B-heading) Futil.closeOutputStream	255
(A-heading) The PrintStream Class	256
(B-heading) Class Summary	256
(B-heading) Class Usage.....	257
(B-heading) Futil.makeTochtml.....	260
(A-heading) The FileInputStream Class	262
(B-heading) Class Summary	262
(B-heading) Class Usage.....	263
(B-heading) Futil.getFileInputStream	264
(B-heading) Futil.available	265
(B-heading) The futils.DirList class.....	266
(A-heading) The DataInputStream Class	270
(B-heading) Class Summary	270
(B-heading) Class Usage.....	271
(B-heading) Cat.fileToStream.....	274
(A-heading) The DataOutputStream Class	275
(B-heading) Class Summary	276
(B-heading) Class Usage.....	277
(B-heading) Java, C, C++ -> HTML	279
(A-heading) StreamTokenizer.....	282
(B-heading) Class Summary	283
(B-heading) Class Usage.....	284
(B-heading) Futil.readDataFile	286
(B-heading) Futil.Print	289
(B-heading) Futil.writeFilteredHrefFile.....	291
(A heading) Exercises	294
(A-heading) Summary.....	295
(CN) 5 Digital Audio	296
(A-heading) What is Digital Signal Processing	296
(A-heading) Why do we need digital signal processing?	297
(A-heading) What is the spectrum	298
(A-Heading) What does sampling.....	302
(A-heading) Audio Files	304
(A-heading) The sun.audio	304
(A-heading) The AudioStream.....	305
(B-heading) Class Summary	306
(B-heading) Class Usage.....	306
(A-heading) The AudioData Class.....	307
(B-heading) Class Summary	308

(B-heading) Class Usage.....	308
(A-heading) The AudioDataStream Class	308
(B-heading) Class Summary	309
(B-heading) Class Usage.....	309
(B-heading) Reading and Playing an AU File	309
(A-heading) The AudioStreamSequence	311
(B-heading) Class Summary	311
(B-heading) Class Usage.....	312
(A-heading) The AudioPlayer Class	313
(B-heading) Class Summary	313
(B-heading) Class Usage.....	313
(A-heading) The μ -law CODEC.....	314
(A-heading) The UlawCodec Class	319
(B-heading) Class Summary	319
(B-heading) Class Usage.....	320
(B-heading) Reading and writing μ -law	322
(A-heading) The Oscillator Class	323
(B-heading) Class Summary	323
(B-heading) Class Usage.....	324
(B-heading) Class Examples	326
(B-heading) Class Implementation	328
(B-heading) Building the WaveTable	330
(A-heading) The DoubleDataProducer Interface	333
(A-heading) The OscopFrame Class	333
(B-heading) Class Summary	335
(B-heading) Class Usage.....	336
(B-heading) Modifying the OscopFrame	338
(B-heading) How does the OscopFrame do the scaling labels.....	339
(A-heading) The DoubleGraph Class	343
(B-heading) Class Summary	343
(B-heading) Class Usage.....	343
(A-heading) Summary.....	344
(CN) 6 Digital Audio Transform Recipes	345
(A-heading) The Discrete Fourier Transform	346
(B-heading) Bit Computations and a Log Review	351
(A-heading) The futils.Timer Class	353
(B-heading) Class Summary	354
(B-heading) Class Usage.....	354
(B-heading) BenchMarking the DFT	355
(A-heading) The Inverse DFT.....	356
(A-heading) Numeric Check of the DFT and IDFT	359
(A-heading) The FFT	361
(A-heading) The FFT Class	368
(B-heading) Class Summary	368
(B-heading) Class Usage.....	370
(B-heading) Testing the FFT and IFFT.....	373
(B-heading) Implementing the FFT.testFFT	375
(A-heading) PSD Computations	378
(B-heading) Implementation of the Transforms in the AudioFrame	380
(B-heading) A Noise filter using the FFT	387
(A-heading) Spectral Leakage of the DFT	390
(A-heading) The Hi-pass filter	397
(A-heading) Frequency shifting using the FFT.....	400

(A-heading) Resampling	402
(A-Heading) Centering the FFT	403
(A-heading) Summary	405
(CN) 7. An Introduction to Image Processing	406
(B-heading) Video	407
(A-heading) The Observer Interface	410
(B-heading) Interface Summary	411
(A-heading) The Observable Class	411
(B-heading) Class Summary	412
(B-heading) The NamedObservable	412
(B-heading) The ObservableDouble	413
(B-Heading) DoubleDialog	415
(B-heading) Dialogs in the ImageFrame	420
(A-heading) The Image Class	421
(B-heading) Class Summary	421
(B-heading) Class Usage	422
(A-heading) The ImageObserver	424
(B-heading) Summary	424
(B-heading) Image Instancing	425
(A-heading) The PixelPlane Class	426
(B-heading) Class Summary	427
(B-heading) Class Usage	429
(A-heading) The ProcessPlane Class	432
(B-heading) Class Summary	432
(B-heading) Class Usage	433
(B-heading) Class Implementation, The negate method	436
(B-heading) Class Implementation, The Shadow method	437
(B-heading) Class Implementation, The edge method	438
(CN) 9. Image Processing in Java	484
(A-heading) The Histogram	485
(A-Heading) The 2D DFT	487
(A Heading) The FFTPlane Class	492
(B heading) Class Summary	492
(B heading) Class Usage	493
(B heading) The ProcessPlane	494
(B heading) DiffCAD and the Example 2D FFT	496
(A Heading) Raster to Vector Conversion	500
(B Heading) A raster to vector algorithm	503
(B heading) The Slope class	506
(B heading) The Points class	508
(A Heading) Color Models	510
(B Heading) The HLS System	513
(B Heading) The IYQ	515
(A Heading) The FloatImage class	517
(B heading) Class Summary	517
(B heading) Class Usage	518
(A Heading) The ColorConverter class	521
(b heading) Class Summary	521
(b heading) Class Usage	522
(A Heading) The Mat3 Class	525
(B heading) Class Summary	525
(B heading) Class Usage	526
(B heading) Maple	527
(A heading) Image Geometry	531

(B heading) 2D translation	531
(B heading) 2D scaling	532
(B heading) 2D rotation	533
(B heading) Applications of affine transforms	546
(A Heading) Summary	551

(CN) 1. (CT) Java and Its Promise

In this chapter we introduce the reader to Java's good points, its bad points and its really ugly points. The overview we provide for Java must discuss the Java language specification and the Java language programming environment. The language and environment are, collectively called Java technology. The Java technology can include hardware, as well as software.

This chapter is divided up into 5 main sections.

The first section, "What is Java and where did it come from", introduces the Java technology. We show that the term Java has come to mean both the Java programming language and the technology needed to support that language.

The second section, "The Big Idea, WEBOS" describes the current state of the art in Java. An overview is given of the picoJava technology, the core of the Java chips which Sun intends to release soon. We also describe what the effect may be when inexpensive Java appliances become embedded in our society.

In the third section, "Java: the good, the bad, and the ugly", we tell it like it really is. Java has some really good points, but it also has problems. This section outlines both. Please keep in mind that we really like Java (No, REALLY!). Still, we do not pull punches here. Every programming language has problems, and Java is no different. You, the reader, should put your best foot forward when stepping into Java, but watch where you are putting it!

The fourth section, "The HTML Model vs. the Java Model" describes the HTML model which has formed the basis of the world wide web and the current problems with the diverse nature of data representations. We also speak about the big idea behind the Java model and how it may help to reduce the decoding problems that our web browsers currently face.

(CD-ROM icon) The fifth section, "The Java Developer Environments" gives a summary of how the reader can get started using the book's software. (END CD-ROM icon) A few software products are reviewed and some are actually useful. There are several products which are only just out and appear to consume more time and money than they are worth. The reader is advised to select a programming tool with care. Often this means budgeting for more than one compiler, and testing it yourself!

(A-heading) What is Java and where did it come from?

Java is a name which represents both a language and a technology for the support of the language. When we speak of the Java programming language we are talking about an object-oriented language developed by Sun Microsystems. This language has syntactic similarities with several other languages. It has the braces ‘{’ of C, C++ and Objective C. It has the exception model of ZetaLisp, a flavors-based Lisp that ran on Lisp Machines. This is the same exception handling that has been proposed by Bjarne Stroustrup for use with the ANSI C++ standard [Spuler].

When we speak of the Java technology, we are talking about the Java programming language and its support systems. These systems include a large library of classes, called the Java class libraries. Java technology also includes a specification on run-time behavior, achieved using a Java machine specification. The Java technology provides that the Java machine may be implemented in any combination of software or hardware.

When the Java machine is implemented in software it is called the Java virtual machine. Java started life as a language called Oak. It was designed to incorporate the best features of past languages into a single new language. Just as important, the design of the Java programming language would leave out features which were thought to make the language less reliable. In the balance between speed and reliability, the Java designers chose reliability. This is a design criterion that is inherently different from C++, for example, with C++, features were added to the language, without any features being removed. Further, it was an important design feature that C++ run as fast as C [Stroustrup].

A design objective of Java is that it be useful for distributed computing. In the distributed computing model, code can be downloaded for execution on demand in a secure fashion. Security became an important issue in the design criterion. If the code source was not trusted, the code itself had to be treated as potentially harmful to both the user’s data and the computer hardware. The danger to the users’ data could include access to and distribution of sensitive information, like credit card numbers, bank account numbers, and other proprietary data.

The Java machine specification is a Java language support technology that has become an integral part of the language. The tight integration of the Java language with the Java machine specification is probably one of the main contributions of Java to the computer science community. The Java virtual machine achieves a layer of isolation between the running Java program and the underlying hardware. This isolation provides security and portability. The Java virtual machine provides security by optionally creating a security manager. The security manager can keep a program from performing those tasks considered a security risk. The Java virtual machine provides portability in that the virtual machine itself can run on several hardware platforms.

(A-heading) The Big Idea, WEBOS

If we consider all the web servers on the internet as being part of a large computer system, then the web is the largest operating system in the world. In fact, the web's programming language is Java and so, from this point of view, Java is an operating systems programming language. Sun is planning to release Java machines which are not virtual. This means that the Java machines will be implemented in hardware. The operating system for these machines will be written in Java. No longer will people have to write cryptic C code to modify the kernel of an operating system, they will be able to write in Java. Java will truly become an operating systems programming language. When this happens Java will probably spread into embedded system design until every appliance on the planet supports Java, even our toasters!

Consider, if you will, the telephone. When unplugged from the network, the telephone is a useless piece of plastic, not worth the \$20 it costs to buy. The value added by the telephone is the network into which it is plugged. The same may be said of embedded systems on the internet. A toaster on the net can download operational parameters (when to turn on, for how long, etc.) and can use the network to communicate issues regarding its state (sorry to interrupt your net surfing, but the toast is done!).

Java chips are going to greatly reduce the price of an embedded Java controller.

Dedicated chips will give embedded controllers speed and price advantages over their non-specialized hardware counter-parts. Sun is targeting the consumer market with mass sales of cheap chips.

Some devices targeted include TV set-top boxes, cellular telephones, pagers, digital TVs, smart VCR's, PDAs, printers, copiers, etc. In short, anywhere we find an embedded computer, Sun wants that computer to run Java. These chips will run byte code natively hence there will be no need for a just-in-time compiler. Such devices may not have a display, much memory or no connection to a network. As a result, the API targeted for such embedded controllers is stripped down to a bare minimum. This minimal API is called the Java Embedded API. As of this writing, there has been no published standard for the Java Embedded API.

Sun Microelectronics, the Sun semiconductor division, calls its first chip architecture Java One. Sun plans to release two families of chips; microJava and ultraJava. MicroJava is a low-cost (<\$25) chip intended to target the embedded controller market. UltraJava is a higher-cost (<\$100) chip intended to target the workstation market. At the heart of the technology is a *super-scalar* stack-based RISC machine called picoJava. PicoJava is super-scalar because it implements a 4-stage pipeline which enables different parts of the processor to work on 4 different tasks at once. It is RISC (Reduced Instruction Set Computer) because it executes most instructions in a single clock cycle.

Computing in a super-scalar pipe-line is like using an assembly line. Data is passed from one worker to the next, and a process is applied to it. Figure 1.1 shows a sketch of the pipeline which, when filled, will permit picoJava to fetch, decode, execute & cache and then write-back its results [Varhol].

Figure 1.1. Four-stage picoJava pipeline

During the fetch operation, picoJava will load a 4 byte cache line into its processing stack. The stack consists of 64 32-bit registers implemented on-chip. After the on-chip storage is exceeded, RAM is used to implement the stack.

In addition to using ultraJava to target the workstation market, Sun will attempt to use ultraJava to penetrate the *network computer* market. The network computer is a stand-alone computer connected to an enterprise's network infrastructure. The primary market consists of companies that want to centralize administration by maintaining a few servers. This simplifies the deployment of applications, by permitting them to be automatically downloaded over a network [Madany].

The proposed picoJava system shortens the path between the Java programs and the hardware, by implementing the Java machine in hardware. This cuts out the adapter layer and uses a special operating system that is designed for the picoJava machine, called Kona. This is shown in Figure 1.2.

Figure 1.2. The picoJava Kona system
The reader should keep in mind that picoJava is still in development. No silicon has been built yet and so there have been no benchmarks run.

(A-heading) Java: The Good, the Bad, and the Ugly

Java is spreading through the computer science and engineering community like wildfire; yet, there is cause for caution. People are asking hard questions. Is Java suitable for engineering? Is Java suitable as a first programming language? Can Java be used throughout the computer science and engineering curriculum? Is Java suitable for writing large programs? What are the drawbacks in being an early adopter of the Java technology?

There is great hype in the media today, as result objective answers to these questions are not easy to come by. In fact, Java may not be suitable for writing large programs and there may not be enough textbooks to use Java across the curriculum. Beta software is enough of a drawback to make any early adopter of a technology cringe. Being a beta tester of a compiler is not everyone's idea of a good time!

In this chapter, we attempt to balance our view of the language with a list of Java's good points, its bad points, and yes, its really ugly points. We owe it to you, the reader, to say that being an early adopter of this technology comes at a cost.

This cost comes from the time spent reading **many** Java books, writing custom libraries, buying new software, using beta compilers and being the first (and sometimes only) Java programmer on the block. For us the cost has well been worth it, but you, the reader, must make your own decision. Use your judgment!

(B-heading) The Good

In this section we describe the good points about Java. Sometimes a good point about a language is also a bad point! For example, we cite garbage collection as both a good point and a bad point about the language. It is good because it permits the programmer to forget about memory management during the programming task. Garbage collection simplifies design and eliminates a source of errors. The garbage collection is bad because it takes system resources and could make Java unsuitable for low-level embedded control, a task for which it was intended.

(C-heading) Java is a strongly-typed language

Java is a strongly-typed language. All class names are treated as types and used to check any reference to a class when passed as an argument to a method. Most modern languages have this feature although the old style of C avoids it.

(C-heading) Java is small

Java is based on a small byte code interpreter. Including the self-contained microkernel, the byte code interpreter plus supporting classes is 215k bytes. This is a remarkable achievement. It means that byte code interpreters can reside on small ROMs and provide micro-controllers a means of running Java programs.

(C-heading) Java is portable

Java is a multi-platform language. In Java, the model is that you “Write Once, Run Anywhere”™. Because there is only one virtual machine specification Java can provide a standard, uniform programming interface to applets and applications on any hardware. The Java Platform is therefore ideal for the Internet, where one program should be capable of running on any computer in the world. When you compile Java source, you obtain *byte codes*. Byte codes are output by the Java compiler and form instructions to a *Java virtual machine*. Java is said to be a portable language in that it can run on any hardware on which the Java virtual machine can run. Byte codes are stored into *class* files. Class files are downloaded to a Java virtual machine that contains a byte code interpreter. Thus Java is a “Write Once, Run Anywhere”™ type language. This is like the Pascal P-code concept of 20 years ago (which required a P-machine to execute the P-code) [Bowles]. So, when we speak of Java as a multi-platform language, we mean that it will run wherever there is

an implementation of a Java virtual machine. A sketch of the relationship between the Java program and the hardware is shown in Figure 1.3.

Figure 1.3 A Sketch of the Java Model

The multi-platform nature of Java is one of its strongest selling points. This can have a profound impact on how we judge our computing resources. For the first time, we can bench-mark precompiled code on a wide variety of platforms. This enables us to ignore compiler optimization for a specific machine. If we have a Java virtual machine that is optimized for the hardware on which it runs, we should have a good measure of the machine's relative speed when running Java. Optimizing a Java virtual machine for specific hardware is not an easy task, however. At present, for example, there are no Java virtual machines optimized for multi-processor systems [Oaks et al.]. Thus a threaded Java program cannot take advantage of the existence of more than one CPU. When this changes, Java will be a portable concurrent programming language.

(C-heading) Java is object-oriented

Java is an object-oriented programming language. In an object-oriented paradigm, an instance of an object contains both data and the algorithms needed to manipulate the data. This is held in contrast to the programming languages that pass data as arguments to procedures. There are no functions in Java, unlike Pascal, C, FORTRAN or C++. In Java all *methods* must reside in *classes*. C++ is a language with object-oriented extensions. This means that non-object oriented programs can still be written in C++. This is generally not true in Java.

(C-heading) Java has no pointers

Java is a more crash-proof language than C, C++, and Pascal. This is a very good feature, indeed! One reason why is that Java does not provide a mechanism for directly manipulating pointers. Thus there is no way for the programmer to obtain a memory address. Further, there is no pointer arithmetic and there are no pointer operations. Java eliminates the possibility of overwriting memory and corrupting data.

In C or C++ you may dereference a NULL point using

```
*ptr
```

When *ptr* is NULL, this causes a “segmentation fault” error on UNIX, or an immediate crash, on some other machines. Some times pointers in C or C++ are pointing to illegal locations in memory. When these locations are accessed, this too can cause a crash. This type of error is called a dangling reference. Another type of error is called a memory leak. This is created when data that has been discarded is not reclaimed. This can create an out of memory error which will crash the program (or computer) if it is not tested for [Spuler].

There are *many* ways in which incorrect pointer use can crash a computer or program. There is simply not enough space to list them all. We can be thankful that Java has no pointers.

(C-heading) Java has no multiple inheritance

Multiple inheritance, as it was known in C++, has been eliminated in Java. Multiple inheritance is the ability to have two or more direct base classes. In 1966, multiple inheritance was rejected as a feature in Simula by Ole-Johan Dahl. The rationale for the rejection is that it would complicate the garbage collection. Also Smalltalk does not support multiple inheritance [Stroustrup 94].

The problem with multiple-inheritance is that duplicate class variables and method names must override each other, according to some policy. This policy becomes a part of the language, and can often be forgotten by the programmer. Elimination of multiple inheritance reduces the possibility of programmers getting confused about which method is in effect. It also limits the kinds of inheritance which can be performed. Java will only permit an A-Kind-Of (AKO) type taxonomy of class inheritance. Figure 1.4 shows a sample of an AKO class inheritance.

Figure 1.4. Example of an AKO inheritance

In Java, an animal class may be extended to create a mammal sub-class, thereby indicating that a mammal is a kind of animal. Since a student and a professor are both humans they also inherit traits from mammals. In most other object-oriented languages, two or more AKO inheritance chains may be mixed. Figure 1.5 shows an example of multiple inheritance.

Figure 1.5. An Example of Multiple Inheritance.

With multiple inheritance, the attributes associated with the *stream* class, may be inherited by both the *input-stream* and the *output-stream*. When these two classes are joined by a third class, the *input-output-stream* we have *multiple inheritance*. Multiple inheritance is a language feature which enables the programmer to reuse the data-structures and methods from two parent classes.

The multiple inheritance controversy is a language feature discussion that appears to lack practical evidence. There is no doubt that method name ambiguity must be resolved, either at compile-time or at run-time. Also, since Java must load libraries dynamically, it seems that the ambiguity would have to be resolved by the class loader. Thus we suggest that one reason Java eliminated multiple inheritance was not because of the language feature controversy, but because of implementation simplicity.

The removal of multiple inheritance is a design trade-off. The decision to remove multiple inheritance probably helped the code become more reliable, simplified garbage collection and simplified the class loader. Multiple inheritance is probably missed by all who are used to having it.

(C-heading) Java has no gotos

There are no goto's in Java. It is possible, however, to perform a multi-loop break using the *break <label>* feature. This is a good thing because it will probably lead to more structured code and eliminate a fruitful source of bugs.

(C-heading) Java has no global variables

There are no global variables in Java. Instead, there is access control to classes that have variables and methods. Access control enables the programmer to create policy about visibility. Visibility restrictions permit public access. When access control is applied to a class that has a *static class variable*, it could be argued that the variable is global to all other classes through the class name. The variable is accessed by *className.variableName*. For example *Math.PI* is a global reference to a class variable.

(C-heading) Java has no macros

There is no preprocessor in Java, no macro language (like in RatFOR, C and C++). There are no compiler directives. It is not possible, as in C to create language extensions, for example:

```
#define { Begin
```

is likely to confuse editors and allow people to create an ALGOL or Pascal like style for code blocks. This is not permitted in Java.

C/C++ macro facilities permit confusing function calls errors. For example:

```
#define cube(x) x * x * x
x = cube(x+1);
```

expands into

```
x = x + 1 * x + 1 * x + 1;
```

Thus *x* becomes $3*x + 1$ and not $x*x*x$.

(C-heading) Java has only object oriented structures

There are no structures or union operations in Java. All of the types are implemented as classes. This is probably a good thing, since the data manipulation methods can be built into the data structures.

(C-heading) Java has garbage collection

Java has automatic storage operations, these include a garbage collection mechanism. Garbage collection enables the Java virtual machine to reclaim storage used by discarded instances. The garbage collector may be explicitly invoked by using the *gc* method of the *System* class. The Java virtual machine will perform garbage collection without explicitly invoking *System.gc()*. The garbage collector is the only mechanism available to free storage in Java. To get an instance to be reclaimed by the garbage collector, you must remove all references to the object, then, either invoke the *System.gc()* or wait for the garbage collector to come and reclaim the storage. The existence of the garbage collection mechanism in the Java virtual machine means that the programmer will never have to worry about keeping track of storage.

(C-heading) Java has standard class libraries

Java has standard libraries that include an Abstract Window Toolkit (AWT). The AWT enables object-oriented Graphic User Interface (GUI) based programs to be portable. Others have tried this in the past, but have not had much commercial success [Watson]. The class libraries have eight major packages, and this number is growing. There is an input-output package, *java.io*, that enables a user to perform input and output stream manipulations. The intention is that this makes file and network data I/O manipulations into stream manipulations. Further, that these stream manipulations work without direct involvement with the source of the stream. This provides a layer of abstraction which makes I/O programming much easier to perform in a more general manner.

There is a network package, *java.net*, that enables socket and Universal Resource Locator (URL) manipulations. The network package provides a standard, built-in method for turning sources and sinks of network data into streams. Once this occurs, the I/O package can be used to manipulate the streams.

There is a utilities package, *java.util*, that contains several features held standard in operating systems. Features like getting the date, time, random numbers, etc.

These packages are a starting point upon which, Java programmers may build portable programs. Any packages that build upon these core Java packages will be portable. Also, the core Java packages are typically built into the systems that support Java. As a result, the core packages do not have to be downloaded every time a Java program needs something in a package. This makes the Java programs faster to download and it makes the byte code files more compact.

(C-heading) Java has boolean types

Java uses a conditional statement that takes a boolean as an argument, whereas languages like C or C++, permit an integer to be used as an argument to a conditional.

For example, in C, or C++ it is possible to write:

```
if (1) {fprintf(stderr, "1 is not a boolean");}
```

In Java the argument to a conditional must be of boolean type. Using an integer as an argument can create confusion between assignments and tests for equality. For example:

```
if (a = 0) {fprintf(stderr, "this will never be printed");}
```

is a bug, since *a* will be assigned to the value 0. In fact the ‘==’ operator is needed so that:

```
if (a == 0) {fprintf(stderr, "this might be printed");}
```

With the use of Java, the assignment-test confusion becomes a bug of the past.

(C-heading) Java has security

There is a class of programs called "applet viewers". Applet viewers have their own Java virtual machines. Java enabled browsers have built in applet viewers.

Secure viewers protect the system by making an instance of the security manager. Most Java-enabled browsers make an instance of the security manager.

Some applet viewers will typically permit the running of Java programs without making an instance of the security manager. Thus applets and applications are subject to the same security procedures.

The security manager can disable operations that are considered dangerous (e.g., file i/o, creating consoles, or running native methods).

A Java program that makes an instance of a frame when an instance of the security manager is in-place will get an "untrusted Java Applet" label on the windows. This alerts the user not to type sensitive data into the applet.

(C-heading) Java has exceptions

Java has a form of control structuring known as *exception handling*. Exception handling is a provision for handling those abnormal circumstances which can prevent execution from successfully continuing. For example, subscript boundary violation, division by zero, overflow, I/O errors (from unavailable files, or insufficient disk space) etc. Java's exception handling mechanism can prevent a program from terminating abnormally.

Exception handling is not a new idea and has been widely available in some languages (i.e., Ada, COBOL, C++, Delphi (<http://www.borland.com>), PL/I, and ZetaLisp) but not in others (i.e., Basic, FORTRAN, Pascal, C) [Goodenough]. The Java exception handling is closer to ZetaLisp than to any other language. Exceptions are subclassed in Java and so are treated in an object-oriented fashion (as opposed to Ada, COBOL or PL/I).

The Java language specification identifies compile-time and run-time errors. For example, accessing an array index out of bounds is a run-time error, according to the Java Specification [Gosling et al.]. In C, the effect is dependent on the operating system. For example, in Solaris (the Sun operating system) this causes a segmentation fault, and a core file is dumped. On the Windows 3.x/95 or MacOS the computer crashes. Windows NT may handle the error with a little more grace. But Java emits the following message:

```
java.lang.ArrayIndexOutOfBoundsException: 10
  at TrivialApplet.test(TrivialApplet.java:18)
  at TrivialApplet.main(TrivialApplet.java:12)
```

This is really pretty civilized, compared with crashing the computer.

(C-heading) Java has threading

Threading is a built-in feature of Java. A thread is a low-overhead context switch which enables a processor to change from one task to another very quickly. All the threads in Java could execute in parallel, if the Java virtual machine existed which could take advantage of multiple processors. This is not the case, however, and so only one thread can run at a given time. Threading is a high-level concurrent programming facility. Besides Java, several other languages provide a high-level concurrent programming facility. Examples include Concurrent C, Concurrent C++, Concurrent Pascal, Concurrent Euclid, Modula-2 and Ada [Gehani].

In the past multiple threads were programmed using support from the operating system. Java abstracts this relationship with the operating system by specifying how the virtual machine will behave when threading. There is nothing in the Java threads API that requires any operating-system involvement. In fact, the thread library of Solaris, on the Sun workstation, is unused in Java 1.0.

(C-heading) Java has a uniform floating point specification

Java uses the IEEE 754-1985 floating point specification as a part of its language definition. Thus round-off errors can be predicted in a platform independent manner.

(C-heading) The compilers are getting fast

The compiler technology is improving for Java. For example, there are now “Just-in-Time” compilers which permit the compile once-run anywhere model of Java to be just as fast as compiled native code. The just-in-time compilers take the Java byte codes and compile them to native machine language. Our benchmark indicated an 18x speed-up over interpreted byte codes!

This comes at a cost, however. There may be longer start-up time, though we could not verify this. Also, the JIT compiler is supposed to take more RAM, though we could not verify this, either. The overall 18x speed up more than made up for any initial start-up costs on the DiffCAD program. Our benchmark was performed with a Metrowerks compiler running under MacOS. JIT compilers are not available on all platforms. If they were they would probably replace the interpreter model.

(C-heading) Strings are first-class objects

Strings are not character arrays, they are instances of the *String* class. Thus, you must access them via method invocation and not like the array of characters in Pascal, C or C++. This is a much cleaner way to manipulate strings and leads to better code.

(C-heading) Identifiers have unlimited length

According to the Java language specification, identifiers may have an unlimited length. We have verified this for some very large values, at least. For example:

```

    int
    ThisIsAVeryLongNameInJavaWithMoreCharactersThanOneWouldTypi
    callyUseYouMayUseNumbers1234567890ButNoOperators=0;
    int
    ThisIsAVeryLongNameInJavaWithMoreCharactersThanOneWouldTypi
    callyUseYouMayUseNumbers1234567890ButNoOperatorsAndTheyAreU
    nique=1;

    System.out.println(ThisIsAVeryLongNameInJavaWithMoreCharact
    ersThanOneWouldTypicallyUseYouMayUseNumbers1234567890ButNoO
    perators+

    ThisIsAVeryLongNameInJavaWithMoreCharactersThanOneWouldTypi
    callyUseYouMayUseNumbers1234567890ButNoOperatorsAndTheyAreU
    nique
    );

```

The unlimited identifier length should apply, in theory, to class names. Some development systems require that public classes be stored into files that have names that match the class name. For these systems, it is not possible to have a public class identifier that exceeds file-name length limitations. These are implementation dependencies and not limitations imposed by the Java language specification.

(B-heading) The Bad

Sometimes the best features of Java are some of the bad features of Java. For example, garbage collection has both good points (hence its listing in the previous section) and its bad points (see below).

(C-heading) Sometimes garbage collection is a rotten business

The draw-backs of garbage collection are:

- The garbage collector can lead to non-deterministic program run times.
- For large systems, garbage collection can use a significant amount of CPU time.

For example, during a time-critical interrupt, the Java virtual machine could sense that it is time for garbage collection. This could result in the loss of data, property or even life! As far as we know, there is no way to turn off the garbage collection from within the Java program. Some Java interpreters (like the Metrowerks *javai*) have flags which disable asynchronous garbage collection. For example:

```
javai -noasyncgc
```

Keep in mind, however, that just because asynchronous garbage collection is turned off, doesn't mean you can stop worrying about it. In fact, quite the opposite is true. Turning off garbage collection means you must invoke it yourself (or running out of memory).

As anyone who has some experience in programming large garbage-collection based systems (like Lisp Machines) knows, finding garbage is no easy task! The Lisp Machine had a *gc-immediate()* function. When run, *gc-immediate()* started the garbage collection (just like Javas' *System.gc()*).

Garbage collection in virtual memory typically causes a condition known as *thrashing*. Thrashing occurs when virtual memory is accessed in a non-sequential fashion. Thrashing causes different parts of the memory to be continually swapped in and out of the disk. Keep in mind, RAM access time (measured in nanoseconds) is six orders of magnitude faster than disk access time (measured in milliseconds) so that thrashing can cripple even the fastest of machines.

(C-heading) Java is not a pure object-oriented language

Java is not a *pure* object oriented language. You cannot make an instance of any *basic* data type. The basic data types in Java are *boolean*, *int*, *long*, *float*, *double*, *char* and *byte*. Compare this situation with Smalltalk, in which even the basic data types are classes.

(C-heading) We want our overloaded operators!

Java does not permit the creation of overloaded operators. Contrast this with C++, which allows a programmer to give operators a context dependent

meaning. For example, in C++, the '*' operator can take two arrays as arguments and then multiply the arrays together. The Java designers did not appear to trust programmers to use the overloaded operator feature without writing cryptic code. (WARNING) To add insult to injury, the Java language designers felt it would be OK for them to overload operators as a part of the language. In Java, the '+' operator is overloaded to concatenate strings. For example:

```
int x = 2; int y=3; String z = "4";
System.out.println( x+z+y );
```

will treat *x*, *y*, and *z* as string objects and output

243

But

```
int x = 2; int y=3; String z = "4";
System.out.println( x+y+z );
```

will treat *x* and *y* as numeric objects, add them, then convert the result to a string, concatenate the string with *z*, then output

54

Thus, the overloaded operators have become argument dependent and have permitted the kind of cryptic code the Java designers' wanted to avoid. (END WARNING)

(C-heading) The API is missing a lot of stuff

Java is missing key features from its supporting API (Application Programmer Interface). For example, you cannot write to a serial port (despite the fact that nearly every computer has one). You cannot (as of this writing) output to a printer from within a Java program. This is due to the newness of the Java language. Since features are missing from the API, Java may not be easy to use for some applications.

(C-heading) No native method support for C++

You may like to extend the features of the API by programming in another language. Unfortunately the choice of language is currently limited to C.

There is no way, at present, to link between Java and C++. This is due, in part, to the problem of *name-space mangling*. In C++, the function identifier in source is mapped into a different function name for the linker. This mapping is called name-space mangling. Functions are typically mangled

according to their argument type. Different compilers may have different mangling schemes. Since Java has no way to know how functions will be mangled, the functions cannot be invoked.

The Java native language interface is not complete (as of this writing) but that is likely to change soon. When it is done, circumstances are likely to get better.

(B-heading) The Ugly

No language is perfect, but Java does have its design flaws. In this section we cover the design flaws of Java that probably will not go away. Some are just harmless and ugly. Others, like the fragile base class problem, could cripple Java for large software system development.

(C-heading) Arrays can be allocated with two styles

Java supports the “C” and “Java” style of array allocation. In fact, the two styles of array allocation are supported within the same statement. Thus,

```
int [][] i = new int[3][3];
int j[][] = new int[3][3];

// and now we put the Ug in Ugly!
int [] k [] = new int[3][3];
```

are three, syntactically acceptable ways of specifying a two-dimensional array of ints.

(C-heading) Java has fragile base classes

Java suffers from the fragile base class and *interface* problem. In Java, an interface can be used to store constants and to permit class and method specifications. For example, in the DiffCAD program (used as a central example in this book) there is an interface called *Constants* that contains a list of commonly held constants. One line in *Constants* is:

```
final double Pi_on_2 = Math.PI/2;
```

Suppose another line were added, say

```
final double Pi_on_4 = Math.PI/4;
```

This requires that every source code file that refers to *Constants*, (in DiffCAD’s case, 7 files) to be recompiled. Including the linking phase, the recompilation takes 56 seconds on the authors’ machine, a PowerMac 8100/100 Mhz with a PowerPC 601 and 72 MB RAM. As another example, there is

an abstract base class called *Computation*. When *Computation* is altered, 5 files require recompilation and, including the linking phase, 70 seconds elapse before the program begins to run. Thus, when programs become large, the fragile base class and fragile interface can cripple the programmers' productivity [Lewis].

(C-heading) “Appletcations” are confusing everybody

The Java language has led to a source of continuous confusion regarding the difference between an *Applet* and an *Application*. There is a package of classes in the core Java API called the *java.applet* package. This is a very unfortunate naming convention. Within the *java.applet* package, there is a class called the *Applet* class. The *Applet* class is extended to create subclasses. Instances of *Applet* subclasses are called *Applets*.

Definition 1.1: An *applet instance* is an instance of a class that extends the *Applet* class.

Definition 1.2: An *application instance* is an instance of a class that contains a *main()*.

Lemma 1.1: To run a Java application it is necessary and sufficient to both have a *main()* and invoke the *main()*.

Corollary 1.1: Having a *main()* in a Java program is a necessary, but not sufficient condition for running a Java application.

Lemma 1.2: To run a Java applet it is necessary and sufficient to extend the *Applet* class, implement the *init()* and invoke the *init()*.

Corollary 1.2: Subclassing the *Applet* class in a Java program is a necessary, but not sufficient condition for running a Java applet.

Note that Corollary 1.1 follows directly from its parent, Lemma 1.1. Similarly, Corollary 1.2 follows from its parent, Lemma 1.2. Definitions do not follow the construction of the lemmata, Pronounce the ‘a’ in lemmata short, like “what’s a-madda?”.

In common use, the term applet has come to mean “a small Java application run from within a browser”. We class such definitions as strictly

incorrect. The reader will see in the following code, a segment of a large application, called *DiffCAD*, which dispatches a large number of different applets from within a Java application.

```

    if (arg.equals("benchmark")) {
        AppletFrame w = new
            AppletFrame("BenchmarkApplet");
        String title ="BenchmarkApplet";
        String args[] ={" "};
        w.startApplet("BenchmarkApplet",title,args);
    }

    if (arg.equals("surface")) {
        AppletFrame w = new AppletFrame("surface");
        String title ="surface";
        String args[] ={" "};
        w.startApplet("surface",title,args);
    }

    if (arg.equals("search yahoo")) {
        AppletFrame w = new AppletFrame("Wa hoo!");
        w.startApplet("SearchYahoo",title,args);
    }

```

In fact, the applet is just a kind of *Frame*. It runs in its own *thread* and has its own *applet context*. The point is that a large program can run many applets. A Java *application* is typically a program which contains a *main*. For example:

```

public class TrivialApplication {

    public static void main(String args[]) {
        System.out.println( "Hello World!" );
    }

}

```

is an application. An *applet* is an instance of an *Applet* subclass. For example:

```

import java.awt.*;
import java.applet.Applet;

public class TrivialApplet extends Applet
{
    public void init() {
        repaint();
    }

    public void paint( Graphics g ) {
        g.drawString( "I am an Applet", 30, 30 );
    }

}

```

One popular Java reference states that a *class* becomes an applet by subclassing the *Applet* class and, that an applet is an “embeddable window” [Chan and Lee]. No wonder even seasoned Java programmers misuse the *applet* term!

To top off the example, we present the *Application-Applet* that is both an extension of the *Applet* class AND contains a main! This is shown in the following listing

```
import java.awt.*;
import java.applet.Applet;

public class TrivialApplet extends Applet
{
    public void init() {
        repaint();
    }

    public static void main(String args[]) {
        System.out.println("An Appletcation");
    }

    public void paint( Graphics g ) {
        g.drawString( "Hello World!", 30, 30 );
    }
}
```

This code may be called as an applet or as an application. When called as an applet, the *init()* method will be invoked and the “Hello World” will be drawn. When called as an application, *TrivialApplet* class is loaded, the *main* will be invoked and

An Appletcation will be emitted to the screen. *TrivialApplet* is a subclass of an *Applet* class, but may be used as an applet or an application, depending on context! This permits the formulation of lemma 1.3 and corollaries 1.3a and 1.3b.

Lemma 1.3: Applets are run by invoking `init()`, applications are run by invoking `main()`.

Corollary 1.3a: The difference between an applet and an application is the invocation and not necessarily the content.

Corollary 1.3b: Applets do not automatically run their `main`'s. Applications do not automatically run their `init`'s.

(C-heading) File name class name matching

Some compilers, like JDK, J++ and the Symantec products, require that the file name of the Java source code matches that of the public class name contained in the file. This is not a part of the language specification [Gosling et al.]. It is a restriction imposed by the compiler implementation. This restriction is not uniformly imposed. For example, the Metrowerks CodeWarrior IDE for MacOS and Windows 95/NT does not impose this file name - class name conformance.

The non-uniform restrictions make the porting of source code from one compiler to another a time consuming task. Anything that prevents Java from being ported is a very bad feature indeed. Thankfully, this bug is an artifact of the implementation of the compiler products and not of the language specification. It is unfortunate that the bug has become so widespread in the compiler community as to become an accepted limitation of the language. It is our hope that Sun will correct this bug in their own compiler soon. It is much easier to distribute one file with several small classes. The alternative is to make several small source files.

(C-heading) No validation system

At present, there is no validation system for a Java compiler, or supporting Java technology. This is a critical need, since there are so many products which appear to violate the compile-time and run-time specifications as laid out by Sun.

A validation system would include series of test programs that would generate known compiler errors and known run-time errors. Such programs should elicit specific kinds of behavior from the Java Class Libraries. This has not been done, as far as we know. For example, the return of the date and time on J++ includes a mention of daylight savings time. This is not the case with any other Java environment that we know of. A run-time validation suit should detect such an error.

Bugs in the J++ compiler, which are described in the “Getting Started in Windows 95 with J++” section, could have been caught and corrected, had a compiler validation suite been applied. This must be a top-priority item, if the quality of the Java tools available is to be maintained.

(A-heading) The HTML Model vs. the Java Model

The HTML (HyperText Markup Language) model is one which permits a document to make references to files in other formats. The responsibility of a browser is to read the references to the HTML files and dispatch them to a decoding program. For example, if the file is compressed, a decompression program may automatically be started by the browser.

The fatal flaw in this model is that browsers (and their supporting applications, known as *helper apps*) can grow without bound. One browser, called Netscape, for example, recommends 16 MB of RAM. As the applications become large and bloated, they also tend to slow down, even for simple tasks.

In this section we compare the HTML model with the Java model. In the Java model, code is compiled into class files and then downloaded, over the net, into an applet viewer. The applet viewer is used to decode the data stream which follows. The theory is that Java will become the language for decoding a wide range of data and that all a browser will have to do is support an applet viewer. For this model to work, the data must point to Java decoders that can be downloaded on demand.

(B-heading) The HTML Model

On the internet, there are computers that run programs called *Hyper-Text Transfer Protocol* (HTTP) servers. HTTP servers typically send data in response to a web browser request. Generally, the data can be in any format, the HTTP server typically does not decode the data. As a result, HTTP servers of the internet provide a wide variety of interesting and wonderful data formats to various browser-based clients. New formats appear all the time. Browsers typically understand some variant of HTML and this has led to the *HTML model*.

In the HTML model, raw data is embedded in the HTML document by a hypertext reference (known as the *href* tag). In order to assist the browser with the decoding of the wide and growing number of data formats, browsers use *helper applications*. In order to map the data to the correct helper application, browsers have a protocol that looks at the *Multipurpose Internet Mail Extension (MIME)* that the HTTP server transmits with the data. Based on the MIME extension, a lookup table determines how to decode and present the data. Figure 1.6 shows a screen capture of a presentation of one such table, known as the *helper window* (Netscape 3.01).

Figure 1.6. The Netscape Helper Window

For each data type supplied by the HTTP server, there is a corresponding *helper application* or *plugin*. When this application is not present, the browser will typically ask if the user wants to save the file format. One of the authors has over 77 items listed in the Netscape helper applications window. Naturally, these do not represent all the possible data formats which a browser can handle. A browser can be customized to handle any data format, by launching a helper application. Thus, there are no limits to the number of data formats which may be present on the web or handled by a browser.

The same content will often be presented to the user in a variety of electronic forms, a veritable electronic tower of Babel. Suppose, for example, a Microsoft Word document is to be supplied via the WEB. One could supply it as a word document, but Word 5 on a Mac cannot read Word 6 or 7 documents. So we could supply it as an RTF (Rich Text Format) file, so that Word 5 will understand most of it. The drawbacks in distributing Word documents using RTF to a variety of Word versions are that some formatting will be lost, and some people will not have Word available as a viewer.

Word documents are often converted to HTML. HTML can be viewed by browsers the world over. Unfortunately, current versions of HTML can only represent equations and vector graphics as GIF images (a popular raster file format). Further more, HTML does not maintain the page layout of the original document. We could use PostScript, which will enable users to download and print the document. Unfortunately users may not be able to edit the document and not all PostScript will print to all printers. Adobe has stepped in with Portable Document Format (PDF). At least with PDF, you can view the document on the screen and print it to all printers (as long as you have Adobe Acrobat). The problem however is that the user may still not be able to edit the PDF document.

The above example is designed to show the rationale for a wide variety of different formats being present on the web server. Having to have a different helper application for decoding each of these formats is cumbersome. Further, having to have so many copies of the same content in different formats is wasteful.

(B-heading) The Java Model

The Java model is able to fix some of the problems with the HTML model. The Java model has yet to gain full acceptance.

In Java, compiled byte-codes are stored in *class* files. *Class* files are files with a *.class* suffix. The class files are downloaded to the client's class loader. After a verification phase, the Java Virtual Machine (JVM) will interpret the byte codes. The role of the Java compiler is shown in Figure 1.7.

Any browser with a JVM is able to load data decoders on demand. Imagine that you have a new image sequence compression scheme based on head-and-shoulders video. Nobody has your algorithm for decoding this new image format.

Figure 1.7 The role of the Java Compiler

With Java, an algorithm for decoding a new data format may be downloaded on-demand. This means that the web has become object oriented in the sense that both the data and the program needed to manipulate the data may be joined. The Java model is a vast improvement over the current state-of-affairs, which requires that we have a wide variety of decoders on our hard-drives. The role of the Java model on the network is shown in Figure 1.8.

Figure 1.8. The role of Java on the network.**(A-heading) The Java Developer Environments**

As of this writing, there are several alternatives available for the development of Java. These include Sun products (Java Workshop, Java Developers Kit (JDK)), Metrowerks CodeWarrior, Symantec products (Visual Café, Visual Café Pro, Café), Natural Intelligence Roaster, Microsoft J++, Asymetrix SuperCede and others.

See <http://www.javasoft.com/products/JDK/>, <ftp://ftp.metrowerks.com/pub/>, <http://www.metrowerks.com/>, <http://www.symantec.com>, <http://www.roaster.com/roaster/> and <http://www.microsoft.com/java/> for more details.

These products vary in quality, price, platform and availability. For example, J++ is available only for Windows 95/NT. Workshop is available only for Solaris and Windows 95/NT. Roaster is available only for MacOS. Symantec products and CodeWarrior are available for MacOS and Windows 95/NT. The SuperCede product is available only for Windows 95/NT. JDK is one of the few products available on all platforms (MacOS, Windows 95/NT and Solaris). We will cover a few of these products in the following sections.

Feedback from students, and the authors' personal experience, has led to the conclusion that a good compiler is a very worthwhile investment. Free tools (like JDK) are good to have around too, no question about it! Some of the more expensive products are, however, much better than the JDK (being both easier to use and faster to run). Also, plan to purchase more than one compiler. Many compilers out today still have several bugs (e.g., the J++ compiler).

Some programmers have code which they would like to use that has been written in C. As far as we know, the only compiler which supports linking to C is Metrowerks' CodeWarrior. Also, CodeWarrior comes with C, C++, Pascal and Java. CodeWarrior is the only IDE we know of that directly supports native method programming.

(B-heading) Getting Started on the Mac with CodeWarrior

(CD-ROM) The CD-ROM which comes with this book has several example tools. The reader is encouraged to try them out. The software of the book is called *DiffCAD*.

DiffCAD is a Java application which resides in several files. Upon opening the CD, the Mac user should look into the Mac folder to find a self-extracting archive, like the one shown in Figure 1.9. (END CD-ROM)

Figure 1.9. The DiffCAD self extracting archive
When you double-click on the DiffCAD.sea icon, you will be prompted with a standard-file-save dialog box. This is shown in Figure 1.10.

Figure 1.10. The Standard File Dialog Box

Once the self-extracting archive is expanded into the DiffCAD folder, you should double-click on the DiffCAD folder and find the CodeWarrior folder, shown in Figure 1.11.

Figure 1.11. The CodeWarrior Folder

Within the CodeWarrior folder is a project, prebuilt with CodeWarrior 11 (the current version). If you double-click on the CodeWarrior folder, you will find a project file called CW.11, shown in Figure 1.12.

Figure 1.12. The CodeWarrior Project.

Double-click on the CW.11 project file and you will start the CodeWarrior IDE (Integrated Development Environment). Upon launch, you will be presented with a project window, shown in Figure 1.13

Figure 1.13. The Project Window


After the project window is displayed, you should be able to type *moth-r* (on the Apple keyboard, the moth key has an  logo, as well as an icon which looks like a moth). After the project begins to run, you will be presented with a display which looks like the one shown in Figure 1.14.

Figure 1.14. Screen Shot of the DiffCAD program.

The DiffCAD is a custom program upon which the book is based. DiffCAD is a Java program which has been used by industry to help design diffraction rangefinders.

(B-heading) Getting Started on Windows 95/NT with Metrowerks CodeWarrior

As of this writing, Metrowerks has DR1 (Developer Release 1) of an IDE for the Windows 95/NT environment. It comes with Pascal, C, C++ and Java. This first release required a patch to work. After the patch is applied, we found that the “batch” files created by the IDE were defective. The Metrowerks technical support is “aware” of the problem. They indicated that the batch files could be modified by hand to permit the IDE to begin working. After the modifications, we found the tool was able to compile the DiffCAD program.

(CD-ROM) The project on the CD is in a file called *CW-WIN.ZIP*. When uncompressed, the file will place the DiffCAD folder on your hard drive. Double click on the DiffCAD CodeWarrior project file, *DiffCAD.cwp*, and the project window should appear. This is shown in Figure 1.15. (END CD-ROM)

Figure 1.15. Metrowerks CodeWarrior DiffCAD Project on Windows 95.
There are three points which work the CodeWarrior IDE’s favor:

1. CodeWarrior is the only IDE for Windows 95/NT that is also able to compile Pascal, C, C++ and Java.
2. CodeWarrior IDE for Windows 95/NT has the same look and feel as the MacOS version.
3. Metrowerks states that they intend to release a version of CodeWarrior for Solaris, the Sun Unix operating system.

If the Metrowerks plan is completed, the CodeWarrior IDE would work on the Solaris, MacOS and Windows 95/NT platforms. Our endorsement for CodeWarrior running under Windows 95/NT comes with the condition that the release be later than the DR1 release.

This is primarily due to the operational bugs in the program (which we were able to work around). The DR2 release is due in April of 1997, and this is well after the book is out, so it is unlikely that the reader will be using a DR1 release.

(B-heading) Getting Started on Windows 95 with J++

In this section we give instructions on compiling the DiffCAD project on a Intel 486DX2 - 66MHz computer, running Windows NT 4.0 OS, and Microsoft Visual J++ Professional Edition.

(CD-ROM) Using an application called *WinZip*, we open and uncompress the *DIFFCAD.ZIP* file on the CD-ROM. This is shown in Figure 1.16. (END CD-ROM)

Figure 1.16. WinZip 6.2 Showing a list of files within DIFFCAD.ZIP. WinZip was used to expand the files into the D:\work\DiffCAD directory. J++ provides a means to insert files into a project. All of the files in the DiffCAD folder must be inserted into the J++ project.

Figure 1.17. Creating a new project workspace in J++
Create a new project workspace of type “Java Workspace” in the DiffCAD directory. This is shown in Figure 1.17. From the *Insert* menu, select “Files into Project...” and add all of the files in the DiffCAD directory to the project. This is shown in Figure 1.18.

Figure 1.18. Inserting the files in the *util* package into the project
This must be done in several passes as there are too many files to add in one pass (a limitation of this version of Developer Studio). Add the source files in the *Raytracer*, and *util* directories to the project.
From the *Build ... Settings* menu selection, select the “Java” tab and set the warning level to none. Build and execute the project.
There are two subprojects assigned to the main DiffCAD project : *util* and *Raytrace*.

Figure 1.19. Starting the `jview.exe` on the MAIN class.

Figure 1.19 shows the `jview.exe` being invoked automatically by the J++ project manager. There were some problems with J++ version 1.0, and this is to be expected with a 1.0 product. We found that local scoping of variables within a method did not work properly. This is a language feature that is a part of the Java language specification and we had to rewrite our code to work around this bug. This indicates that the compiler is not fully compliant with the language specification, a problem we hold as serious.

(B-heading) Getting Started on the Mac with JDK1.02

(CD-ROM) In this section we describe the use of the Java Developers Kit, JDK 1.02, located on the CD-ROM. The reader is cautioned to try a different product, as this is freeware and is missing key features of some of the commercial products. (END CD-ROM)

First double-click on the `JDK.sea` icon and expand it to your hard drive. You will create a folder which looks like the one shown in Figure 1.20.

Figure 1.20. The expanded JDK on disk.

Inside the JDK folder you will find the *Java Compiler*, along with the 4 other items shown in Figure 1.21.

Figure 1.21. The contents of the JDK folder
Make an alias to the Java compiler and place it on your desktop (this will make it easier to use). The Java compiler has a drag-and-drop interface, having an alias will be of direct assistance in its use. Figure 1.22 shows how to find the alias command on the Mac.

Figure 1.22. Finding the alias command on the Mac.
Once the Java compiler is on the desk-top, Uncompress the DiffCAD directory. An image of the DiffCAD self extracting archive is shown in Figure 1.9.

Your first step is to compile the packages in the DiffCAD directory. There is no support for a UNIX-like Makefile in the JDK version of the MAC-OS, and so this must be done by hand. Find the two folders in the DiffCAD directory, Raytracer and util. These are shown in Figure 1.23.

Figure 1.23. The two folders in the DiffCAD directory. Double-click on the Raytracer folder and drag the Java source code to the Javac Compiler alias. Package compilation will create a Raytracer class folder near the Raytracer.java file, shown in Figure 1.24.

Figure 1.24. The Raytracer class folder. Move the class files from the Raytracer class folder into the parent folder (also called Raytracer). Repeat this for the util package. Then, from the DiffCAD folder, select all the source files and drag them to the Java Compiler alias. You may find that you run out of memory during the compile. This is normal. Just view the files by name and drag the files with no corresponding class file, a few at a time, onto the Javac Compiler alias. This is time consuming and error prone. If you miss a file, your program will fail to load (a compelling reason not to use JDK on MacOS!). After all the files are loaded, launch the Applet Viewer in the JDK 1.0.2/Applets directory. The icon for the Applet Viewer is shown in Figure 1.25.

Figure 1.25. The icon of the Applet Viewer

Select the file:properties in the Applet Viewer and set the class access to unrestricted, as shown in Figure 1.26.

Figure 1.26. Class access must be unrestricted

The unrestricted setting will enable DiffCAD to perform otherwise forbidden operations, like opening a file and saving changes. The default class access is restricted and this will cause a security manager exception to be thrown during normal DiffCAD usage. As of this writing, JDK 1.0.2 cannot turn off its restrictions on class access. This means that files cannot be opened with the MacOS version of the JDK and so the version 1.0.2 of the JDK is not recommended for MacOS. It is our hope that this will change in future version of the JDK.

(B-heading) Getting Started on the Mac with Symantec Café

In this section we describe a product called Symantec Café 1.5 for the PowerPC/MacOS (current release). The Integrated Development Environment (IDE) consistently emitted an

Error: `java.lang.OutOfMemoryError`
condition. This error occurs, despite the allocation of 26 MB to the IDE. The IDE supports the Sun Java compiler as well as the Symantec compiler. Switching to the Sun Java compiler within the Symantec Café caused the program to hang.

As a result we cannot recommend Symantec Café for this books software (which currently resides in over 100 Java source files).

(A-heading) Summary

In this chapter we compared Javas' good points with its bad points. Java is probably a successful technology because it specifies a Virtual Machine. The Java

language depends upon a tightly integrated Virtual Machine, and probably could not exist without it. Once the Java language and Virtual Machine were specified, flexibility depended on the Java class libraries. As soon as we transgress the boundary of the class libraries and virtual machine, Java becomes unsafe and non-portable. Java's growth depends on a growing set of portable class libraries. For those elements which are not portable (like serial port support) the programming community must depend on Sun for API growth. This is quite a drawback!

This chapter also looked into the Java model as a means of creating decoders for web browsers. Again, the Java class libraries lacked the richness needed to support a wide variety of formats. Further, Sun was cited for not advertising the API needed to manipulate an audio data stream. The Java language has so many demands, Sun cannot possibly support them all!

In this chapter we reviewed a few tools for developing the software in this book. It seems that some products represent immature technology, as of this writing. The reader is cautioned to try these products before buying them (if possible). Some tools, like Metrowerks on the Mac, were a pleasure to use, but only on the Mac. On the Window 95/NT systems, we were able to review the J++, CodeWarrior and JDK. The DR1 of CodeWarrior had problems, but we were able to work around them. As of this writing we favor CodeWarrior on both the Mac and the Window 95/NT platforms. The J++ version 1.0 from Microsoft seems to work OK on the Windows 95/NT platforms, but there are bugs in the compiler that we consider serious.

(WARNING) All software manufacturers appear to have a phone jail system. This can leave you on hold for long periods of time. Thus, we prefer e-mail technical support. Sun, Natural Intelligence, Symantec and Microsoft did not respond to our e-mail requests. They may yet respond, but it has been a month. (END WARNING)

(NOTE) Metrowerks is the hands-down winner. Metrowerks has also said that it will be developing an IDE for the Sun workstation. If this occurs, and if Metrowerks can iron out its Windows problems, it may rise as a premier development tool for all the major languages. On a final note, the Metrowerks technical support is first-rate. They have helped us with some tricky requests and generally respond within 24 hours to e-mail. (END NOTE)

This chapter showed the basic attributes of Java; that Java has no header files, macros, pointers, multiple inheritance, integer arguments to conditionals, structures, union, or operator overloading. The language is portable, provides built-in garbage collection, a GUI library, array index checking, security features, threading, exception handling and relies on the IEEE 754-1985 floating point specification.

(SHORTCUT) One of the big early drawbacks, that the byte-code interpreter is slow, is currently being addressed. The JIT compilers blow away the byte-code interpreters. It may be the case that byte-code interpreters are better for embedded systems, due to a smaller RAM foot print. Even so, on desktop development systems, the JIT compiler technology is a break through. (END SHORTCUT)

The language has applet/application confusion, mixed mode array declaration, confusing operator overloading for strings, fragile base classes, an inability to call C++, an impoverished API and is not a pure object-oriented language. The API is generally improving, but the other drawbacks may be hard to fix.

No hype, and no hyperbole; Java is a tool, and like any tool, it is really only useful for some applications.

Is Java suitable as a first programming language? As a teaching tool, we could say that Java is better than C or C++, but that would damn Java with faint praise! Many first courses in programming are taught with C or C++ and this is almost certainly because of industrial demands. Now that industry appears willing to accept Java, introductory courses are switching to Java in mass.

As a tool for delivering cross-platform software, Java dominates any other technology that we have seen. When we attempted to write portable programs in the past, we attempted to use only the features supported by ANSI standards. Even under these circumstances, the porting of GUI based code was

tricky, at best. DiffCAD, a program with 155 class files, ported to Java virtual machines running under MacOS, Solaris and Windows 95/NT, without recompilation.

Is Java suitable for engineering? That really depends on the application. At this writing (March 1997) there is no support for serial ports. If the application involves data acquisition and processing via unsupported hardware (like the serial ports, or the video digitization cards) then the answer could be no. Writing drivers for these devices, with interfaces to Java programs is not easy. Perhaps after native methods become easier to write, this will change.

Can Java be used throughout the computer science and engineering curriculum? Probably not yet. There are few college-level textbooks for Java able to target specific courses in the computer science and engineering curricula. We see a market for new books here.

Is Java suitable for writing large programs? The problem is the fragile base class problem. Imagine if an include file, like `<stdio.h>` had to be changed. Makefiles across the UNIX operating system would have to recompile a large percentage of the code. Perhaps if developed libraries can be held stable during the course of development, then yes, the writing of large programs can occur. Their maintenance, however, may prove impractical.

What are the drawbacks in being an early adopter of the Java technology? How can we begin on a trek into a new technological frontier without being willing to invest blood, sweat and code? Programming is a humbling process. We have been writing code for many years. Programming in Java has meant recoding any of our previous work that we wanted to use. It also meant having to deal with buggy beta compilers and many long and frustrating hours trying to get the answers to simple questions. It meant being the first on the block programming in a new language. In some sense, this may isolate workers from each other. We have seen some in industry take our Java courses and attempt to transfer the technology back to the company. Management can be slow on its feet, and some people are resistant to change. Perhaps the greatest drawback of being an early adopter of Java is the drawback of being a boat-rocker. Go ahead and rock away!

In this chapter we saw coverage of the Java model vs. the HTML model. The basic question is: "which will win?". As of this writing, the jury is out. Few, if any decoders exist in pure Java. There are several reasons for this. Foremost is that the Java API is impoverished. It is able to read only one audio format (monophonic, 8 KHz, 8-bit, Sun AU files). This is a voice grade audio file with about 48 dB of signal-to-noise ratio (SNR) and a maximum frequency response of 4 KHz. Compact disc recordings (CDs) are typically stereo 44.1 KHz with 16 bit samples (22.05 KHz cutoff with 96 dB SNR). The Java API does not have a method for saving AU files. Further, Sun does not advertise an API for manipulating

the data stream that results from reading the AU files (something which this book addresses).

The Java API, as it currently stands, cannot process image sequences. The Sun API does not have a method for saving image files, something else which this book addresses. In fact, Sun has stated that there will be a multimedia API in the future, but has not said, as of this writing, when. We have had to write a great deal of support code to make up for the areas where the API is currently lacking.

In this chapter we reviewed a few tools and their use for the purpose of developing the software in this book. Some tools, like Metrowerks were a pleasure to use. On the Window 95/NT systems, we were able to review the J++, CodeWarrior and JDK. The J++ version 1.0 from Microsoft seems to be buggy. The JDK interface appears to be command line based, and was judged to be generally poor for general development work. CodeWarrior IDE is the hands-down winner. It is the only IDE that permits Windows and MacOS development using several languages. Metrowerks has also said that it will be developing an IDE for the Sun workstation. If this occurs CodeWarrior may rise as a premier development tool for all the major languages. The latest version of this tool summary is at <http://lyon.bridgeport.edu/>

(CN) 2. (CT) Java programming—the basics

“I don’t know what the programming language of the year 2000 will look like, but I know that it will be named FORTRAN” – Tony Hoare, 1984

This chapter introduces the reader to Java. We assume that the reader is a strong programmer, in some structured language (like C or Pascal) or that the reader has had some background in object-oriented programming. We divide the subject of the Java language into two parts, the syntax and the semantics. The material presentation of Java takes a different path from that presented in the Java specification. The Java specification presents the material organized by package. We present the material organized by concept. Thus, we place the wrapper classes (Boolean, Character, Integer, Long, Float and Double) into a single wrapper class subsection that resides in the data-types subsection. We have found that the package, `java.lang`, does not present functions for maximum clarity of organization. For example, we prefer not to present threads and integers in the same subsection, as these are basically unrelated topics, though they reside in the same package.

This chapter is divided up into four main sections.

The first section, describes the MBNF notation. To assist us with the presentation of the syntax of the Java language, we have devised a language for describing grammar. The grammar language is a language for describing languages and so is called a *meta-language*. We base our meta-language on a Backus Naur Form, with some modifications. Therefore we refer to our version of the Backus Naur Form as Modified Backus Naur Form (MBNF).

The semantics of the Java language are discussed after the MBNF is presented. Semantics are shown by example. Often we will present several incarnations of in-context Java usage. These incarnations are taken from working programs.

The second section, covers the simple syntax of Java. Simple syntax includes those structures that are already familiar to the C or C++ programmer. You may feel comfortable skipping or skimming this section.

In the third section, we detail the data types in Java. The coverage includes both the object oriented and the non-object oriented data types of Java.

The fourth section covers threads and provides several examples of multi-threaded Java programs.

(A-heading) MBNF Notation

In this book we use MBNF (Modified Backus Naur Form) to describe the syntax of Java. We use the translate functor of Prolog, ‘->’, as the digraphic symbol meaning, “can be written as” [Clocksin]. People who study formal languages will refer to the ‘->’ symbol as a production. Java does not use the ‘->’ symbol, since it has no pointers. We have found that the standard notation, as used in the Java Specification, to be hard to match on the blackboard during lectures. We wanted a compact notation that would have a typeset appearance that does not deviate significantly from the hand-written appearance. This meant no bold or italic font could be used in the syntax definition.

The meta-symbols of BNF are:

<u>Meta-Symbol</u>	<u>Meaning</u>
->	can be written as
(X Y)	grouping alternatively, X or Y
< >	syntactic construct, non-terminal symbol meta identifier
[X]	0 or 1 instance of X
{X}	0 or more instances of X
""	terminal
.	end of production

As an example, we show MBNF in MBNF:

```

syntax      -> { production }.
production -> identifier "->" expression "." .
expression -> term { "|" term } .
term        -> factor { factor } .
factor      -> identifier |
              quotedSymbol |
              "(" expression ")" |

```

```

"[" expression "]" |
  "{" expression "}" .
identifier -> letter { letter | digit } .
quotedSymbol -> "\"" { anyCharacter } "\"" .

```

The syntax of a language will permit the formulation of a statement that compiles.

However, the syntax does not describe the meaning of the statement, nor does it describe the common usage. For this, we use examples and prose.

The MBNF rules of Java follow. These are used in the following sections and chapters.

They will, when used, always refer back to their numbers. Sometimes the MBNF will be refined beyond the basic MBNF given here. When this occurs, it will not be preceded by the numeration.

```

compilationUnit ->
  [packageStatement] < importStatement > < typeDeclaration > .
packageStatement ->
  "package" packageName ";" .
importStatement ->
  "import" ( ( packageName "." "*" ";" ) |
    ( className | interfaceName ) ) ";" .
typeDeclaration ->
  [docComment] ( classDeclaration | interfaceDeclaration ) ";" .
docComment ->
  "/*" "... text ..." "*/" .
classDeclaration ->
  < modifier > "class" identifier [ "extends" className ] [ "implements"
    interfaceName < "," interfaceName > ] "{" < fieldDeclaration > "}" .
interfaceDeclaration ->
  < modifier > "interface" identifier [ "extends" interfaceName < ","
    interfaceName > ] "{" < fieldDeclaration > "}" .
fieldDeclaration ->
  ( [docComment] ( methodDeclaration | constructorDeclaration |
    variableDeclaration ) ) | staticInitializer | ";" .
method_declaration ->
  < modifier > type identifier "(" [parameterList "]" < "[" "]" > (
    statementBlock | ";" ) .
constructorDeclaration ->
  < modifier > identifier "(" [parameterList "]" statementBlock .

```



```

statementBlock ->
    "{" < statement > "}" .
variableDeclaration ->
    < modifier > type variableDeclarator < "," variableDeclarator > ";" .
variableDeclarator ->
    identifier < "[" "]" > [ "=" variableInitializer ] .
variableInitializer ->
    expression | ( "{" [variableInitializer < "," variableInitializer > [ "," ] ]
    "}" ) .
staticInitializer ->
    "static" statementBlock .
parameterList ->
    parameter < "," parameter > .
parameter ->
    type identifier < "[" "]" > .
statement ->
    variableDeclaration | ( expression ";" ) | ( statementBlock ) | (
    ifStatement ) | ( doStatement ) | ( whileStatement ) | ( for_statement ) |
    ( tryStatement ) | ( switchStatement ) | ( "synchronized" "(" expression
    ")" statement ) | ( "return" [expression] ";" ) | ( "throw" expression ";"
    ) | ( identifier ":" statement ) | ( "break" [identifier] ";" ) | (
    "continue" [identifier] ";" ) | ( ";" ) .
ifStatement ->
    "if" "(" expression ")" statement [ "else" statement ] .
doStatement ->
    "do" statement "while" "(" expression ")" ";" .
whileStatement ->
    "while" "(" expression ")" statement .
forStatement ->
    "for" "(" ( variableDeclaration | ( expression ";" ) | ";" ) [expression]
    ";" [expression] ";" ")" statement .
tryStatement ->
    "try" statement < "catch" "(" parameter ")" statement > [ "finally"
    statement ] .
switchStatement ->

```

```

"switch" "(" expression ")" "{" < ( "case" expression ":" ) | ( "default"
":" ) | statement > "}" .
expression ->
numericExpression | testingExpression |
logicalExpression | stringExpression | bitExpression | castingExpression |
creatingExpression |
literalExpression | "null" | "super" | "this" | identifier | ( "("
expression ")" ) | ( expression ( ( "(" [arglist] ")" ) | ( "[" expression
"]" ) | ( "." expression ) | ( "," expression ) | ( "instanceof" (
className | interfaceName ) ) ) ) .
numericExpression ->
( ( "-" | "++" | "--" ) expression ) |
( expression ( "++" | "--" ) ) |
( expression ( "+" | "+=" | "-"
| "-=" | "*" | "*=" | "/" | "/=" | "%" | "%=" ) expression ) .
testingExpression ->
( expression ( ">" | "<" | ">=" | "<=" | "==" | "!=" ) expression ) .
logicalExpression ->
( "!" expression ) | ( expression ( "&" | "&=" | "|" | "|=" | "^" |
"^=" | ( "&&" ) | "||=" | "%" | "%=" ) expression ) | ( expression
"?" expression ":" expression ) | "true" | "false" .
stringExpression = ( expression ( "+" | "+=" ) expression ) .
bitExpression ->
( "~" expression ) | ( expression ( ">>=" | "<<" | ">>" | ">>>" )
expression ) .
castingExpression ->
 "(" type )" expression .
creatingExpression ->
"new" ( ( className "(" [arglist] ")" ) | ( type_specifier [ "[" expression
"]" ] < "[" "]" > ) | ( "(" expression ")" ) ) .
literalExpression ->
integerLiteral | floatLiteral | string | character .
arglist ->
expression < "," expression > .

```

```

type ->
    typeSpecifier < "[" "]" > .
typeSpecifier ->
    "boolean" | "byte" | "char" | "short" | "int" | "float" | "long" |
    "double" | className | interfaceName .
modifier ->
    "public" | "private" | "protected" | "static" | "final" | "native" |
    "synchronized" | "abstract" | "threadsafe" | "transient" .
packageName ->
    identifier | ( packageName "." identifier ) .
className ->
    identifier | ( packageName "." identifier ) .
interfaceName ->
    identifier | ( packageName "." identifier ) .
integerLiteral ->
    ( ( "1..9" < "0..9" > ) | < "0..7" > |
    ( "0" "x" "0..9a..f" < "0..9a..f" > ) ) [ "l" ] .
floatLiteral ->
    ( decimalDigits "." [decimalDigits] [exponentPart] [floatTypeSuffix] ) |
    ( "." decimalDigits [exponentPart] [floatTypeSuffix] ) | ( decimalDigits
    [exponentPart] [floatTypeSuffix] ) .
decimalDigits ->
    "0..9" < "0..9" > .
exponentPart ->
    "e" [ "+" | "-" ] decimalDigits .
floatTypeSuffix ->
    "f" | "d" .
character ->
    "based on the unicode character set" .
string ->
    """" < character > """" .
identifier ->
    "a..z,$,_" < "a..z,$,_,0..9,unicode character over 00C0" > .

```

(A-heading) Simple Syntax

This section covers the aspect of Java that should be trivial to an experienced programmer. In all probability, the experienced C/C++ programmer will be tempted to skim or skip it. For the C/C++ programmer, the differences are small: documentation comments, and an extension to the for loop. You will probably do fine if you just read these two sections, then skip to data types. (NOTE) The MBNF of the last section will be reintroduced, and sometimes extended, on a gradual basis, as new syntactic features are covered.

(END NOTE)

(B-heading) Comments

There are three types of comments in Java:

1. C style comments that begin with `/*` and end with `*/`,
2. C++ style comments that begin with `//` and end at the end of a line and
3. javadoc style comments that start with `/**` and end with `*/`.

Comments in Java are just like those of C++. For example:

```
// This is a comment
/* and so is
   this
*/
```

There are good rules of style to be followed when using comments. For example, use C style comments when you have multiple lines because C++ style comments require the `“//”` in front of every line. For example:

```
/* This is a multi
   line comment
   it is easier to type
   because you don't need a
   // in front
   of every line.
*/
```

The C++ style comment is excellent for writing quick comments, since the end of the line denotes the end of the comment, you don't have to worry about the terminating */.

Javadoc comments start with a /** and end with a */. The MBNF for the doc comment is

```
docComment -> "/*" "... text ..." */
```

(CD-ROM icon) The JDK javadoc tool reads Java source code and scans for the javadoc comments. When they are encountered, javadoc emits HTML. These documentation comments are typically included before a class declaration, class member or constructor. Code will work without them, but it is good practice to include them. They are not supported by all development tools and, more specifically, are not supported by Metrowerks CodeWarrior. As a result we do not make use of the javadoc comments in this book. We have, as an alternative, built a custom documentation generator that comes on the CD-ROM that comes with this book. (end CD-ROM icon)

(B-heading) Identifiers

An identifier starts with a letter and then contains letters or digits. It may not contain a keyword. Identifiers in Java may be of unlimited length! Recall rule 47:

```
identifier ->
```

```
"a..z,$,_" < "a..z,$,_,0..9,unicode character over 00C0" > .
```

The MBNF for the keyword follows:

```
keyword -> "abstract" | "default" | "if" | "private" | "throw" | "boolean" | "do" |
"implements" | "protected" | "throws" | "break" | "double" | "import" |
"public" | "transient" | "byte" | "else" | "instanceof" | "return" | "try" | "case" |
"extends" | "int" | "short" | "void" | "catch" | "final" | "interface" | "static" |
"volatile" | "char" | "finally" | "long" | "super" | "while" | "class" | "float" |
"native" | "switch" | "const" | "for" | "new" | "synchronized" | "continue" |
"goto" | "package" | "this".
```

We were quite impressed when

```
int
theUnlimitedLengthConstraintOfThe_Java_Identifiers_can_bring_1234Interesting_Variables_into_your_code = 90;
```

compiled and ran!

We were also surprised when *goto* was listed as one of the keywords. Java does not support *goto*, but reserves the *goto* keyword so that it may not be used as an identifier. (Begin Note)

Operators are not permitted as a part of the identifier. Operators are discussed in the following section. Identifiers are used in several places in Java, as discussed in the following sections.

(End Note)

(B-heading) Operators

In Java there are assignment operators that are just like those of most other languages. There are also the shortcut operators of C and C++. A typical assignment operator with the shortcut version, common in C, C++ and Java follow:

```
i = i + 1;
i++;
```

Both versions have the same effect. The “+” operator is a binary operator, since it takes two arguments. The “++” operator is a unary operator, since it takes only one argument. Unary_operators -> “+” | “-” | “++” | “--” | “~” | “!” .

unary operators group right-to-left, so that:

```
~!i
```

means the same as:

```
~(!i).
```

The MBNF for the Java operators follows:

```
Operator -> “=” | “>” | “<” | “!” | “~” | “?” | “.” | “==” | “<=” | “>=” | “!=” | “&&” | “||” |
“++” | “--” | “+” | “-” | “*” | “/” | “&” | “|” | “^” | “%” | “<<” | “>>” | “>>>” |
“+=” | “-=” | “*=” | “/=” | “&=” | “|=” | “^=” | “%=” | “<<=”.
```

In Java, you may have either prefix unary operators:

```
++i;
```

or postfix unary operators:

```
i++;
```

Unary operators work just like their C counter-parts. The order of precedence is listed below, along with the MBNF.

postfix_operators->

```
"[]" | "." | "(" <parameters> ")" | "++" | "--" .
```

unary_operators ->

```
"+" | "-" | "++" | "--" | "~" | "!" .
```

creation_operators->

```
"new" | "(" <type> ")" .
```

multiplicative_operators->

```
"*" | "/" | "%" .
```

additive_operators ->

```
"+" | "-" .
```

shift_operators ->

<<" | ">>" | ">>>" .

relational_operators ->

```
<" | ">" | ">=" | "<=" | "instanceof" .
```

equality_operators ->

```
"==" | "!=" .
```

bitwise_AND_operator ->

```
"&" .
```

bitwise_XOR ->

```
"^" .
```

bitwise_OR ->

```
"|" .
```

logical_AND ->

```
"&&" .
```

logical_OR ->

```
"||" .
```

conditional_operator ->

```
"?:" .
```

assignment_operators ->

```
"=" | "+=" | "-=" | "*=" | "/=" | "%=" | ">>=" | "<<=" | ">>>=" | "&=" | "^=" |
|= " .
```

It is a part of the C idiom that assignment operators be augmented with other operators. We shall give some examples of the augmented operators, "+=", "-=", "*=", "/=", "%=", ">>=", "<<=", ">>>=", "&=", "^=" and "|=", as well as the condition operator, as these are the most commonly misused by our students.

For example:

```
if (theCowsComeHome) {System.out.println("moo");}
    else System.out.println("no milk for you!");
```

is the same as

```
System.out.println(
    theCowsComeHome ? "moo" : "no milk for you!");
```

So the conditional_operator requires a boolean type expression followed by an expression to return if true with a ":", and then an expression to return if false.

Some examples of the augmented operators, with their non-augmented counterparts, follow:

```
i *= 10; // is the same as:
i = i * 10;
i += 10; // is the same as:
i = i + 10;
i /= 10; // is the same as:
i = i / 10;
i >>= 3; // is the same as:
i = i >> 3; // a right shift with sign in the extension
i >>>= 9; // is the same as:
i = i >>> 9; // a right shift with zeros in the extension
```

The following code example will reverse the bits in an int. Bit reversal code is used to perform transforms such as the Fast Hartley Transform (FHT) and the Fast Fourier Transform (FFT):

```
int bitr(int j) {
    int ans = 0;
```



```

    for (int i = 0; i < nu; i++) {
        ans = (ans <<1) + (j&1);
        j = j>>1;
    }
    return ans;
}

```

The variable *nu* is declared as an int and is equal to the number of bits to be processed. It is common to replace bit shifting code above with look-up tables that group the bits into 8 or 16 bit groups. This will be discussed when FFT algorithms are introduced.

(B-heading) Flow of Control

This section describes Java's features for altering the flow of control. We note that Java is strongly typed and so *if*, *for*, and *while* statements require boolean expressions as arguments. In fact, all flow of control statements require boolean expressions, with the single exception of the *switch* statement, which permits an integer type expression.

(C-heading) Expressions

Central to the use of the flow of control is the notion of a boolean type expression. Java will not accept an expression that is of integer type, like C and C++ languages do. The following MBNF defines the expression statement in Java, along with the testing expression and the logical expression:

```

expression ->
    numericExpression | testingExpression |
    logicalExpression | stringExpression | bitExpression | castingExpression |
    creatingExpression |
    literalExpression | "null" | "super" | "this" | identifier | ( "("
    expression ")" ) | ( expression ( ( "(" [arglist] ")" ) | ( "[" expression
    "]" ) | ( "." expression ) | ( "," expression ) | ( "instanceof" (
    className | interfaceName ) ) ) ) .
numericExpression ->

```

```

( ( "-" | "++" | "--" ) expression ) |
( expression ( "++" | "--" ) ) |
( expression ( "+" | "+=" | "-"
| "-=" | "*" | "*=" | "/" | "/=" | "%" | "%=" ) expression ) .
testingExpression ->
( expression ( ">" | "<" | ">=" | "<=" | "==" | "!=" ) expression ) .
logicalExpression ->
( "!" expression ) | ( expression ( "ampersand" | "ampersand=" | "|" |
"|" | "^" | "^=" | ( "ampersand" "ampersand" ) | "||=" | "%" |
"%=" ) expression ) | ( expression "?" expression ":" expression ) |
"true" | "false" .
stringExpression = ( expression ( "+" | "+=" ) expression ) .
bitExpression ->
( "~" expression ) | ( expression ( ">>=" | "<<" | ">>" | ">>>" )
expression ) .

```

In the following sections we give examples of expressions being used to alter the flow of control.

(C-heading) If

The *if* statement in Java requires a boolean type expression

```

ifStatement ->
    "if" "(" expression ")" statement [ "else" statement ] .
    if (type.compareTo("ConstantValue") == 0) {
    if (i >= out.length) {
    if (e.target == saveSound_mi) {
    if (e.target == graphSound_mi) {

```

The *then* part of the *if* may take the shape of any valid statement, no ‘{ }’ are required:

```

    if (Math.abs(return_val) < 0.99)
        return return_val;
    else return 0;

```

An extended clause is often indented to indicate when the clause has ended. Sometimes it is good style to indicate the end of the clause with a comment. For example:

```

    if (interfaces == null) dos.writeShort(0);
    else {
        dos.writeShort(interfaces.length);

```

```

        for (int i = 0; i < interfaces.length; i++)
            dos.writeShort(
                ConstantPoolInfo.indexOf(
                    interfaces[i], constantPool));
    } // end else

```

This is particularly good to do when the beginning of a clause is off the programmers' screen. Poor use of indentation and comments can make code hard to read. Consider the following example:

```

        if(x1 < datarect.x) x1 = datarect.x;
        else
            if(x1 > datarect.x + datarect.width )
                x1 = datarect.x + datarect.width;

        if(y1 < datarect.y) y1 = datarect.y;
        else
            if(y1 > datarect.y + datarect.height )
                y1 = datarect.y + datarect.height;

```

Now compare the above code with the code below:

```

        if (x1 < datarect.x) x1 = datarect.x;
        else if (x1 > datarect.x + datarect.width )
            x1 = datarect.x + datarect.width;

        if (y1 < datarect.y) y1 = datarect.y;
        else if (y1 > datarect.y + datarect.height )
            y1 = datarect.y + datarect.height;

```

(BEGIN NOTE)

Else-if is used as a word pair. This significantly cleans up and shortens the code.

(END NOTE)

Here is another example, a long if-else chain:

```

        if(xminText.equals(e.target)) {
            xmaxText.requestFocus();
            return true;
        } else
            if(xmaxText.equals(e.target)) {
                yminText.requestFocus();
                return true;
            } else
                if(yminText.equals(e.target)) {
                    ymaxText.requestFocus();
                    return true;
                } else
                    if(ymaxText.equals(e.target)) {
                        xminText.requestFocus();
                        return true;
                    }

```

Now converted into the else-if style:

```
if(xminText.equals(e.target)) {
    xmaxText.requestFocus();
    return true;
} else if(xmaxText.equals(e.target)) {
    yminText.requestFocus();
    return true;
} else if(yminText.equals(e.target)) {
    ymaxText.requestFocus();
    return true;
} else if(ymaxText.equals(e.target)) {
    xminText.requestFocus();
    return true;
}
```

Such long dispatches are common in Java.

(C-heading) While and do statements

The syntax of the while statement is given by the following MBNF:

```
whileStatement ->
    "while" "(" expression ")" statement .
```

The while statement must have a boolean type expression, just like the if statement. Also, the expression and following statement will be evaluated as long as the boolean expression returns true. If the expression returns false, the statement will not be executed.

If the statement throws an exception, the while statement will throw one too. The while statement always evaluates the expression first, before evaluating the statement. To change this order use the 'do' statement. The MBNF for the 'do' statement is:

doStatement ->

```
"do" statement "while" "(" expression ")" ";" .
```

The do statement will always evaluate the statement at least once, before evaluating the while statement. The do-while structure of Java is just like the repeat-until structure of Pascal. It is a compile-time error to pass a non-boolean type expression to the do-while. To make an infinite loop, just make the boolean expression a constant *true*, as in:

```
while (true) {
    draw();
    try {Thread.sleep(1000);}
    catch (InterruptedException e) {}
}
```

It is common to embed assignments in the boolean expression given as an argument to the while statement. For example:

```
while ((next = tokens.nextToken()) != tokens.TT_EOF) {
```

Note the required use of the parentheses, a compilation error would result if

```
while (next = tokens.nextToken() != tokens.TT_EOF) { //BUG!
```

were written instead. This is due to the assignment operator taking lowest precedence. The "tokens.nextToken()" sets the *next* int to a value and, checks the value against the tokens.TT_EOF class variable, an action that returns a boolean.

(C-heading) Switch

The switch statement transfers control to one of several statements depending on the value of an expression. It is a compile-time error if the expression is not constant or one

of char, byte, short, or int. The following is the MBNF for the switch statement:

```
switchStatement ->
    "switch" "(" expression ")" "{" < ( "case" expression ":" ) | ( "default"
    ":" ) | statement > "}" .
```

The use of the switch statement generally involves a *Break*.. For example:

```
switch (expression) {
    case const_1:
        statement1;
        break;
    case const_2:
        statement2;
        break;
    default:
        statement3;
        break;
}
```

If you leave out the *break* any remaining branches are executed. There may only be one default branch in the switch statement. It is typical in Java to have long switch statements when dispatching keyboard events. For example:

```
public boolean keyDown(Event e, int key) {
    switch (key) {
        case 'o':
            openAudioStream();
            return true;
        case 'v':
            saveAs();
            return true;
        case 'p':
            play();
            return true;
        case 'n':
            normalize();
            return true;
    }
}
```

```
case 'm':
    am();
    return true;
case 'f':
    fm();
    return true;
case '^':
    sawWave();
    return true;
case 'e':
    ulawData=Audio.encodeUlaw(doubleData);
    return true;
case '1':
    fft();
    return true;
case '2':
    ifft();
    return true;
case '3':
    dft();
    return true;
case 't':
    timeDelay();
    return true;
case 'g':
    graphSound();
    return true;
case 'd':
    doubleData = Audio.decodeUlaw(ulawData);
    return true;
case 's':
    sineWave();
    return true;
case '[':
    squareWave();
    return true;
```

```

    case 'T':
        triangleWave();
        return true;
    case 'r':
        ulawData = Audio.reverse(ulawData);
        return true;
    case 'u':
        graphUlaw();
        return true;
}

```

Keyboard shortcuts are very popular with the more experienced users of an interface. It is unfortunate that the switch statement can only take scalar values. The same test is typically performed for menu-item instances, with lengthy code as the result.

(C-heading) For

The *for* statement is 10 times more common than the *while* statement in the DiffCAD program. The MBNF for the *for* statement follows:

```

forStatement ->
    "for" "(" ( variableDeclaration | ( expression ";" ) | ";" ) [expression]
    ";" [expression] ";" ")" statement .
variableDeclaration ->
    < modifier > type variableDeclarator < "," variableDeclarator > ";" .
variableDeclarator ->
    identifier < "[" "]" > [ "=" variableInitializer ] .
variableInitializer ->
    expression | ( "{" [variableInitializer < "," variableInitializer > [ "," ] ]
    "}" ) .

```

Typically the *for* statement takes the form:

```
for (initialization; test; update) statement.
```

The test must be a boolean type expression, or a compile-time error results. If the test is not satisfied, the statement will not execute. The statement does not have to execute, even once. Some examples of the *for* statement follow:

```

for (i=1; i < 99; i++) {
    }; // null statement
for (;;) { // an infinite loop
    if (expression) { break}
}

```



```

        // more junk here
    }
    for (int i = 99; i < 50; i++)
        System.out.println("I never printed");

```

The “,” is permitted in the initialization and increment section of the for loop. For example:

```

    for (i=0, j=10; i < 100; i++, j +=2) {
        //more stuff here
    }

```

If the initializations or update parts of the for loop throw an exception, then the for loop will throw an exception. The for loop is the only statement in Java that uses the comma as a separator.

(C-heading) Continue

The continue statement in Java aborts out of an iteration. It is a compile-time error to have a continue in something other than a while, do, or for statement. A continue statement with no label identifier proceeds to the next enclosing iteration. A continue statement with a label identifier proceeds to the next enclosing *labeled* statement. The MBNF for the continue statement follows:

```

continue_statement ->
    "continue" [identifier] ";"

```

For example:

```

foo: for (int i = 1; i < 5; i ++) {
    for (int j=1; j < 5; j++) {
        if ( i % j == 2) {
            System.out.println("continue");
            continue foo;
        }
        System.out.print(i*j + " ");
    }
    System.out.println();
}

```

The above code will output:

```
1 2 3 4
2 4 continue
3 6 9 12
4 8 12 16
```

A more typical usage is shown below

```
for (int i = 1; i < constantPool.length; i++) {
    if (constantPool[i] == null)
        continue;
    // more stuff follows
}
```

Here `continue` will proceed to the next i without finishing the rest of the loop.

The labeled `continue` appears to be less popular than the unlabeled `continue`. In DiffCAD there is not a single use of the labeled `continue`.

(C-heading) Break

The `break` statement appears much more often in the DiffCAD program than the `continue` statement (345 times vs. 15 times). The MBNF for the `break` statement is given by:

```
break_statement ->
    "break" [identifier] ";"
```

It is a compile-time error not to enclose `break` within a break target. Valid break targets are labels, `switch`, `while`, `do` or `for` statements. `Break` may be used with or without an identifier label. `Break` causes control to pass to the innermost enclosing break target. The break target completes normally. Compare this to a `continue` statement in a loop.

`Continue` will continue with the loop, `break` will break out of it.

The following example permits the `for`-loop to terminate normally for two reasons. The first is the `for`-loop expression, $n < 2 * \text{Math.PI}$, that is only tested at the top of the loop, the second is the $(i \geq \text{out.length})$ expression, that is tested at the bottom of the loop.

```
done: for (double n = 0; n < 2*Math.PI; n = n + step) {
```

```

        out[i] = in[i]* Math.sin(n);
        i++;
        if (i >= out.length) { break done;}
    }

```

In the following case, the break is removed by adding a more complex test in the for-statement.

```

    for (double n = 0;
        (n < 2*Math.PI) && (i < out.length);
        n += step, i++) {
        out[i] = in[i]* Math.sin(n);
    }

```

Sometimes the break statement is essential, like in a switch statement.

(C-heading) Return

The return statement in Java takes an optional expression. It transfers control to the invoker. The MBNF for the return statement follows:

```

return_statement ->
    "return" [expression] ";"

```

When the expression is omitted, *void* is returned.

(A-heading) Data Types

Data types in Java are divided into two basic categories, the reference types and the primitive types. The reference types consist of 3 sub-types, class types, interface types and array types. We defer discussion of interface types and array types until later. The following two subsections describe the primitive types and the class types.

(B-heading) Primitive Types

Java has several primitive types. Primitive types can store primitive values, each of which needs a specific number of bits of storage and has a specific precision.

The primitive data types of Java are sometimes also called *scalar types*. There are two kinds of scalar types, *boolean* and *numeric*. Boolean types may have two predefined values, *true* or *false*. Numeric types may be sub-classed into two sub-types, *integer* or *floating point*. There are five kinds of integer types and two kinds of floating point types. Figure 2.1 shows the taxonomy of the primitive types in Java.

Figure 2.1 Taxonomy of primitive types in Java.

Each of the primitive data types is assigned a specific amount of storage. Its value and range are shown in Figure 2.2.

Figure 2.2 Values and ranges for primitive data types.

All variables in Java are held as undefined until set.

For example, in the following code:

```
public class TrivialApplication {  
  
    public static void main(String args[]) {
```

```

        int x;
        System.out.println( x );
    }

```

a variable, x , was declared, but not initialized before being accessed. This is a compile-time error in Java and the compiler emits:

```
Error : Variable x may not have been initialized.
```

```
TrivialApplication.java line 7      System.out.println( x );
```

All integer data types (byte, short, int and long) are signed. All characters in Java are *Unicode*. Unicode is an international standard [Unicode]. The character type is defined as a 16-bit unsigned unique code values. For example:

```
char c = 'a';
```

It is possible to assign a numeric literal to a character-typed variable and a character literal to an integer variable:

```
char theChar = 48;
```

```
integer theValue = 'a';
```

All reals are stored as single precision floating point numbers, called *floats*, or double precision floating point numbers, called *doubles*. Java reals use the IEEE 754-1985 format [IEEE]. The smallest and largest positive non zero values for floats range from 1.40239846e-45 to 3.40282347e+38 (Float.MIN_VALUE and Float.MAX_VALUE). The smallest and largest positive non zero values for doubles range from 4.94065645841246544e-324 to 1.79769313486231570e+308 (Double.MIN_VALUE and Double.MAX_VALUE).

Now for a reality check. Using the Metrowerks CodeWarrior we discover the floating bugs in the Java floating point libraries:

```

class fp_error {
    public static void main(String argv[]) {
        float fmin = Float.MIN_VALUE;
        float fmax = Float.MAX_VALUE;
        double dmin = Double.MIN_VALUE;
        double dmax = Double.MAX_VALUE;
        String b = " ";
    }
}

```

```

        // bug in FP
        System.out.println(fmin + b + fmax +
            b + dmin + b + dmax);
    }
}

```

The above outputs:

```
1.4013e-45 3.40282e+38 2.22507e-308 1.79769e+308
```

Please note that $\text{round}(1.40239846e-45) = 1.4024e-45$ not $1.4013e-45$, as CodeWarrior indicated. It has output:

```
3.40282e+38 instead of 3.40282347e+38
2.22507e-308 instead of 4.94065645841246544e-324
1.79769e+308 instead of 1.79769313486231570e+308
```

Note that the $2.22507e-308$ is off from $4.94065645841246544e-324$ by 16 orders of magnitude! The problem with the output from `println` is also documented in the language specification (20.10.15). In short, it says that “it renders finite values in the same form as the `%g` format of the `printf` function in the C programming language, which can lose information because it produces at most six digits after the decimal point.” In fact, our search of the `println` source code shows the implementation depends upon a call to a `String` class conversion method called `toString`. This explains why the error exists in the `toString` method as well as in the `System.out.println`.

Finally, the wrong value for the `MIN_VALUE` variable is actually Sun's fault. If you look at the source for `Double.java` (for the Mac and Windows versions of the 1.0.2 API), it declares `MIN_VALUE` as $2.225..e-308$ instead of $4.94..e-324$:

```
public static final double MIN_VALUE =
    2.2250738585072014E-308;
```

We have verified this with cross platform testing (Solaris, Windows 95/NT and MacOS). The HTML docs, however, show the correct value (as do all books we have seen on Java)! The 1.1 sources should correct this problem.

The following program shows how to code the escape sequences into Java:

```

package stringUtilities;

public interface Char {

    char backspace = '\b';
    char horizontalTab = '\t';
    char newLine = '\n';
    char formFeed = '\f';
    char carriageReturn = '\r';
    char doubleQuote = '\"';
    char singleQuote = '\'';
    char backSlash = '\\';
    char maxOctal = '\377';
    char minOctal = '\000';
    char maxUnicode = '\uFFFF';
    char minUnicode = '\u0000';
}

```

(B-heading) Named Constants

Constants are values that remain unchanged during their life. For example

```
static final double PI = 3.14159265358979323846;
```

The term, *static* is known as a modifier. It indicates that there is to be only one incarnation of the field; PI. Final indicates that PI cannot change during the life of the PI variable. Assignment of a final field is a compile-time error. You may, if you like, restrict the visibility of the static final field by using a modifier prefix. You may perform computations with knowns on the right-hand side of the equals sign. For example:

```
private final double pi_2 = Math.PI * 2;
```

Here we see that the field, *pi_2* has a *private* visibility. This means that the *pi_2* name is not visible outside of the class (a discussion which we defer until later). Any valid type specifier may follow the final modifier. The MBNF for this is:

typeSpecifier ->

```
"boolean" | "byte" | "char" | "short" | "int" | "float" | "long" |
"double" | className | interfaceName .
```

The following are examples of the final modifier:

```
static final int HORIZONTAL = 0;
static final String version = "1.0";
public static final int ACC_PUBLIC = 0x1;
static final float twoPI = (float) (2*Math.PI);
public static final char NOT_CODE = (char)12;
private static final long UNIT = 1000;
```

Note the use of the "0x1". This is used to denote hexadecimal code. Also note the use of the cast operators "(float)" and "(char)". Casting is run-time type conversion. Casting is discussed in more detail later in the chapter.

(B-heading) Classes

This section introduces a reference type known as a class. A class is a combination of code (methods) and data (variables) joined together into a single entity. A class is essentially a description of how to make an instance of an object. This is what makes Java object oriented. Every instance has a class which is used to determine how to create the instance, what variables the instance will contain, and what messages the instance will respond to. Instances of classes are of reference type. In Java a reference is just like a pointer, only no pointer arithmetic is permitted.

The following is a summary of the MBNF needed to declare a class:

```
classDeclaration ->
    < modifier > "class" identifier [ "extends" className ] [ "implements"
    interfaceName < "," interfaceName > ] "{" < fieldDeclaration > "}" .
modifier ->
    "public" | "private" | "protected" | "static" | "final" | "native" |
    "synchronized" | "abstract" | "threadsafe" | "transient" .
identifier ->
    "a..z,$,_" < "a..z,$,_,0..9,unicode character over 00C0" > .
className ->
    identifier | ( packageName "." identifier ) .
interfaceName ->
    identifier | ( packageName "." identifier ) .
fieldDeclaration ->
    ( [docComment] ( methodDeclaration | constructorDeclaration |
    variableDeclaration ) ) | staticInitializer | ";" .
```

In Java a class is an entity that is able to define both the data structure and the algorithm for manipulating the data structure. The class name consists of an identifier (which may be of any length), and the class may *extend* another class. In Java, classes are able to form an AKO (A-Kind-Of) taxonomy, as described in Chapter 1.

```
public class AppletFrame extends Frame { }
```

An instance of a class has a type. The *new* operator is used to make an instance of a class. For example:

```
class point {
    public double x,y;
}
```



```
point p1 = new point();
p1.x = 10;
p1.y = 11;
```

In order to gain access to the Java class libraries, the programmer must import them. This is done with an *import* statement. For example:

```
import java.awt.*;
import java.applet.Applet;
```

Import is described in more detail later in this chapter. To build on the methods and data structures of a parent class, a subclass is constructed that *extends* the parent class. For example:

```
public class AppletFrame extends Frame {
```

In this case, the *AppletFrame* class extends the *Frame* class. This means that the *AppletFrame* is a kind of *Frame*. Sometimes we would like to have a class that can never be instantiated, only extended. When this occurs, we would declare the class as *abstract*.

For example:

```
abstract class AppletUtil {
```

AppletUtil is an abstract class. A reason we might like to keep *AppletUtil* abstract is that it contains some combination of methods and fields that are to be inherited or overridden by a subclass. For example:

```
import java.applet.*;
import java.awt.*;
```

```
abstract class AppletUtil {
    static Frame appletFrame = new Frame();
```

```
    static void run ( Applet applet) {

        appletFrame.addNotify();
        appletFrame.add("Center", applet);

        appletFrame.resize(400,400);

        applet.init();
        appletFrame.show();
        applet.start();

    }
}
```

In this case, we can never make an instance of the *AppletUtil* class, because it is abstract. However, we can access the *appletFrame* variable and we can even invoke the *run* method, without ever making an instance. For example,

```
AppletUtil.run(a);
```

will run an instance of the *Applet* class, called *a*. This has introduced the basic concept of a *method*. Classes in Java have two possible members, fields and methods. For example:

```
public class IHateHelloWorldExamples {  
    public static void main(String args[]) {  
        System.out.println("hello world");  
    }  
}
```

Above we see a class called *IHateHelloWorldExamples*. A Java application, *IHateHelloWorldExamples* contains a *main* method that is invoked at run-time. This causes the "hello world" string to be printed on the console. The following, more elaborate example shows several numbered lines. The line numbers are for reference only. Lines 1 and 2 are used to import the Java class libraries.

```
1. import java.util.*;
2. import java.awt.*;
```

Line 3 shows one class subclassing (also known as extending) another. Line 4 shows a class method with no return, not even void. This class method has the same name as the class and is called the *constructor*. The constructor returns an instance of the class when *new* is invoked.

```
3. class Camera_grating_line extends Shape {
4.   Camera_grating_line() {
5.     color =
6.       Color.blue;
7.   }...
```

The *Camera_grating_line* class is a kind of *Shape*. It contains one method and no fields. The *color* field is stored in the *Shape* base class.

(C-heading) Overloaded Methods

As mentioned before, in Java there are no functions, only methods. Methods provide the algorithm for manipulating data that is stored in the class member variables. The constructor is a method. If no constructor is specified then a default constructor is provided. The default constructor takes null as an argument. The default constructor is overridden when another constructor is specified. For example:

```
class Lamp {
  boolean on;
  int Wattage;
  Lamp (int w) {
    Wattage = w;
  }
  Lamp ( ) {
    Wattage = 100;
  }
}
```

The *Lamp* constructor has been over loaded with two versions. The first version will support:

```
Lamp dim = new Lamp(40);
```

While the second version supports the constructor invocation:

```
Lamp bright = new Lamp();
```

Java requires that the methods have difference signatures. The signature of the method is determined by the number of arguments and their compile-time types.

(C-heading) Static Methods

Classes that contain static fields have only one copy (or instance) of the static member variable. The static member variable is therefore global to all instances of the class. For example:

```
import java.awt.*;
import java.applet.Applet;
class apple {
    static int numberOfApples = 0;
    apple() { numberOfApples++; }
}
```

Every time a new *apple* instance is made, the *numberOfApples* member will be incremented. Further, all instances of the *apple* class will be able to access the same *numberOfApples* field. Thus, the *numberOfApples* int is stored in only a single place in memory.

Static members are instantiated when the class is initialized. They may not throw an exception without creating a compile-time error. Also, they may not refer to variables that have not been defined. So, for example:

```
public class Lamp {
    static int wattage = voltage*current;
    static int voltage = 110;
```

```
    static int current = 1;
```

```
}
```

Results in a compile-time error:

Error : Can't make forward reference to voltage in class Lamp.

Lamp.java line 2 static int wattage = voltage*current;

On the other hand, if power is computed by a method, then no such check is performed, but perhaps it should be. For example:

```
public class Lamp {
    static int wattage = power();
    static int voltage = 110;
    static int current = 1;

    static int power() {
        return voltage * current;
    }
    public static void main(String argv[]) {
        Lamp i = new Lamp();
        System.out.println("The power is "+ i.wattage);
    }
}
```

Will print:

The power is 0

Why should the power be zero? Power is current times voltage. Both are set to non-zero values. Yes, it does compile and run. So what gives? The answer is the order of dependency. The power method is invoked first. The values for voltage and current are unset (and default to zero). Thus power is set to the proper 110 watt answer, only if both current and voltage are defined first. For example:

```
public class Lamp {

    static int voltage = 110;
    static int current = 1;
```

```
static int wattage = power();

static int power() {
    return voltage * current;
}

public static void main(String argv[]) {
    Lamp i = new Lamp();
    System.out.println("The power is "+ i.wattage);
}
```

}
Will print:

The power is 110

So there are two points we learned from this example; 1. static variables must be defined in the order of independent first, dependent second, 2. static methods cannot be used to avoid constraint 1.

(C-heading) Null

One of the literals of Java is *null*. Null is what you get when nothing has been created.

For example:

```
if (some_object != null) {
    System.out.println("Object Exists!");
}
```

Null has a null type and is the default value for any type that has not been created. For example:

```
class test {
    int i[99];
}

test foo = new test();
```

At this point, *foo.i* is equal to null. It is not until the memory is allocated for the array contained in the *foo* instance of the test class, will the *i* field be non-null. For example:

```
foo.i = new i[99];
```

Will set the *i* array, and *i* will no longer be null in value.

(C-heading) Casting

Type conversion in Java is called *casting*. When casting is performed, it is a run-time operation. Casting is able to convert only between compatible types and always results in a value, not a variable.

Sometimes the only way to know for sure when types are compatible is to run the program. If a `ClassCastException` is thrown at run-time, then the type conversion failed. It is always correct to cast an instance from a subclass to its super class. For example:

```
1. for (int i=0; i < v.size(); i++) {
2.     s = (Shape) v.elementAt(i);
3.     s.print();
4. }
```

In line 1, an instance of a `Vector`, *v* is accessed for size. The elements in the vector are accessed using line 2. Note that each element in the vector is a class that extends the `Shape` class. It is always correct to cast the subclass of the `Shape` class back into the super-class. This enables `print()` method invocation on each shape in the vector instance.

(C-heading) Subclassing and Super

One feature of the Java class is that it can intrinsically represent taxonomic structures (like those described in Chapter 1). The taxonomic structures are formed by Java classes when a sub-class *extends* a super class. This type of extension is called direct inheritance. Thus, in terms of knowledge representation, Java classes can represent the AKO (a-kind-of) relationship. In addition, Java classes can represent the has-a relationship using the class member variables. For example, we can represent the statement: “A student is a-kind-of human” by creating a student class that extends the human class. We can also

represent the statement: “The student has-a pencil” by placing a class member variable of pencil class type into the student class construct. In the following section we present the syntax of Java and its relationship to the semantics of Java.

A class may be used to provide a container for an instance variable of any primitive type. For example:

```
class Lamp {
    boolean on;
}
...
Lamp l = new Lamp ( );
l.on = true;
```

A Java class may be used to store a reference to named constants:

```
class Constants {
    static final double Pion2 = Math.PI / 2;
}
```

Notice that these class examples have no methods. When one class extends another, we are sub-classing a super class. The sub-class will inherit the member-variables, and methods, of the super-class. In the case of a name conflict, the sub-class implementation always over-rides the super-class implementation. For example:

```
class Lamp extends Constants {
    double power = 100 / Pion2; // watts
    boolean on = true;
}
```

(BEGIN NOTE)

The power in the Lamp class is set using a Pion2 constant that is inherited from the Constants class. In this case, it is not strictly correct to say that the Lamp is a-kind-of Constants and thus the *extends* is being used as a programming convenience, not a means for knowledge representation.

(END NOTE)

On the other hand:

```
class Student extends Human {
    Pencil p;
```



```

}
class Human {
    boolean bald = false;
}

```

Now we represent the statement that “Doug is a bald student with a pencil”:

```

Student doug = new Student();
doug.p = new Pencil();
doug.bald = true;

```

Super is a keyword that permits a subclass to call-upon the instance variable or method of the super class. For example:

```

1.  public class ClosableFrame extends Frame {

2.  // constructor needed to pass window title to class
    Frame
3.  public ClosableFrame(String name) {
4.      // call java.awt.Frame(String) constructor
5.      super(name);
6.  }

7.  // needed to allow window close
8.  public boolean handleEvent(Event e) {
9.      // Window Destroy event
10.     if (e.id == Event.WINDOW_DESTROY) {
11.         dispose();
12.         return true;
13.     }

14.     // it's good form to let the super class look at
    any unhandled events
15.     return super.handleEvent(e);

16. } // end handleEvent()

17. } // end class ClosableFrame

```

Line 15 of class `ClosableFrame` invokes `super.handleEvent` because the event that was passed, `e`, may not have been a `Event.WINDOW_DESTROY` event. In that case, it may be that the super class is able to decode and handle the event properly. A frame that extends the `ClosableFrame` will inherit the ability to handle the

Event.WINDOW_DESTROY events by invoking `super.handleEvent(e)`.

(C-heading) Abstract Classes and Methods

An abstract class is a class that can never be instantiated. It is a compile-time error to attempt to instantiate an abstract class. An abstract class must be extended before it can be instantiated. A class may become abstract in several ways. One way is to declare the class as abstract. Another way is to declare a method within the class as abstract. Yet another way is to extend an interface or abstract class without providing an implementation for the abstract methods in the interface or superclass.

(BEGIN NOTE)

A class should be declared abstract only if the intent is that subclasses can be created to complete the implementation.

(END NOTE)

Abstract classes are used to create super classes that have to be extended. This situation typically involves the implementation of some, but not all methods. Methods without an implementation are declared abstract. Abstract methods must be implemented by a subclass before the subclass may be instantiated. If a subclass does not implement the abstract methods, then the subclass is also an abstract class, even if it is not explicitly declared as abstract. Java supports polymorphism by casting instances into their common super classes. The super classes will typically have abstract methods that are implemented by the subclasses. For example, in DiffCAD there is an abstract class called `shape`:

```
abstract class Shape extends computation {
...
    abstract void draw(Graphics g);
...
}
```

There are many different kinds of Shapes in the DiffCAD program. They are all stored in a vector instance called *drawnShapes*. Polymorphism is performed when the shape subclass instances are accessed in the *drawnShapes* vector, cast into their Shape super class and then used as the target of the draw method. For example:

```
Shape s;
for (int i = 0; i < drawnShapes.size(); i++) {
```

```

        s = (Shape)drawnShapes.elementAt(i);
        s.draw(g);
    }

```

Shape polymorphism is possible for any of the methods (even the non-abstract ones) in the Shape class.

(C-heading) Final Classes and Methods

In contrast with abstract classing, which requires that a class be subclassed to be extended, there is final classing. A class that is declared as final may not be extended. It may be instantiated, however. Fields in a class that are declared as final become constant throughout the life of the class. (BEGIN NOTE) To prevent a final class from being instantiated, declare a single private constructor and never instance the class internally. (END NOTE) For example:

```

public final class Audio {

    // Prevent instantiation
    private Audio() {}

```

The final modifier will prevent the Audio class from being extended. The private modifier will prevent the constructor from being visible. Further, the existence of a constructor will override the default constructor. Thus, if no method within the Audio class instantiates the Audio class, the Audio class cannot be instantiated.

Final may also be used as a method modifier. This prevents the method from being overridden. For example:

```

public final boolean keyDown(Event e, int key) {
    . . .
}

```

May lead to a compiler optimization, as well as prevent any subclass from over-riding the keyDown implementation.

When final is used on a class, type checking can occur at compile time.

(C-heading) Packages

The package statement is used by the programmer to isolate the class and interface name space so that only public classes and interfaces will be used by non-package programmers. This restricts default access of classes and interfaces to package programmers.

The MBNF for packages follows:

packageStatement ->

```
"package" packageName ";" .
```

packageName ->

```
identifier | ( packageName "." identifier ) .
```

In order to build a large program, it is often to your advantage to divide it into subsections, each of which reside in a package. Import classes that are declared as public and have public constructors. For example, DiffCAD has an HTML generator. HTML stands for HyperText Markup Language and is a common file format that is read by browsers on the world wide web (WWW). The DiffCAD HTML generator reads in C, C++ or Java and outputs a colorized version of the source in HTML. The generator, which consists of 15 difference source files, has a single public class with a single public constructor. Anything that is not declared public in the package (i.e., that has private or default visibility) will not be visible, even after the import statement is issued. The error:

```
Error      : No constructor matching
             HtmlGenerator(java.lang.String) found in class
             htmlconverter.HtmlGenerator.
             process_menuitem.java line 21      new HtmlGenerator("The
             HTML Generator");
```

is emitted when

```
HtmlGenerator hg =
    new HtmlGenerator("The HTML Generator");
    hg.main(null);
```

is processed by the compiler, because the default visibility is not sufficient for the package. Only by providing the public accessor for the constructor:

```
public HtmlGenerator(String title) {
can the HtmlGenerator class be instantiated.
```

(BEGIN NOTE)

A package permits a grouping of multiple classes and interfaces into a unit that can be developed and deployed separately.

(END NOTE)

The declaration of `public` from within a class is an explicit advertisement to those who would import your package. It, in effect, says “you need this to use my package”.

When classes are written into a source file with no package declaration, they become a part of the unnamed package. Code that resides in packages should not use code in an unnamed package. Java has a compilation unit design goal. Packages are the mechanism that Java provides for creating separate compilation units.

Every class must reside in a compilation unit. An unnamed package is, in effect, an unnamed compilation unit. It is probably a Java design flaw to permit unnamed compilation units to compile.

Packaging is essential for writing large programs. Once upon a time, in a fit of quick prototyping frenzy, a programmer decided to write many classes without placing them in a package. Of course the sophisticated reader will NEVER do this! Everything worked just fine, the days were sunny and programmer was productive. Then, one day, the program got so big and intertwined that even the programmer had trouble sorting out the interdependencies. “If only packaging had been enforced, then I could have done it all along”, he thought. The sun began to set as the well-intentioned efforts to introduce packages into a large system began to fail. The programmer is still trying to grapple with this massive code block, even today. The programmer’s productivity has fallen. The sun no longer shines. He has taken to drinking lots of java. His blood pressure shot up as he became a Javaholic. This led to no-good and he soon began writing books on Java.

Moral of story: Package small, package often!

(C-heading) Imports

Import is a reserved keyword in Java that permits packages to be brought into the name space. Packages are like libraries in C. Packages are used to organize Java program libraries and to control type access. For example, a class or interface type is only visible outside of a package if it is declared as *public*. The required Java packages are `java.lang`, `java.util` and `java.io`. The MBNF for the import statement follows:

```
importStatement ->
    "import" ( ( packageName "." "*" ";" ) |
              ( className | interfaceName ) ) ";" .
```

Extra imports are discarded, but can make compilation take longer. For example:

```
import java.io.*;
import java.util.*;
```

Import all the public classes and interfaces in the java.io and java.util packages. It is possible to import a single class or interface from a package. This may prevent a name space conflict. For example:

```
import java.io.FileNameFilter;
import java.io.File;
```

Are generally the only two classes needed to build a FileNameFilter. You can build up a package by the use of the package statement.

(C-heading) Visibility

Visibility is the aspect of an identifier that permits access to the name space that contains the identifier. If an identifier is visible, it may collide with another identifier. This is called name-space contention. Further, if a method is not visible, it may not be invoked.

The MBNF for the modifier is:

modifier ->

```
"public" | "private" | "protected" | "static" | "final" | "native" |
"synchronized" | "abstract" | "threadsafe" | "transient" .
```

The visibility modifier is a subset of the consisting of

visibility_modifier ->

```
"public" | "private" | "protected" .
```

1. *public* – gives the world access to your class, method or field variable. It is required for those types you wish to advertise for use outside of your package. The public modifier applies to classes, methods and variables.
2. *protected* – gives access to the class, subclass and package. It may only be applied to methods and variables.

- 3. default – gives package access. Classes that import the package will not be able to access types that have default access. To give such access, the type must be declared as public.
- 4. *private protected* - class/subclass access. It may only be applied to methods and variables. Private protected prevents package access. *Private protected* is to be removed with JDK 1.1.
- 5. *private* – accessible only within this class. Private access prevents inheritance. It may only be applied to methods and variables.

It is idiomatic of Java to have a collection of public static methods contained in a public final class with a private constructor that is never invoked. The Math package is like this. So is the file utility package in DiffCAD, called *futil*. For example:

```
public final class futil {
    /**
     * Don't let anyone instantiate this class.
     */
    private futil() {}
}
```

	Access by non-subclass from same package	Access to subclass from same package	Access to non-subclass from different package	Access to subclass from different package	Inherited by subclass in a same package	Inherited by subclass in different package	w=
private	0	0	0	0	0	0	0
private protected	0	0	0	0	1	1	2
default	1	1	0	0	1	0	3
protected	1	1	0	0	1	1	4
public	1	1	1	1	1	1	6

Table 2.1. Visibility Matrix

Table 2.1 shows the visibility matrix for Java. A '1' in the visibility matrix indicates a true condition, a 0 indicates a false condition. The row is summed to obtain a "weight" for the visibility vector. The weight is shown in the 'w' column. For example, private gives no access outside of the class, so the weight is zero. Public gives you complete access so the weight is 6. An example of both the use, and abuse, of the visibility modifiers follows:

```
public class visible {
    public String publicString = "publicString";
    String defaultString = "defaultString";
    protected String protectedString = "protectedString";
    private protected String privateProtectedString =
"privateProtectedString";
    private String privateString = "privateString";
    // What follows is not in the Spec...it just worked
here.
    private public String privatePublicString =
"privatePublic"; //gosh!
    public private String publicPrivateString =
"publicPrivate"; //a stranger in a stranger land...
    protected public String protectedPublicString =
"protectedPublic"; //can you stand it?
    public protected String publicProtectedString =
"publicProtected"; //art is what you get away with.
    public protected private String
publicProtectedPrivateString = "publicProtectedPrivate";
}

public class invisible extends visible {
    String iGetPublicStrings = publicString;
    String iGetDefaultStrings = defaultString;
    String iGetProtectedStrings = protectedString;
    String iGetPrivateProtectedStrings =
privateProtectedString;
    // String iDontGetPrivateStrings = privateString;
    String iGetPrivatePublicStrings = privatePublicString;
    String iGetpublicPrivateStrings = publicPrivateString;
    String iGetProtectedPublicStrings =
protectedPublicString;
    String iGetPublicProtectedStrings =
publicProtectedString;
    String iGetPublicProtectedPrivateStrings =
publicProtectedPrivateString;
}
```


(BEGIN NOTE) The public modifier appears to override private, no matter what the order. (END NOTE) This is not a part of the specification for Java. It just appears to work. It is probably within the vendors right to declare a private public declaration a semantic error.

(C-heading) Interfaces

An interface is like an abstract class with only abstract methods and constant fields. The interface can hold no method implementations and is defined just like a class except that it uses the keyword *interface* rather than class. What follows is the MBNF for the interface declaration:

```
interfaceDeclaration ->
    < modifier > "interface" identifier [ "extends" interfaceName < ","
    interfaceName > ] "{" < fieldDeclaration > "}" .
fieldDeclaration ->
    ( [docComment] ( methodDeclaration | constructorDeclaration |
    variableDeclaration ) ) | staticInitializer | ";" .
interfaceName ->
    identifier | ( packageName "." identifier ) .
```

Note that an interface declaration can extend multiple interfaces. The interface can serve as another reference type, but can never be instantiated. Thus classes that implement an interface can always be cast back to the interface type.

Class identifiers and interface identifiers are always global to the package in which they reside. Therefore classes and interfaces in the same package must have different identifiers. It is redundant to declare an interface as abstract as interfaces are implicitly abstract. It is redundant to declare the field declarations as public static and final as they are implicitly public static and final. All fields must be initialized. There is no primordial interface (like the Object class is a primordial class). It is a compile-time error to have ambiguous inherited fields in an interface. For example:

```
public interface real_dumb {
    double PI = 4;
}

public interface dum_constants {
    double PI = 3;
}

public interface mixed_up_constants extends dum_constants,
real_dumb
{double foo=PI;}
Error    : Reference to PI is ambiguous. It is defined in
interface real_dumb and interface dum_constants.
constants.java line 17    {double foo=PI;}
```

The following are some correct uses of interfaces. For the first example we show how interfaces may be used to group constants together:

```
public interface constants {
    double Pi_on_180 = Math.PI / 180;
    double PI = Math.PI;
    double Pi_on_2 = Math.PI/2;
```

```
    double Pi_on_4 = Math.PI/4;  
}
```

Here is one where there is a large array of symbols being stored:

```
public interface CplusplusText {  
    public static String cplusplusReservedWords[] = {  
        "asm",  
        "auto",  
        "break",  
        "case",  
        "catch",  
        "char",  
        "class",  
        "const",  
        "continue",  
        "default",  
        "delete",  
        "do",  
        "double",  
        "else",  
        "enum",  
        "extern",
```

```
"float",  
"friend",  
"for",  
"goto",  
"if",  
"inline",  
"int",  
"long",  
"new",  
"operator",  
"private",  
"protected",  
"public",  
"register",  
"return",  
"short",  
"signed",  
"sizeof",  
"static",  
"struct",  
"switch",  
"this",  
"throw",  
"try",  
"typedef",  
"union",  
"unsigned",  
"virtual",  
"void",  
"volatile",  
"while"
```

```
};
```

In the 125 files that make up our DiffCAD program, the interface was never used to extend multiple interfaces. We find assertions in other books, that the interface gives Java a kind of multiple inheritance, to be groundless. Java has no multiple inheritance for implementations. Java has multiple inheritance of prototypes via interfaces. A good use for multiple inheritance

of prototypes is as a work-around for Java's strong typing. Strong typing constrains method invocation. We will see examples of this in the following chapter.

(B-heading) Wrapper classes

The wrapper classes are used to promote a primitive data type into a reference data type. After this promotion is accomplished, the reference type permits a series of operations on the primitive data type that were not otherwise possible. There are some things that can only be done with an instance, such as the adding of an element to a vector. Primitive values cannot be the target of method invocations. All of the wrapper types support a method invocation that converts them to a string representation. All may be constructed from a string. All support an

```
public boolean equals(Object obj)
```

to check an object for equality of value. Keep in mind that the value of a wrapper object is stored in a private class field. When comparing two strings, use equals and never ==.

For example:

```
if (arg.equals("this is right")) {...
```

will check the value of the string in arg against the value of "this is right". The following example checks the value of the arg reference with the compile-time constant string reference. This is probably not what the programmer wants:

```
if (arg == "this is wrong") {...
```

(WARNING)

Always use the equals method to check value when comparing wrapper instances.

(END WARNING)

(C-heading) Boolean

The Boolean class promotes the primitive boolean type to a reference type. Construction is overloaded to handle both the primitive boolean type and a string. The string must be equal to the word "true", ignoring case, before the Boolean instance will be true. For example:

```

Boolean b = Boolean("true");
Boolean b = Boolean("True");
Boolean b = Boolean("TRuE");

```

will all lead to *b* representing true. There are two static conversion methods available for use in the Boolean class:

```

public static Boolean valueOf(String s) - returns a Boolean
instance.
Boolean b = Boolean.valueOf("true"); // sets b to true
Boolean b = Boolean.valueOf("yes"); // sets b to false

```

(NOTE)

This different from `getBoolean`, which has as its return, a primitive boolean type. Note that sections 20.4.9 and 20.4.10 of the Java Specification appear to show both of these functions returning the primitive boolean, which is an error in the specification.

(END NOTE)

```

public static boolean getBoolean(String name)
boolean b = Boolean.getBoolean("yes"); // sets b to false

```

(C-heading) Character

The Character class is a wrapper class for the primitive Java char data type. When embedded in the Java code, the 'a' character is surrounded by single quotes, whereas strings use double quotes. Any Unicode character is permitted as an argument for the construction of a Character instance. In addition, there are methods for converting to and from Character and char. For example:

```

Character aCharacter = new Character('π');
char aChar = '@';
a = new Character(aChar);
aChar = aCharacter.charValue();

```

Neither an array of char nor an array of Character constitute a String. The Character class supports several public static methods that are useful for testing and processing char's.

All chars are Unicode and so all comparisons are written for Unicode. For example:

```

public static boolean isDigit(char ch)

```

return true if ch is a Unicode digit
 Java version 1.0.1 supports ISO-LATIN-1('0'through'9'), Arabic-Indic, ExtendedArabic-Indic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, and Lao digits.

The Character has a series of public static Char test facilities that return a boolean. For example:

```
boolean aBoolean;
char aChar = 'a';

aBoolean = Character.isDefined(aChar);
aBoolean = Character.isLowerCase(aChar);
aBoolean = Character.isUpperCase(aChar);
aBoolean = Character.isTitleCase(aChar);
aBoolean = Character.isDigit(aChar);
aBoolean = Character.isLetter(aChar);
aBoolean = Character.isLetterOrDigit(aChar);
aBoolean = Character.isJavaLetter(aChar);
aBoolean = Character.isJavaLetterOrDigit(aChar);
aBoolean = Character.isSpace(aChar);
```

The Character class also support a series of char conversions that are public and static. These may be accessed without ever making a Character instance. For example:

```
aChar = Character.toLowerCase(aChar);
aChar = Character.toUpperCase(aChar);
aChar = Character.titleCase(aChar);

// returns an 'A', hex representation of 10.
aChar = Character.forDigit(10, 16);

// returns 10, the hex value of 'A'
int digit = Character.digit(aChar, 16);
int i = Character.MIN_RADIX // i = 2;
int i = Character.MAX_RADIX // i = 36;
```

```
char aChar = Character.MIN_VALUE // aChar = '\u0000';
char aChar = Character.MAX_VALUE // aChar = '\uffff';
```

About the only thing you can do with a Character instance is

```
Character aCharacter = new Character('π');
String aString = aCharacter.toString();
aBoolean = aCharacter.equals(aCharacter);
int anInt = aCharacter.hashCode();
```

(C-heading) The numeric wrapper classes

Each of the numeric primitive types has an associated wrapper class. In the following examples let:

```
int i = 0;
long l = 0;
float f = 0;
double d = 0;
```

The name of the primitive numeric type, followed by its constructor wrapper class, is given below:

```
Integer I = new Integer(i);
Long L = new Long(l);
Float F = new Float(f);
Double D = new Double(d);
```

Each of the numeric wrapper classes supports a conversion to a string:

```
String SI = I.toString();
String SL = L.toString();
String SF = F.toString();
String SD = D.toString();
```

Each of the numeric wrapper classes supports a conversion from a string to a reference type:

```
Integer I = new Integer(SI);
```



```
Long L = new Long(SL);
Float F = new Float(SF);
Double D = new Double(SD);
```

(NOTE) Any string to numeric conversion can throw a `NumberFormatException`. If you read a string (from a user's input or file) you are strongly advised to check the input.

(END NOTE)

Each of the numeric wrapper classes supports a conversion from a string to a primitive type using a static method invocation:

```
I = Integer.valueOf(SI);
L = Long.valueOf(SL);
F = Float.valueOf(SF);
D = Double.valueOf(SD);
```

An example of the use of the static `valueOf` method follows:

```
try {value = Float.valueOf(getText()).floatValue();}
catch(NumberFormatException e) {
    value = 0;
}
```

The scalar numeric wrappers (`Integer` and `Long`) support an overloaded `valueOf` and `toString` method. The `toString` method will yield a string in any radix between 2 and 36, inclusive. For example

```
SI = Integer.toString(i, 2); // yields a binary string
SL = Long.toString(l, 16); // yields a hex string
I = Integer.valueOf(SI,2); // converts from binary string
L = Long.valueOf(SL,16); // converts from hex string
```

To perform string conversions to primitive types, there are a static methods called `parseInt` and `parseLong`:

```
i = Integer.parseInt(SI); // converts from decimal string
i = Integer.parseInt(SI,2); // converts from binary string
```

```
l = Long.parseLong(SL,16); // converts from hex string
```

(B-heading) Strings

In Java's `java.lang` package there is a class known as `String`. `String` instances in Java cannot be changed once they are created. The reason for this is that once created, a string instance is added to an internal string pool. The pool is a private `HashTable` instance that speeds string lookup operations. There is no public or private method (that we could find) for removal of a `String` instance from the internal hash table. Thus, once created strings remain unchanged for the life of the virtual machine.

The `String` class supports a series of public static methods that may be accessed without ever making an instance of the string class. Suppose the following constants are pre-defined:

```
boolean aBoolean;
char achar;
int anInt;
long aLong;
float aFloat;
double aDouble;
Object anObject;
String aString;
char[] aCharArray;
int arrayOffset, numberOfElementsToConvert;
int endIndex; // last position in array - 1;
StringBuffer aStringBuffer = new StringBuffer(100);
Byte [] ASCIIByteArray;
int hiByte;
```

The following examples show how to use the `String` class, given the above, pre-defined contents:

```
aString = String.valueOf(anObject);
aString = String.valueOf(aCharArray);
aString = String.valueOf(aCharArray, arrayOffset,
    numberOfElementsToConvert);
aString = String.valueOf(aBoolean);
aString = String.valueOf(aChar);
```

```

aString = String.valueOf(anInt);
aString = String.valueOf(aLong);
aString = String.valueOf(aFloat);
aString = String.valueOf(aDouble);

```

The String class constructor is overloaded so that:

```

aString = new String();
aString = new String(anotherString);
aString = new String(aStringBuffer);
aString = new String(aCharArray);
aString = new String(aCharArray, arrayOffset,
    numberOfElementsToConvert);

```

The following examples show how to build aString with hiByte in most significant 8 bits, and ASCIIByteArray elements in least significant 8 bits (the hiByte is typically 0):

```

aString = new String(ASCIIByteArray, hiByte);
aString = new String(ASCIIByteArray, hiByte,
    arrayOffset, numberOfElementsToConvert);

```

The String class has a series of methods that use a string instance as a target. These method typically involve extraction or conversion. For example:

```

aString = aString.toString();
aString = aString.substring(arrayOffset);
aString = aString.substring(arrayOffset, endIndex);
aString = aString.concat(anotherString)
//replace oldChar with aChar in aString
aString = aString.replace(oldChar, aChar);
aString = aString.toLowerCase();
aString = aString.toUpperCase();
// Trims leading and trailing whitespace
aString = aString.trim();
// intern returns a string from the private internal
// hash table set that will pass the
// aString == anotherString test

```

```

aString = anotherString.intern();

aBoolean = aString.equals(anObject);
anInt = aString.hashCode();
anInt = aString.length();
aChar = astring.charAt(anInt);
// get charArray from aString
aString.getChars(beginIndex, endIndex,
    charArray, arrayOffset);
aString.getBytes(beginIndex, endIndex,
    byteArray, arrayOffset)
charArray = aString.toCharArray();
aBoolean = aString.equalsIgnoreCase(anotherString);
CompareTo is a method that returns -1, +1 or 0. The string comparison is performed
using the underlying Unicode characters.
anInt = aString.compareTo(anotherString)
aBoolean = aString.regionMatches(beginIndex, anotherString,
    arrayOffset, numberOfElementsToConvert)
aBoolean = aString.regionMatches(
    boolean ignoreCase, arrayOffset,
    anotherString, arrayOffset, numberOfElementsToConvert)
aBoolean = aString.startsWith(aString)
aBoolean = aString.startsWith(aString, arrayOffset)
aBoolean = aString.endsWith(aString)
There are some string search functions in the String class. When they fail they return -1.
// first location of aChar in aString
anInt = aString.indexOf(aChar);
// first location of aChar in aString starting from anInt
anInt = aString.indexOf(aChar, anInt);
// first location of aString in anotherString
anInt = anotherString.indexOf(aString);
// first location of aString in
// anotherString starting from anInt
anInt = anotherString.indexOf(aString, anInt);

```

```
// Find the last occurrence of a string or char in a string
// starting from anInt
anInt = aString.lastIndexOf(aChar);
anInt = aString.lastIndexOf(aChar, anInt);
anInt = aString.lastIndexOf(aString)
anInt = aString.lastIndexOf(aString, anInt);
```

(B-heading) Arrays

An instance of an array is a reference-type instance. There is a variable that may be accessed, but not set, called *length*. The length indicates the number of elements in the array. Array elements start at element number zero and end at element number length - 1. Any access beyond the end of the length of an array results in the `ArrayIndexOutOfBoundsException` being thrown. The method, *arraycopy* will copy values between arrays of the same type.

An array may hold any data type, but every element in the array must hold the same data type. Arrays have fixed length and may be dynamically allocated. It is possible to make deep arrays. For example:

```
int deep_array[] [] [] [] [] [] [] [] [] [] [] = new
int [2] [2] [2] [2] [2] [2] [2] [2] [2] [2] ;
int sum = 0;
for (int i=0; i<2; i++) {
    deep_array[i] [i] [i] [i] [i] [i] [i] [i] [i] [i]
[i] = 1;
    sum += deep_array[i] [i] [i] [i] [i] [i] [i] [i] [i]
[i] [i] [i];
    System.out.println("sum =" + sum);
}
sum =1
sum =2
```

As long as there is enough memory. As pointed out in chapter 1, there both Java style arrays and C/C++ style arrays.

(B-heading) Vectors

The `Vector` is a container class that resides in the `java.util` package. An instance of the `Vector` class may be expanded dynamically. A container class is one that is designed to store reference data types. Vectors are an excellent choice for implementing data structures of variable length. The `Vector` class resides in the `java.util` package. It must be imported before using. Vectors can hold any combination of objects. For example:

```
Vector v = new Vector();
Rectangle a = new Rectangle(10,20);
v.addElement(A);
```

Vectors cannot hold `int`, `float`, `char`, `byte` or any of the non-reference data types. The following example shows a series of different data types all being added to the a `Vector` instance:

```
static public void add_elements(Vector v) {
    for (int i=0; i < number_of_rays.getValue(); i++) {
        v.addElement(grating_target_line[i]);
        v.addElement(camera_grating_line[i]);
    }
}
```

```

    }
    v.addElement(camera);
    v.addElement(grating);
    v.addElement(wedge);
    v.addElement(laser);
    v.addElement(laser_wedge_line);
} // add_elements

```

Having a collection of different instances in a Vector instance is a good way to implement polymorphism. With polymorphism, you can cast all the instances stored in the Vector instance into a common super-class that supports a common method. In the example below, drawShapes is a Vector that contains several different instances. All are subclasses of the Shape class. Thus, all can be cast into the Shape type. Also, the Shape class has an abstract draw method. Thus, the draw method may be invoked on any subclass of the Shape class, provided it has been properly cast. For example:

```

for (int i = 0; i < drawnShapes.size(); i++) {
    s = (Shape)drawnShapes.elementAt(i);
    s.draw(g);
}

```

(B-heading) Exceptions

The grammar of a language allows the formulation of code that will compile. However, after compile-time, comes run-time. The Java language specification covers both the syntax (compile-time) and the semantics (run-time) behavior of the Java environment. Thus, the specification goes beyond the compiler and describes the behavior of the Java machine. The Java machine will throw an exception under special run-time conditions. This throw is a non-local transfer of the flow of control from the area that generated the exception to a place that can *catch* the exception.

The MBNF of the *try* statement follows:

22. tryStatement ->
 "try" statement < "catch" "(" parameter ")" statement > ["finally"
 statement] .

Typically, a statement that can possibly fail is surrounded with the try and catch keywords. This enables failure in the statement following the try to be intercepted and handled. For example:

```

try {
    // create an instance of your applet class
    a = (Applet)
Class.forName(className).newInstance();
    } catch (ClassNotFoundException e) {
        System.out.println("ClassNotFoundException in
AppletFrame");
        return;
    } catch (InstantiationException e) {
        System.out.println("InstantiationException in
AppletFrame");
        return;
    }

```

```
        } catch (IllegalAccessException e) {  
            System.out.println("IllegalAccessException in  
AppletFrame");  
            return;  
        }  
    }
```

Sometimes no error is to be generated at all, we simply want the program to continue running. For example:

```
    try {  
        out.write(buffer);  
    } catch (Exception e) { }
```

The programmer can make up exception names at will, just by extending the exception class. For example:

```
class FileFormatException extends Exception {  
    public FileFormatException(String s) {  
        super(s);  
    }  
}
```

An overview of the java.lang exception classes is shown in Figure 2.3.

Figure 2.3. An overview of the java.lang exception classes

A zoom in of the java.lang.RuntimeException class hierarchy is shown in Figure 2.4.

Figure 2.4. An overview of the `java.lang.RuntimeException` classes

(A-heading) Threads

Threads enable parallelism with a program. Threading provides a low-overhead context switch. In the DiffCAD program, we use threads to display a digital clock in the lower left-hand corner of the screen. This is shown in figure 2.5.

Figure 2.5. The Digital Clock

The Java specification says that threads use a shared memory paradigm to perform inter-thread communication. This is the typical usage, however threads can use any number of techniques to perform communication. For example, simulation of a distributed computation scheme could be performed using threads and another communication technique, perhaps involving the socket API. The Java language specification also says that “Threads may be supported by having many hardware processors”. In fact this is not yet implemented on any virtual machine that we know of, though we expect this situation to change. Java makes use of the *synchronized* statement to make sure that a method contained within a thread is executed atomically. Atomic operations appear to happen all at once. Thus, synchronized methods cannot be interrupted by other threads. To put it another way, synchronized methods prevent

other threads from running. If the synchronized method takes longer than the time provided for sleeping, the method may not be updated frequently enough. The thread class hierarchy is shown in Figure 2.6.

Figure 2.6 The Thread class hierarchy

Synchronized methods allow only one thread to access data at a time. This can prevent data corruption. Synchronized methods can also prevent race conditions. The idea is that different threads may have access to the same variable in an instance. If both threads access the instance variable at the same time, then it is a race to see which thread will effect the change first, hence the term, race condition. The solution is to create synchronized accessor methods for every variable in an instance. Then, any change made through the synchronized accessor methods will be atomic. If there is a mix between synchronized accessor methods and asynchronous accessor methods, then the alterations to the instance variables' value are not atomic and the race condition may still exist.

Threads are run with a round-robin priority-queue driven scheduler. This can be overridden with custom schedulers (whose composition are beyond the scope of this text). Further if the Java virtual machine is written that does take advantage of multiple CPU's, task completion on multiple threads will not be predictable.

To make a new thread, subclass the Thread class and implement the run() method. Instance the thread sub-class and then invoke the start() method on the instance to begin thread execution.

Another way to make a thread is to use the Runnable interface. Any class that implements the Runnable interface must implement the run method. The run method is invoked when the thread is started. The thread is started when an instance of the runnable class is passed to the Thread constructor and start is invoked.

The following is a simple thread example called RaceThread. (NOTE) The RaceThread

class extends the Thread class. To run an instance of the RaceThread, you must make an instance of the RaceThread and use this instance as the target of a run() invocation. (END NOTE)

```

1.  class RaceThread extends Thread {
2.      public void run() {
3.          while (true) {
4.              System.out.println("Priority=\t" +
getPriority());
5.              System.out.println("toString=\t"+toString());
6.              System.out.println("getName=\t"+
getName());
7.              System.out.println("isDaemon=\t"+isDaemon());
8.              System.out.println("isAlive=\t"+isAlive());
9.                  try {Thread.sleep(10000);}
10.                 catch (InterruptedException e) {}
11.                 }
12.             }
13. }

```

Now, take a good look at the run method. The run method is at the heart of any threads activity. Classes that extend the thread class must implement the run method or they will not run (they compile ok, but upon launch, nothing happens!). Next, have a look at the while statement on line 3. It is an infinite loop! It will run until terminated. This makes line 9 and 10 real important. Line 9 will put the thread to sleep for 10000 milliseconds (10 seconds). The argument to Thread.sleep has to be a long integer. This makes sense when you consider that there are $2^{\log_2(1000*60*60*24)} = 2^{26}$ milliseconds in a day. Put it another way, a signed 32 bit integer will overflow in $2^{31} - 1$ milliseconds, which is just

over 3.5 weeks. The 64 bit integer will overflow in 292.4 billion years. At this point we will probably not be using 64 bit integers (or even FORTRAN). To put this in perspective, the Sun's corona will have engulfed the Earth by then. Thus this book will almost certainly be out of print. To run the RaceThread example, use the following code:

```
RaceThread r = new RaceThread();
r.start();
```

The following will be printed every 10 seconds at the console:

```
Priority=      5
toString=     Thread[Thread-2,5,main]
getName=      Thread-2
isDaemon=     false
isAlive=      true
```

Java makes threading a part of the java.lang package. This means that you can expect threading to be implemented, even on an embedded picoJava controller. Having threading described in the Java specification means that multi-threaded Java programs are as portable as Java.

The following code creates the digital clock in DiffCAD:

```
1. import java.util.*;
2. import java.awt.*;
3. import java.applet.Applet;
```

Lines 1, 2 and 3 import the packages that DiffCAD needs in the DigitalThread class.

```
4. // DigitalThread must extend applet to getFontMetrics.
5. public class DigitalThreads extends Applet implements
Runnable {
```

A class that implements Runnable must provide an implementation for the *run* method. Note that the Thread class has a specific type:

```
6. Thread runner;
```

The Java machine will execute threads until they die. A thread is dead when its run() method returns, or when the stop() method is called.

```

7.  Graphics g;
8.  Frame f;

9.  public void start() {
10.     if (runner == null) {

11.         runner = new Thread(this);
12.         runner.start();
13.     }
14. }
```

The stop method allows you to invoke the stop method of a thread that has already been stopped. Stopping a thread may be needed in order to avoid a race condition or to make sure that system resources are made available for other threads. After the stop() method is invoked, it may be that the thread should be restarted. Hence the reason for lines 9-14. They enable you to start a thread that has been stopped. You may invoke start on a method that is already started.

```

15. public void stop() {
16.     if (runner != null) {
17.         runner.stop();
18.         runner = null;
19.     }
20. }
```

In line 21, note the use of the synchronized keyword. This makes the draw method atomic. Draw is one method we do not want to interrupt, particularly with another draw method. Should this occur, draw could leave the screen in an inconsistent state. The frame field, *f* is set by the calling program. On line 25 we see that the date is being accessed from the java.util API.

```

21. synchronized private void draw() {
22.     Dimension dim = f.size();
23.     int height = dim.height - 60;
24.     int width = dim.width;
25.     Date theDate = new Date();
26.     String date_string = theDate.toString();
```

A particularly notable problem with the theDate.toString() method is that it is going to yield different returns on different platforms! For example, on Microsoft's J++ compiler, running under Windows 95/NT we found that the Eastern Daylight Time or Standard Time (EDT, ST) initials are present. For this graphics routine to work, we must therefore

query the graphics context for the font metrics, in order to size a background fill rectangle properly.

```
27.         int xloc = 10;
28.         int yloc = dim.height - 60;
```

The upper left-hand corner of the screen is the origin in Java. Thus, `dim.width`, `dim.height` are the coordinates of the lower-right hand corner.

```
29.         // g.setFont(f);
30.         // get the string width in pixels.
31.         int string_width = getFontMetrics(
32.             g.getFont()).stringWidth(date_string);
33.         int string_height = getFontMetrics(
34.             g.getFont()).getHeight();
```

Line 32 makes use of the graphics context, `g`. This is set using the calling program. Thus both the graphics context and the frame for the clock must be given before this program will have a draw target. Line 35 draws a filled clear rectangle under the string so that the string can be seen clearly, even if there is clutter on the screen.

```
35.
36.         g.clearRect(xloc,yloc,string_width,string_height);
37.         g.drawString(date_string,
xloc,height+xloc);
37.     }
```

On line 38 we see the `run` method. This is required of classes that implement the `run` method. Note that the only synchronized method, `draw`, is invoked on line 40. This gives us a return within a second, which is the sleep time for this thread, as see on line 41.

```
38. public void run() {
39.     while (true) {
40.         draw();
41.         try {Thread.sleep(1000);}
42.         catch (InterruptedException e) {}
43.     }
44. }
45. }
```

Start the `DigitalThreads` class by making an instance, setting the frame and graphics variables, then invoking the run method. This is shown in the code below:

```
1.     public static DigitalThreads
2.         clock_thread = new DigitalThreads();
3.     void start_clock() {
4.         clock_thread.g = getGraphics();
5.     clock_thread.f = main_frame;
6.     clock_thread.start();
7.     }
```

Note how in line 4 the graphics context was obtained from the local graphics context. This is assuming that the graphics context for the instance of the class exists. Further note how a `main_frame` variable was available for use in line 5. In fact the clock could be placed on any frame and graphics context. Notice that the clock will update the main frame, even if it becomes hidden by other frames. The clock thread does not check when its frame is hidden.

(B-heading) Thread Groups

One of the issues facing the programmer who wants to create many threads is that of limited system resources. CPU time is a limited resource and the programmer would like to keep an eye on the CPU usage and space requirements of the started threads. This is critical information, that is needed by both the program designer and by the program itself. The designer requires this information in order to engineer a program that reacts well to the changing load on a system.

Every thread belongs to a *thread group*. The Thread groups are arranged in a thread group hierarchy. At the root of this hierarchy is the *System Thread Group*. The System Thread Group consists of 4 threads : the GarbageCollector, clock handler, idle thread and finalizer thread. The main group is a child of the system thread group. The main group has the default thread that starts at the `main()` method. When windows are started, new threads are added to the main group. These threads include, but are not limited to, the AWT-Input thread, AWT-Toolkit thread and the ScreenUpdater thread.

(B-heading) The Thread Manager

In this section shows how to write a program that will enable the display of the various running threads on a system. At the heart of the thread management concept is the idea that we need to access the System Thread Group (i.e., root thread) and then list all of the children. The procedure is to create a thread and then traverse up the parent groups until reaching the root group. The root group is identified by its null parentage.

```
Thread.currentThread()
```

Will return a reference to the current thread. and

```
Thread.currentThread().getThreadGroup()
```

returns a reference to the group that the current thread belongs to. The reference is of ThreadGroup type. Once we have an instance of the current thread group, we can obtain the parent of the current thread group, which is also a thread group. We continue this traversal until the parent of the current thread group is null.

The following routine returns the SystemThreadGroup:

```
public ThreadGroup getSystemThreadGroup() {

    ThreadGroup systemThreadGroup;
    ThreadGroup parentThreadGroup;

    systemThreadGroup =
Thread.currentThread().getThreadGroup();

    while ( ( parentThreadGroup =
systemThreadGroup.getParent() ) != null)
        systemThreadGroup = parentThreadGroup;

    return systemThreadGroup;
}
```

The following code returns the groups of the system in an array of groups. We note that the number of groups is increase by 1 so that the root group may be added onto the end of the array:

```
1. public ThreadGroup[] getThreadGroupsArray() {
```

```

2.      ThreadGroup systemThreadGroup =
getSystemThreadGroup();

3.      int numberOfGroups =
systemThreadGroup.activeGroupCount() +1;

4.      ThreadGroup threadGroupsArray[] = new
5.          ThreadGroup[numberOfGroups];

6.      systemThreadGroup.enumerate(threadGroupsArray);

7.      threadGroupsArray[numberOfGroups] =
systemThreadGroup;

8.      return threadGroupsArray;
9.  }

```

When *group_instance.enumerate(ThreadGroup list[])* is called, all the thread group instances and their descendants are placed into the list. It does not include the *group_instance*, itself, however. This is why it is added to the end of the array on line 7. Another form of *enumerate* is associated with a thread instance, *group_instance.enumerate(Thread list{})* returns an array of threads in this thread group and all descendant thread groups.

```

java.lang.ThreadGroup[name=system,maxpri=10]
  Thread[Finalizer thread,1,system]
  Thread[Idle thread,1,system]
java.lang.ThreadGroup[name=main,maxpri=10]
  Thread[main,5,main]
  Thread[Thread-2,5,main]
  Thread[AWT-macos,5,main]
  Thread[Thread-3,5,main]
  Thread[Image Fetcher 0,8,main]
  Thread[Image Fetcher 1,8,main]
  Thread[Image Fetcher 2,8,main]
  Thread[Image Fetcher 3,8,main]
  Thread[Screen Updater,4,main]
  Thread[Audio Player,10,main]

```

The entire thread, called *RaceThread* (the new and improved version) appears below:

```

class RaceThread extends Thread {

    public void run() {
        while (true) {

```



```
        printThreadGroups();
        try {Thread.sleep(10000);}
        catch (InterruptedException e) {}
    }
}

public ThreadGroup getSystemThreadGroup() {

    ThreadGroup systemThreadGroup;
    ThreadGroup parentThreadGroup;

    systemThreadGroup =
Thread.currentThread().getThreadGroup();

    while ( ( parentThreadGroup =
systemThreadGroup.getParent()) != null)
        systemThreadGroup = parentThreadGroup;

    return systemThreadGroup;
}

public ThreadGroup[] getThreadGroupsArray() {

    ThreadGroup systemThreadGroup =
getSystemThreadGroup();

    int numberOfGroups =
systemThreadGroup.activeGroupCount() +1;

    ThreadGroup threadGroupsArray[] = new
        ThreadGroup[numberOfGroups];

    systemThreadGroup.enumerate(threadGroupsArray);

    threadGroupsArray[numberOfGroups] =
systemThreadGroup;

    return threadGroupsArray;
}

public Thread [] getThreadsArray() {

    ThreadGroup systemThreadGroup =
getSystemThreadGroup();
```

```

        Thread threadsArray[] = new
Thread[systemThreadGroup.activeCount()];
        systemThreadGroup.enumerate(threadsArray);

        return threadsArray ;
    }

    public void printThreadGroups() {
        getSystemThreadGroup().list();
    }

    public void printThreads() {
        Thread [] threadsArray = getThreadsArray();
        for (int i = 0; i < threadsArray.length; i++)
            System.out.println(threadsArray[i]);
    }

    public void printThreadsAndGroups() {
        System.out.println("The threads are:");
        printThreads();
        System.out.println("The groups are:");
        printThreadGroups();
    }
}

```

(A-heading) Summary

This chapter introduced some of the Java programming basics. A great deal of Java was not covered. The Java API was hardly touched and generally fills a large reference book. It is the case that several excellent reference books on Java already exist, and we wholeheartedly suggest that the reader obtain a few of them.

Roasting Your Own Java

Yield: 1/4 LB of the finest French Roast I ever had.

Place 113 GMs (about 1/4 LB) of Kenya AA green coffee into a Presto PopCornNow™ Plus Air Popper.

For a French Roast, roast for 5 minutes.

(WARNING)

Never leave roaster unattended.

Always roast outdoors as roasting makes a heavy blue smoke.

Replace the plastic hood with a

7.6 cm (3 inch) diameter by 20.3 cm (8 inch)

long metal stove pipe to prevent melting and discoloration of the hood.

Stove pipes are available in Home Centers.

Roasting with a hot-air popper violates the warranty and makes popcorn taste funny.

(END WARNING)

(CN) 3. (CT) The Graphic User Interface

This chapter provides an overview of the GUI (Graphic User Interface) in Java. Central to the look and feel of the Java GUI is the AWT (Abstract Window Toolkit). The AWT consists of a collection of classes, some of which are extended 6 levels deep or more. For example, a programmer who extends an Applet class inherits the methods and instance variables from Object, Component, Container and Panel super classes. This means that for complete coverage of the Applet class, we would have to describe the four base classes, in addition to the applet class. Thorough coverage can lead to a dry presentation, and is better left to references like [Chan and Lee]. Our approach is to cover the material that is essential to understanding how to write digital signal processing programs. The AWT is rich with many GUI options and widgets. We shall cover only those that are essential to the task of digital signal processing. Many of the finer points of the AWT are better left to book dedicated to the task, like [Geary and McClellan].

The first section is the Color class. We then lead directly into the graphics class. For the purpose of digital signal processing we need to learn how to draw a line, draw a disk, draw a string and draw an image. We also want to learn how to set colors and set fonts. The Graphics class is deep and we do not attempt to give complete coverage of it.

(A-heading) The Color Class

The Color class is a public final class that is based on an RGB color model. The Color class has several public final static colors that are predefined. An instance of the Color class may be specified using two variants of the additive color synthesis system, RGB (red, green and blue) or HSB (hue, saturation or brightness). The additive color synthesis system is based on the premise that black is the absence of color and that white is the combinations of all colors. For example, when red green and blue are combined in equal amounts, the resulting color is white. This is consistent with electronic displays that emit light. The subtractive synthesis color system, is based on the premise that black is the combination of all colors and that white is the absence of all colors. This is a result of light absorbing pigments that are applied to a white background. The subtractive synthesis color primaries are CMY (cyan, magenta and yellow). The AWT does not support subtractive synthesis.

(B-heading) Class Summary

```

public final class Color {
public final static Color white
public final static Color lightGray
public final static Color gray
public final static Color darkGray
public final static Color black
public final static Color red
public final static Color pink
public final static Color orange
public final static Color yellow
public final static Color green
public final static Color magenta
public final static Color cyan
public final static Color blue
public Color(int r, int g, int b)
public Color(int rgb)
public Color(float r, float g, float b)
public int getRed()
public int getGreen()
public int getBlue()
public int getRGB()
public Color brighter()
public Color darker()
public int hashCode()
public boolean equals(Object obj)
public String toString()
public static Color getColor(String nm)
public static Color getColor(String nm, Color v)
public static Color getColor(String nm, int v)
public static int HSBtoRGB(float hue, float saturation,
float brightness)
public static float[] RGBtoHSB(int r, int g, int b, float[]
hsbvals)
public static Color getHSBColor(float h, float s, float b)
}

```

(B-heading) Class Usage

Suppose that the following variables are defined:

```

Color c, c1;
int r, g, b;
float rf, gf, bf;
int anInt;

```

Then to access the built-in color names use:

```
c = Color.white;
c = Color.gray;
c = Color.lightGray;
c = Color.darkGray;
c = Color.black;
c = Color.red;
c = Color.pink;
c = Color.orange;
c = Color.yellow;
c = Color.green;
c = Color.magenta;
c = Color.cyan;
c = Color.blue;
```

To construct a new color use:

```
c = new Color(r, g, b);
```

The AWT assumes that the r, g, and b quantities range from 0..255. Truncation results if the range assumption is violated. In fact, the color is typically a packed int consisting of a blue component in bits 0-7, a green component in bits 8-15, and a red component in bits 16-23. To instantiate a new color from an existing instance:

```
c1 = new Color(c);
```

To instantiate a new color from red, green and blue components in the range from 0..1, inclusive use:

```
c = new Color(rf, gf, bf);
```

To get the components of a color use:

```
r = c.getRed();
g = c.getGreen();
b = c.getBlue();
```

To get a 24 bit RGB color packed into an int and typed as an int:

```
anInt = c.getRGB();
```

To brighten or darken a color:

```
c.brighter();
c.darker();
```

To obtain the color hashCode (implemented as getRGB()):

```
anInt = c.hashCode();
```

To compare color values:

```
aBoolean = c.equals(c2);
```

To convert to string:

```
str = c.toString();
```

If nm is a valid color property name, the Integer class can use the getInteger method to map the property name into a color. This is done with the following static methods:

```
c1 = Color.getColor(nm);
```

To return a color if the property name is undefined:

```
c1 = Color.getColor(nm, aColor);
```

To return a color based on a 24 bit RGB color int, if name is undefined:

```
c1 = Color.getColor(nm, anInt);
```

(BEGIN NOTE)

Color properties must be set by the Java application, they are not predefined.

(END NOTE)

To convert a floating point description of hue, saturation and brightness to the RGB model (with a 24 bit int) use:

```
// ranging from 0..1 inclusive  
float hue, saturation, brightness;  
anInt = Color.HSBtoRGB(hue, saturation, brightness);
```

To convert from RGB to HSB, there are two methods to choose from:

```
float hsbvals[];  
int r, g, b; // r, g and b range from 0..255  
hsbvals = Color.RGBtoHSB(r, g, b, null);  
Color.RGBtoHSB(r, g, b, hsbvals);
```

Both return 3 values in the `hsbvals` array.

To make an instance of a color from an HSB floating point triplet use:

```
Color aColor = Color.HSBtoRGB(h, s, b);
```

(A-heading) The Graphics Class

The graphics class is used to draw lines, shapes, images and characters. The Graphics class resides in the `java.awt` package and is an abstract base class. Since the Graphics class is abstract, it may never be created by the invocation of `new`. Instead, it is created by an instance of a Component. The graphics class has a default coordinate system, as shown in Figure 3.1.

Figure 3.1 The Default Coordinate System of the Graphics Class

(B-heading) Class Summary

```
public abstract class Graphics {  
    public abstract Graphics create()  
    public Graphics create(int x, int y, int width, int height)  
    public abstract void translate(int x, int y)  
    public abstract Color getColor()  
    public abstract void setColor(Color c)  
    public abstract void setPaintMode()  
    public abstract void setXORMode(Color c1)  
    public abstract Font getFont()  
    public abstract void setFont(Font font)
```

```
public FontMetrics getFontMetrics()
public abstract FontMetrics getFontMetrics(Font f)
public abstract Rectangle getClipRect()
public abstract void clipRect(int x, int y, int width, int
height)
public abstract void copyArea(int x, int y, int width, int
height, int dx, int dy)
public abstract void drawLine(int x1, int y1, int x2, int
y2)
public abstract void fillRect(int x, int y, int width, int
height)
public void drawRect(int x, int y, int width, int height)
public abstract void clearRect(int x, int y, int width, int
height)
public abstract void drawRoundRect(int x, int y, int width,
int height, int arcWidth, int arcHeight)
public abstract void fillRoundRect(int x, int y, int width,
int height, int arcWidth, int arcHeight)
public void draw3DRect(int x, int y, int width, int height,
boolean raised)
public void fill3DRect(int x, int y, int width, int height,
boolean raised)
public abstract void drawOval(int x, int y, int width, int
height)
public abstract void fillOval(int x, int y, int width, int
height)
public abstract void drawArc(int x, int y, int width, int
height, int startAngle, int arcAngle)
public abstract void fillArc(int x, int y, int width, int
height, int startAngle, int arcAngle)
public abstract void drawPolygon(int xPoints[], int
yPoints[], int nPoints)
public void drawPolygon(Polygon p)
public abstract void fillPolygon(int xPoints[], int
yPoints[], int nPoints)
public void fillPolygon(Polygon p)
public abstract void drawString(String str, int x, int y)
public void drawChars(char data[], int offset, int length,
int x, int y)
public void drawBytes(byte data[], int offset, int length,
int x, int y)
public abstract boolean drawImage(Image img, int x, int y,
ImageObserver observer)
public abstract boolean drawImage(Image img, int x, int y,
int width, int height, ImageObserver observer)
public abstract boolean drawImage(Image img, int x, int y,
Color bgcolor, ImageObserver observer)
```



```

public abstract boolean drawImage(Image img, int x, int y,
int width, int height, Color bgcolor, ImageObserver
observer)
public abstract void dispose()
public void finalize()
public String toString()
}

```

(B-heading) Class Usage

The Graphics class supports a simple raster graphics package. Suppose the following constants are predefined:

```

Color aColor;
Graphics g;
Rectangle aRectangle;
int x, y, x1, y1, x2, y2;
Boolean raised;
int height, width, arcHeight, arcWidth, nPoint;
int xArray[], yArray[];
char charArray[];
int numberOfItemToDraw, offsetIntoArray;
byte byteArray[];
Image img;
ImageObserver anImageObserver;

```

To create a graphics instance use:

```
Graphics g1 = g.create(x, y, width, height);
```

There are color methods for getting and setting the foreground color of the graphics instance:

```

aColor = g.getColor();
g.setColor(aColor);

```

There are paint methods for setting the graphics instance to overwrite in xor paint mode:

```

g.setPaintMode();
g.setXORMode(aColor);

```

There are clipping methods for shrinking and for getting the clipping rectangle:

```

aRectangle = g.getClipRect();
g.clipRect(x, y, width, height);

```

The Graphics class supports a method for painting a rectangle with the background color:

```
clearRect(int x, int y, int width, int height);
```

To translate the coordinates of the Graphics instance use:

```
g.translate(x, y);
```

To get and set the fonts of the Graphics instance:

```

aFont = g.getFont();
g.setFont(aFont);

```

To get the font metrics for a graphics instance or for a font:

```

theFontMetrics = g.getFontMetrics();
theFontMetrics = g.getFontMetrics(aFont);

```

To get the clipping rectangle:

```
aRectangle = g.getClipRect();
```

To shrink the clipping rectangle:

```
g.clipRect(x, y, width, height);
```

To copy and translate a rectangular area of the screen by dx, dy:

```
g.copyArea(x, y, width, height, dx, dy);
```

To draw a line between x1, y1 and x2, y2:

```
g.drawLine(x1, y1, x2, y2);
```

To fill a rectangle with the current color:

```
g.fillRect(x, y, width, height);
```

To draw a rectangle outline with with the current color:

```
g.drawRect(x, y, width, height);
```

The drawRect method is currently implemented with a series of four draw-lines.

Hopefully this will change as the AWT becomes more optimized.

To clear a rectangular area of the screen, the Graphics class draws a rectangle with the current background color:

```
g.clearRect(x, y, width, height);
```

The Graphics class provides rounded shape drawing. When a rounded shape is drawn, the arcWidth and arcHeight are used. The arcWidth and arcHeight are the width and height of an ellipse used to draw the rounded corners. To draw the outline of a rectangle with

rounded corners, using the current color:

```
g.drawRoundRect(x, y, width, height, arcWidth, arcHeight);
```

To draw a filled rectangle with rounded corners, using the current color:

```
g.fillRoundRect(x, y, width, height, arcWidth, arcHeight);
```

To draw a highlighted 3-D rectangle using two colors:

```
g.draw3DRect(x, y, width, height, raised);
```

If raised is true, the rectangle appears raised. This is currently implemented using darker and brighter colors to create hi-lights. The same rectangle may also be drawn filled:

```
g.fill3DRect(x, y, width, height, raised);
```

To draw an oval outline or a filled oval use:

```
g.drawOval(x, y, width, height);
```

```
g.fillOval(x, y, width, height);
```

To draw an arc inscribed in a rectangle, starting at a startAngle and ending at startAngle+arcAngle, use:

```
g.drawArc(x, y, width, height, startAngle, arcAngle);
```

Angles are in degrees. Zero degrees is parallel to the X-axis and positive angles are measured in a counter-clockwise direction. For the filled form of the arc:

```
g.fillArc(x, y, width, height, startAngle, arcAngle);
```

To draw a polygon using an array of (x,y) coordinates:

```
g.drawPolygon(xArray, yArray, nPoints);
```

To draw a polygon using an instance of the Polygon class:

```
g.drawPolygon(aPolygon);
```

The drawPolygon method is implemented with a call to

```
drawPolygon(aPolygon.xpoints, aPolygon.ypoints,  
aPolygon.npoints);
```

To fill the polygon with the current color

```
g.fillPolygon(xArray, yArray, nPoints);
```

There must be more than 3 points or the polygon call is ignored. Further, if the polygon has overlapping parts the even-odd fill rule is used. The even-odd fill rule is typically implemented using a scanline that is drawn from left to right. If the scanline crosses the edge of the polygon an odd number of times, the pixels are inside the polygon and will be filled, otherwise they fall outside of the polygon. This is a typical approach to filling concave polygons.

```
g.fillPolygon(aPolygon);
```

To draw a string starting at (x,y), using the current font and color:

```
String str = "Java is fun";
```

```
g.drawString(str,x,y);
```

To draw a character or byte array starting at (x,y), using the current font and color:

```
g.drawChars(charArray, offsetIntoArray, numberOfItemToDraw,
x, y);
```

```
g.drawBytes(byteArray, offsetIntoArray,
numberOfCharactersToDraw, x, y);
```

The drawBytes method assumes that the bytes represent the least significant 8 bits of the Unicode characters. To draw an image at x,y and notify anImageObserver:

```
aBoolean = g.drawImage(img, x, y, anImageObserver);
```

To draw an image inside a rectangle (with optional scaling):

```
aBoolean = g.drawImage(img, x, y, width, height,
anImageObserver);
```

To draw an image with a background color:

```
aBoolean = g.drawImage(img, x, y, aColor,
anImageObserver);
```

To draw an image with a background color and in a rectangle:

```
aBoolean = g.drawImage(img, x, y, width, height, aColor,
anImageObserver);
```

To dispose of the graphics context (before the garbage collector does):

```
g.dispose();
```

The Graphics class overrides Object.finalize() with a dispose call:

```
g.finalize();
```

To convert the Graphics instance into a string:

```
g.toString();
```

The following sections give practical examples for the use of the methods in the Graphic class.

(B-heading) How to draw a grid

In this section you will learn how to write a class that can draw a grid into a Graphics instance. Consider the following code for drawing a grid:

```
1. import java.applet.*;
2. import java.util.*;
3. import java.awt.*;
```

Lines 1-3 are used to import packages into your program. Line 4 declares the class as a final class. This prevents other programmers from subclassing the draw class.

```
4. final public class draw {
5. // prevent instantiation
6. private draw() {}
```

Line 6 declares the constructor for the draw class to be private. This prevents the draw class from being instantiated by programmers who develop outside of the draw class. Line 7 is the grid method. The grid method takes an instance of a Graphics class. The dimensions of the clip rectangle are extracted from a Rectangle instance using the width and height members of the rectangle class. The width and height are stored as integers and in units of pixels.

```
7. static void grid(Graphics g) {
8. Rectangle r = g.getClipRect();
```

The Geometry class is a custom class used by the DiffCAD program to store global instances. Grid_spacing is an int that has units of pixels.

```
9. int grid_spacing = Geometry.grid_width.getValue();
10. int w = r.width;
```

```
11.     int h = r.height;
12.     for (int x = 0; x < w; x = x + grid_spacing)
13.         {g.drawLine(x,0,x,h);}
14.         for (int y = 0; y < h; y = y + grid_spacing)
15.             {g.drawLine(0,y,w,y);}
16.     }
17. }
```

(A-heading) The FontMetrics Class

The FontMetrics class is a public abstract class in the java.awt package. Font metrics are based on dimensions that are special to fonts. Font dimensions include the baseline, ascent, descent, leading and height. Distances are generally given in pixels (when they are returned by the FontMetrics attribute methods). The font metrics Dimensions are shown in Figure 3.2

Figure 3.2 Font Metrics Dimensions

The leading is the space between lines. The name comes from the lead strips that were inserted between slugs by a compositor. A slug is a line of cast type that is assembled by a compositor. A compositor may be manual or automatic. An automatic compositor is called a typesetting machine. The first practical typesetting machine, the Linotype machine, was first used in 1886 by the New York Tribune [Mertle]. Nomenclature has passed down from this era, essentially unchanged. Typical font metrics are in units of points (at 72 points per inch). The raster orientation of the java.awt describes all the font metrics in units of pixels. The number of pixels per inch is called the display's pitch. Thus, there is no display independent way to automatically relate the font metrics of the AWT to the font metrics of traditional typesetting. An automatic means would have to have the ability to detect the pitch of the display before the traditional fontmetrics could be computed.

(B-heading) Class Summary

```

package java.awt;
public abstract class FontMetrics {
    public Font getFont()
    public int getLeading()
    public int getAscent()
    public int getDescent()
    public int getHeight()
    public int getMaxAscent()
    public int getMaxDescent()
    public int getMaxDecent()
    public int getMaxAdvance()
    public int charWidth(int ch)
    public int charWidth(char ch)
    public int stringWidth(String str)
    public int charsWidth(char data[], int off, int len)
    public int bytesWidth(byte data[], int off, int len)
    public int[] getWidths()
    public String toString()
}

```

(B-heading) Class Usage

Suppose that the following constants are defined:

```

Graphics g;
FontMetrics theFontMetrics = g.getFontMetrics();
Font aFont;
int distanceInPixels;
char ch;
char charArray[];
byte byteArray[];
int offset, length

```

To get the instance of the Font class upon which theFontMetrics are based:

```
afont = theFontMetrics.getFont();
```

To get the standard leading (line spacing between descent and ascent):

```
distanceInPixels = theFontMetrics.getLeading();
```

To get the ascent, descent and height:

```
distanceInPixels = theFontMetrics.getAscent();  
distanceInPixels = theFontMetrics.getDescent();  
distanceInPixels = theFontMetrics.getHeight();
```

To get the maximum ascent, and descent for a font:

```
distanceInPixels = theFontMetrics.getMaxAscent();  
distanceInPixels = theFontMetrics.getMaxDescent();
```

To get the maximum height for a font, add the maximum descent and ascent. To get the width of a character in the font:

```
distanceInPixels = theFontMetrics.charWidth(ch);
```

To get the width of a string in the font:

```
distanceInPixels = theFontMetrics.stringWidth(str);
```

To get the width of an array of characters in the font, starting at the offset and proceeding for length characters:

```
distanceInPixels = theFontMetrics.charsWidth(charArray,  
offset, length);
```

To get the width of an array of bytes in the font, starting at the offset and proceeding for length characters (characters come in as bytes and are converted to 16 bit characters):

```
distanceInPixels = theFontMetrics.bytesWidth(byteArray,  
offset, length);
```

To get the width of the first 256 characters in the font:

```
distanceInPixels = theFontMetrics.getWidths();
```

To get the string representation of the FontMetrics:

```
str = theFontMetrics.toString();
```

(B-heading) How to Draw a String with a Background

Often the user will wish to draw a string that has a background imposed. This is important if the string is to be changed dynamically. In the

following example we show how to draw the date and time. The `clearRect` call will erase a part of the display so that the string does not over-write itself. An example of the string, with filled background in place, is shown in Figure 3.3.

Figure 3.3 String With Filled Background

```
synchronized private void draw() {
Dimension dim = f.size();
int height = dim.height - 60;
int width = dim.width;
Date theDate = new Date();
String date_string = theDate.toString();
int xloc = 10;
int yloc = dim.height - 60;
int string_width = getFontMetrics(
    g.getFont()).stringWidth(date_string);
int string_height = getFontMetrics(
    g.getFont()).getHeight();
g.clearRect(xloc,yloc,string_width,string_height);
g.drawString(date_string, xloc,height+xloc);
}
```

(B-heading) How to Draw a Vertical String

The following method takes a string and draws it vertically (something the AWT does not normally do):

```
public void drawVerticalString(Graphics g,String str,int
x,int y) {
int str_height = g.getFontMetrics().getHeight();
(str_height*str.length())/2;
for (int i = 0; i<str.length(); i++) {
int char_width =
g.getFontMetrics().stringWidth(str.substring(i,i+1));
g.drawString(str.substring(i,i+1),x-char_width/2,y);
y+=str_height;
} // end for
} // end drawVerticalString
```

An example of the use of the vertical string is shown in Figure 3.4

Figure 3.4 An example of the Vertical String, The Histogram

(A-heading) The MenuItem Class

An instance of the *MenuItem* class is typically added to a *Menu* instance. When an event is posted by a component, an instance of the *MenuItem* class may be passed in the *target* member of the *Event* instance. Thus, it is typical to write *handleEvent* methods that will compare targets with *MenuItem* instances, to see what the user has selected. The event handler routines are discussed in more detail later in this chapter.

(B-Heading) Class Summary

```

public class MenuItem extends java.awt.MenuComponent {
    /* Instance Variables */
        boolean enabled;
        String label;
    /* Methods */
        public void MenuItem(String label);
        public synchronized void addNotify();
        public String getLabel();
        public void setLabel(String label);
        public boolean isEnabled();
        public void enable();
        public void enable(boolean on);
        public void disable();
        public String paramString();
}

```

(B-heading) Class Usage

The MenuItem class is a subclass of the MenuComponent and is used to store a string item that represents a menu choice. Assume that the following variables are predefined:

```

MenuItem ifft_mi          = new MenuItem("[2] IFFT");
String label;
boolean on;

```

To get and set the string contained in the MenuItem instance, use:

```

str = ifft_mi.getLabel();
ifft_mi.setLabel(label);

```

To get and set the enabling on the MenuItem instance:

```

on = ifft_mi.isEnabled();
ifft_mi.enable(); // always enables
ifft_mi.enable(on); // conditionally enables
ifft_mi.disable(); // always disables

```

To obtain "label=" + ifft_mi.getLabel() use:

```

label = ifft_mi.paramString();

```

(BEGIN NOTE)

The MenuItem class has a special “-” label that is used as a separator. An example of the separator’s appearance in the menu is shown in Figure 3.5

(END NOTE)

Figure 3.5 The Separator is shown between SendMail and Is.

(A-heading) The Event Class

The Event class is a reference data type that is a direct descendent of the java.lang.Object class. The handleEvent method is invoked with an instance of the Event class whenever user input is detected. *Components* are said to *post* events.

(B-heading) Class Summary

```
public class Event {
    public static final int SHIFT_MASK
    public static final int CTRL_MASK
    public static final int META_MASK
    public static final int ALT_MASK
    public static final int HOME
    public static final int END
    public static final int PGUP
    public static final int PGDN
    public static final int UP
    public static final int DOWN
    public static final int LEFT
    public static final int RIGHT
    public static final int F1
    public static final int F2
    public static final int F3
    public static final int F4
    public static final int F5
    public static final int F6
    public static final int F7
    public static final int F8
    public static final int F9
    public static final int F10
    public static final int F11
    public static final int F12
    public static final int WINDOW_DESTROY
    public static final int WINDOW_EXPOSE
    public static final int WINDOW_ICONIFY
    public static final int WINDOW_DEICONIFY
    public static final int WINDOW_MOVED
    public static final int KEY_PRESS
    public static final int KEY_RELEASE
    public static final int KEY_ACTION
    public static final int KEY_ACTION_RELEASE
    public static final int MOUSE_DOWN
    public static final int MOUSE_UP
    public static final int MOUSE_MOVE
}
```

```
public static final int MOUSE_ENTER
public static final int MOUSE_EXIT
public static final int MOUSE_DRAG
public static final int SCROLL_LINE_UP
public static final int SCROLL_LINE_DOWN
public static final int SCROLL_PAGE_UP
public static final int SCROLL_PAGE_DOWN
public static final int SCROLL_ABSOLUTE
public static final int LIST_SELECT
public static final int LIST_DESELECT
public static final int ACTION_EVENT
public static final int LOAD_FILE
public static final int SAVE_FILE
public static final int GOT_FOCUS
public static final int LOST_FOCUS
public Object target;
public long when;
public int id;
public int x;
public int y;
public int key;
public int modifiers;
public int clickCount;
public Object arg;
```

```

    public Event evt;
/* Methods */
    public Event(Object target, long when, int id, int x,
int y, int key,int modifiers, Object arg)
    public Event(Object target, long when, int id, int x,
int y, int key, int modifiers)
    public Event(Object target, int id, Object arg)
    public void translate(int x, int y)
    public boolean shiftDown()
    public boolean controlDown()
    public boolean metaDown()
    protected String paramString()
    public String toString()
}

```

(B-heading) Class Usage

Suppose the following constants are pre-defined:

```

Object target;
long when; // a creation time stamp
int id; // the event type
int x, y; // event location
int key;
int modifiers;
Object arg;

```

The *target* is the instance of the Component class that posted the event. The *when* member holds a creation time stamp, measured in milliseconds. The *id* permits decoding of the event type with the use of a switch statement. The location of the event is contained in (x,y). The state of the modifier keys is encoded into the *modifiers* parameter. Arg is the instance of the class that is to be associated with the event. The Event class constructor is overloaded so that we can write:

```

Event evt = new Event(target, id, arg);
Event evt = new Event(target, when, id, x, y, key,
    modifiers);
Event evt = new Event(target, when, id, x, y, key,
    modifiers, arg);

```

Typically, it is the users' job to write code to decode the events. Unfortunately the services provided by the AWT lead to an ad-hoc approach for event handling that causes poor software engineering. As a preview, we will see that the AWT assumes that keyboard events and menu choice events should be handled in different parts of the source code. Often keyboard short-cuts are a means for speeding menu selections. Thus, having menu selections processed in a different part of the source code from that of the keyboard event leads to duplication of code. Further, it requires additional development effort to add keyboard short cuts. A remedy for this design flaw in the AWT is discussed in the following section.

There are several public variables associated with the Event class. Some of the public variables are key-masks designed to help with decoding. An example of this is shown in the Component section:

KEY_EVENT, SHIFT_MASK, CTRL_MASK, META_MASK, ALT_MASK, HOME,
END, PGUP, PGDN, KEY_PRESS, KEY_RELEASE, KEY_ACTION,
KEY_ACTION_RELEASE

The arrow keys are decoded with:

UP, DOWN, LEFT, RIGHT

The function keys are decoded with:

F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12

The window events are decoded with:

WINDOW_EVENT, WINDOW_DESTROY, WINDOW_EXPOSE,
WINDOW_ICONIFY, WINDOW_DEICONIFY, WINDOW_MOVED,
WINDOW_EVENT

The mouse events are decoded with:

MOUSE_EVENT, MOUSE_DOWN, MOUSE_UP, MOUSE_MOVE, MOUSE_ENTER,
MOUSE_EXIT, MOUSE_DRAG

The scrolling events are decoded with:

SCROLL_EVENT, SCROLL_LINE_UP, SCROLL_LINE_DOWN,
SCROLL_PAGE_UP, SCROLL_PAGE_DOWN, SCROLL_ABSOLUTE

The list events are decoded with:

LIST_EVENT, LIST_SELECT, LIST_DESELECT

There are a series of events that are classified as miscellaneous events:

MISC_EVENT, ACTION_EVENT, LOAD_FILE, SAVE_FILE, GOT_FOCUS,
LOST_FOCUS

The processing of Event instances is discussed in the following section.

(B-heading) Event Handling

The AWT model for handling events can lead to poor software engineering. This is due, in part, to the event modularization policy that has been adopted by the AWT designers. The designers sought to map mouse events, keyboard events and menu selection events using a preprocessing

facility built into the component class. This is based on the assumption that the relationship between event production and code invocation is isomorphic. This is a fallacious assumption because multiple events can often be used to trigger a single task. Often there will be a menu choice that has a keyboard short so that both a menu event and a keypress event can trigger the same processing.

(B-heading) The Keyboard

To handle keyboard events in the AWT, a class that extends the Component class will typically have its *keyDown* method invoked. In the following code fragment, the *AudioFrame* class extends the *Frame* class (which is two generations removed from the Component super class):

```

...
public class AudioFrame extends Frame {
...
public boolean keyDown(Event e, int key) {
    switch (key) {
        case '2':
            ifft();
            return true;
    }
    System.out.println("Unknown key function:"+ key);
    return true;
}
}

```

(B-heading) The Target

To handle the passing of an instance, the Event class supports a target member that is of Object type. To process this event a method must be written called *handleEvent*:

```

...
public class AudioFrame extends Frame {
...
Menu m = new Menu("Audio Menu");
...
MenuItem ifft_mi = addItem("[2] IFFT");
public MenuItem addItem(String itemName) {
    MenuItem mi = new MenuItem(itemName);
    m.add(mi);
    return(mi);
}
public void init_menu() {
    // my menu items
    MenuBar menuBar = new MenuBar();
}
}

```



```
// Initialize the menu bar
    setMenuBar(menuBar);
    menuBar.add(m);
...
public boolean handleEvent(Event e) {
...
    if (e.target == ifft_mi) {
```

```
    ifft();  
        return true;  
    }
```

One observation that the astute reader may have made is that the keyboard event called the same code as the `handleEvent`. This is because the `MenuItem` instance (which appears in the main menu bar) has a keyboard shortcut. Keyboard shortcuts are typically provided as a convenience to the user. The duplication of keyboard and menu event processing is typical of AWT event processing code. It is also unsound software engineering. The reason is that parallel maintenance must be performed in both the keyboard and menu event processing code. A more sound software engineering practice appears in the following section.

(B-heading) The Evt Class

The `Evt` class is a public static class that contains a series of methods that address the software engineering problem cited in the previous section. The basic assumption is that several different event *combinations* can cause a single result, the example being a keypress event *or* a menu item event. The solution involves the creation of an overloaded *match* method:

```
class Evt {  
    ...
```

```

    public static boolean matchKey(Event e, int key) {
        return ((e.id == Event.KEY_PRESS) && (e.key==key));
    }
    public static boolean match(Event e, int key, String
str) {
        return (matchKey(e,key) || e.arg.equals(str));
    }

    public static boolean match(Event e, int key, Object
target) {
        return (matchKey(e,key) || e.target.equals(target));
    }

```

...
The `Evt` class gives the immediate benefit of giving a two-way match between keyboard and menuItem events. In the `AudioFrame` class, the `keyDown` method is eliminated and replaced with the more powerful:

```

public boolean handleEvent(Event e) {
    ...
        if (Evt.match(e, '2', ifft_mi)) {
            ifft();
            return true;
        }
    }

```

...
This single convention eliminates 60 lines of code in the `AudioFrame` `keyDown` event handler. In order to add another level of automation to the event processing, we make the assumption that the keyboard short cut will be encoded into the string representation of the menuItem, as it appears in the main menu bar. Figure 3.6 shows a sample main menu bar from the `AudioFrame` in the `DiffCAD` program:

Figure 3.6 Sample `AudioFrame` Main Menu Bar Pop-Up Menu

One feature of the `AudioFrame` keyboard short-cuts is that they conform to a convention. The convention states: if a keyboard short-cut exists, it is encoded by a '[' followed by a single character. Tognazzini reports that \$50 million dollars of research led to the conclusion that 1. users say keyboard shortcuts are faster and 2. the stopwatch shows that mouse choices are faster. Tognazzini produced a guideline that states that visual interface construction should not have its resources sapped by the keyboard interface [Tog]. As a result of this suggestion (common sense?) we have combined the following check into a single method called *match*:

```

public static boolean match(Event e, Object target) {
    int c = getKeyboardShortCut(e, target);
    return match(e, c, target);
}

```

In the following code, *getKeyboardShortCut* assumes that, if a keyboard short cut is embedded in the target that: 1. the target will be an instance of a `MenuItem`, 2. that the first character of the label will be a '[' and 3. that the second character of the label will be the keyboard shortcut character.

```

    public static int getKeyboardShortCut(Event e, Object
target) {
        if (target instanceof MenuItem) {
            MenuItem mi = (MenuItem) target;
            String str = mi.getLabel();
            int index = str.indexOf('[');
            if (index == 0) { return str.charAt(1);}
        }
        return -1;
    }
}

```

This leads to a more consistent interface (keyboard shortcuts are always encoded visually and consistently), an easier to use API (only one call, instead of two) and better software engineering (centralization of interface changes with fewer lines of code). Keyboard and menu item events are now handled as follows:

```

...
public boolean handleEvent(Event e) {
...
    if (Evt.match(e,ifft_mi)) {
        ifft();
        return true;
    }
}
...

```

It takes no more effort to encode and process a keyboard short cut:

```
MenuItem ifft_mi = addItem("[2] IFFT");
```

than to process and set-up a visual interface, hence fulfilling Tognazzini's basic guideline.

(B-heading) The Mouse

Any pick device may generate mouse events (i.e., track-ball, pen-light, touch-pad, tablet, etc.). The Java AWT only supports a single pick device and, no matter what the pick device is, it is said to generate mouse events. To handle the mouse events, the AWT Component class provides a suite of call back methods.

When mouse button is depressed:

```
public boolean mouseDown(Event evt, int x, int y)
```

When mouse button is depressed and mouse is moved:

```
public boolean mouseDrag(Event evt, int x, int y)
```

When mouse button is released:

```
public boolean mouseUp(Event evt, int x, int y)
```

When mouse is moved:

```
public boolean mouseMove(Event evt, int x, int y)
```

When mouse is over a component:

```
public boolean mouseEnter(Event evt, int x, int y)
```

When mouse leaves the component:

```
public boolean mouseExit(Event evt, int x, int y)
```

For the programmer to implement a mouse event requires that either the above events are over-riden or that a switch statement be used to process the Event.id. The choice between switch statement and method is a question of preference, policy and good software engineering practice. The switch statement is more flexible and permits more "combination" events (like a shift-click for extending a selection). (BEGIN CD-ROM)

In the following code fragment taken from GUI.java, a `handleEvent` method is implemented with a switch statement. (END CDROM)

```
1. public boolean handleEvent(Event e) {  
2.     switch (e.id) {
```

On line 3, the `Event.MOUSE_UP` indicates that the mouse button was released. When this occurs, the `processMouseUp` method is invoked. This causes an object to change its appearance.

```
3.     case Event.MOUSE_UP: {
4.     String objectName = mouse_choice.getSelectedItem();
5.     processMouseUp( objectName, e.x, e.y);
6.     repaint();
7.     }
```

On line 8, the Event.MOUSE_DOWN triggers a recording of the location of the mouse cursor. The anchor point is used to provide relative mouse motion.

```
8.     case Event.MOUSE_DOWN:
9.         anchor = new point( e.x, e.y);

10.    } // switch
```

(A-heading) The Component Class

The component class is an abstract class. Subclasses of components are used to make instances that are able to be displayed on the screen. Examples of such component subclasses include the Button, Canvas, Checkbox, Choice, Container, Label, List, Scrollbar and TextComponent. Figure 3.7 shows the component hierarchy

Figure 3.7 Component Hierarchy

The Component class resides in the java.awt package. It implements the ImageObserver interface. A component instance supports services that includes drawing support, event handling, font control, color control, image handling, size and position control. These services are provided by a series of methods and instance variables.

A suite of methods are available that support event handling. They are *handleEvent*, *mouseEnter*, *mouseExit*, *mouseMove*, *mouseDown*, *mouseDrag*, *mouseUp*, *keyDown* and *action*.

The Component class provides a mechanism for invoking the specialized event handlers. This mechanism is based on a switch statement that calls methods that are to be implemented by subclasses. The methods are implemented in the Component.java class by returning false. A return of false signals that the event was not processed.

(BEGIN NOTE)

Some components are used for input, some for output and some function as both input and output. In a later section we will see an example of a Button Component which is used only as an input. We will also see an example of a Label Component, which serves only as an output. Later we will also see the Scrollbar Component. This can serve as both an input and an output Component.

(END NOTE)

(B-heading) Class Summary

```
public abstract class Component implements ImageObserver {
public Container getParent()
public ComponentPeer getPeer()
public Toolkit getToolkit()
public boolean isValid()
public boolean isVisible()
public boolean isShowing()
public boolean isEnabled()
public Point location()
public Dimension size()
public Rectangle bounds()
public synchronized void enable()
public void enable(boolean cond)
public synchronized void disable()
public synchronized void show()
public void show(boolean cond)
public synchronized void hide()
public Color getForeground()
public synchronized void setForeground(Color c)
public Color getBackground()
public synchronized void setBackground(Color c)
public Font getFont()
public synchronized void setFont(Font f)
public synchronized ColorModel getColorModel()
public void move(int x, int y)
public void resize(int width, int height)
public void resize(Dimension d)
```

```
public synchronized void reshape(int x, int y, int width,
int height)
public Dimension preferredSize()
public Dimension minimumSize()
public void layout()
public void validate()
public void invalidate()
public Graphics getGraphics()
public FontMetrics getFontMetrics(Font font)
public void paint(Graphics g)
public void update(Graphics g)
public void paintAll(Graphics g)
public void repaint()
public void repaint(long tm)
public void repaint(int x, int y, int width, int height)
public void repaint(long tm, int x, int y, int width, int
height)
public void print(Graphics g)
public void printAll(Graphics g)
public boolean imageUpdate(Image img, int flags,int x, int
y, int w, int h)
public Image createImage(ImageProducer producer)
public Image createImage(int width, int height)
public boolean prepareImage(Image image, ImageObserver
observer)
public boolean prepareImage(Image image, int width, int
height, ImageObserver observer)
public int checkImage(Image image, ImageObserver observer)
public int checkImage(Image image, int width, int
height,ImageObserver observer)
public synchronized boolean inside(int x, int y)
public Component locate(int x, int y)
public void deliverEvent(Event e)
public boolean postEvent(Event e)
public boolean handleEvent(Event evt)
public boolean mouseDown(Event evt, int x, int y)
public boolean mouseDrag(Event evt, int x, int y)
public boolean mouseUp(Event evt, int x, int y)
public boolean mouseMove(Event evt, int x, int y)
public boolean mouseEnter(Event evt, int x, int y)
public boolean mouseExit(Event evt, int x, int y)
public boolean keyDown(Event evt, int key)
public boolean keyUp(Event evt, int key)
public boolean action(Event evt, Object what)
public void addNotify()
public synchronized void removeNotify()
public boolean gotFocus(Event evt, Object what)
public boolean lostFocus(Event evt, Object what)
```



```

    public void requestFocus()
    public void nextFocus()
    public String toString()
    public void list()
    public void list(PrintStream out)
    public void list(PrintStream out, int indent)
}

```

(B-heading) Class Usage

In the following we use the term "peer" to indicate the parallel that exists in the native implementation of the component. For example, FileDialog class is a subclass of a Component that has a different implementation on every platform. Java provides an API that looks the same on all platforms. This is accomplished by the creation of a peer class that provides the services needed to allow the API to function properly. This means that the open dialog box created on a Mac is the standard file open dialog box that is supplied by the operating system. Also, that the open dialog box on a windows system is the load file dialog box that is native to windows. Peers are the mechanism by which Java achieves a portable operating system interface.

Suppose the following variables are defined:

```

Boolean aBoolean;
int flags, indent;
ImageProducer anImageProducer;

```

To get the parent of the component:
parent = component.getParent();

To get the peer of the component:
componentPeer = component.getPeer();

To get the tool kit used by the component (see the toolkit class):
toolKit = component.getToolkit();

To see if the peer is known to the component and if the component is properly laid out:
valid = component.isValid();

To see if the Component is visible (the hide method can make this return false):
visible = component.isVisible();

Visible components may not be showing,
visible = component.isShowing();

Enabled components can generate events:
aBoolean = isEnabled();

To unconditionally enable a component:
component.enable();

To conditionally enable a component:
component.enable(cond);

To disable a component:
component.disable();

To get the location of the component relative to the parents' coordinates system:
aPoint = component.location();

To get the dimensions and bounds of the component:
aDimension = component.size();
aRectangle = component.bounds();

To show the component:

```
component.show();
```

To conditionally show the component:

```
component.show(cond);
```

To hide the component:

```
component.hide();
```

To get the components foreground color (or that of its parent):

```
aColor = component.getForeground();
```

To set the components foreground color:

```
component.setForeground(aColor);
```

To get the components background color (or that of its parent):

```
aColor = component.getBackground();
```

To set the components background color:

```
component.setBackground(aColor);
```

To get the font from a component, or that of its parent if the component does not have one set:

```
font = component.getFont();
```

To set the font of a component:

```
component.setFont(font);
```

To get the ColorModel used for display:

```
colorModel = component.getColorModel();
```

To move the Component using the parents' coordinate system:

```
component.move(x, y);
```

To resize the Component:

```
component.resize(width, height);
component.resize(aDimension);
```

To reshape (resize and move) a component, in the parents coordinates:

```
component.reshape(x, y, width, height);
```

To get the preferred or minimum dimensions for the component:

```
aDimension = component.preferredSize();
aDimension = component.minimumSize();
```

To layout the component (typically, validate is used):

```
component.layout();
```

To layout the component and make the component valid:

```
component.validate();
```

To invalidate a component and its parents (marking for layout):

```
component.invalidate();
```

To get a Graphics context for the component (and to return null if there is no peer):

```
component.getGraphics();
```

To get the font metrics for the components peer, if available, otherwise returning the fontmetrics for the component:

```
component.getFontMetrics(font);
```

To paint the component:

```
component.paint(graphics);
```

To update the component (typically called by repaint):

```
component.update(graphics);
```

To paint the component and the subcomponents:

```
component.paintAll(graphics);
```

To schedule a call to update:

```

component.repaint();
To schedule a call to update for a specific interval:
    component.repaint(milliseconds);
    component.repaint(x, y, width, height);
To schedule a call to update for a part of the component in a specific time:
    component.repaint(milliseconds, x, y, width, height);
To paint the component on the graphics context (typically overridden):
    component.print(graphics);
To invoke print on the component and subcomponents:
    component.printAll(graphics);
To repaint the component after the image has been delivered, typically called by an image
observer, returns false if image has no changed:
    aBoolean = component.imageUpdate(anImage, flags, x, y,
width, height);
Creates an image from the image producer see ImageProducer for more information:
    anImage = component.createImage(anImageProducer);
To create an off-screen image for double buffering:
    anImage = component.createImage(width, height);
To prepare an image for for display on the component. This starts a thread see the
ImageObserver:
    component.prepareImage(anImage, anImageObserver);
To prepare the image with a particular width and height:
    component.prepareImage(anImage, width, height,
anImageObserver);
To obtain the status of the screen representation of an image:
    statusInt = component.checkImage(anImage, anImageObserver);
See ImageObserver to decode statusInt. To get the status for an image to be scaled:
    statusInt = component.checkImage(anImage, width, height,
anImageObserver);
To see if a location is inside a components borders, relative to the coordinate system of
the component:
    aBoolean = component.inside(x,y);
To get the component containing a location:
    component = locate( x, y);
To post an event:
    component.deliverEvent(anEvent); // the same as
    postEvent(anEvent)
To call handleEvent with an event on a component (or, if the component does not handle
it, the parent of the component):
    component.postEvent(anEvent);
See the event class of the previous section for a description of the events and how to
handle them. The component provides call back methods:
    public boolean mouseDown(Event evt, int x, int y)
    public boolean mouseDrag(Event evt, int x, int y)
    public boolean mouseUp(Event evt, int x, int y)
    public boolean mouseMove(Event evt, int x, int y)
    public boolean mouseEnter(Event evt, int x, int y)
    public boolean mouseExit(Event evt, int x, int y)

```

```

    public boolean keyDown(Event evt, int key)
    public boolean keyUp(Event evt, int key)
    public boolean action(Event evt, Object what)
    public boolean gotFocus(Event evt, Object what)
    public boolean lostFocus(Event evt, Object what)

```

An invocation of show or pack will call addNotify (the programmer typically does not). AddNotify will make the component invalid and will cause a peer to be created:

```
component.addNotify();
```

To dispose a components peer:

```
component.removeNotify();
```

To get the keyboards focus (i.e., key strokes stream into your component):

```
component.requestFocus();
```

To advance the keyboard focus to the next component:

```
component.nextFocus();
```

To make a string representation of the component:

```
component.toString();
```

To invoke the list method on System.out:

```
component.list();
```

To invoke the list method on a PrintStream:

```
component.list(aPrintStream);
```

To call toString and print on the component and all its, with indentation: children:

```
component.list(aPrintStream, indent);
```

(A-heading) The Container Class

The Container class is an extension of the Component class that can hold Component instances. The instances that are added to an instance of the Container classes are called children. The Container class that holds the children is called the Parent. Since a Container may hold other Containers, there is a possibility of a long line of descendents. A parent is able to arrange the appearance of the children using a layout manager. Each child has its own layout manager. If a container (or its children) are not properly laid out, the layout is said to be invalid. A parent will layout the children and all their descendents during the layout process. Containers that are valid will not be laid out needlessly. From a knowledge representation point of view, the class structure is that of an AKO (A-Kind-Of) taxonomy. This is different from the Container hierarchy. With the Container hierarchy there is a has-a relationship. For example, a Frame has-a Panel that has-a Checkbox. Both a Frame and a Panel are kinds-of Containers. Containers always have an add method that provides the implementation for adding either components, or other containers to the has-a hierarchy. A Container instance may handle events with its own handleEvent implementation. Recall that getParent is inherited by any sub-class of Component (i.e., the Container class).

(B-heading) Class Summary

```

public abstract class Container extends Component
    public int countComponents()
    public synchronized Component getComponent(int n)
    public synchronized Component[] getComponents()
    public Insets insets()
    public Component add(Component comp)

```

```

    public synchronized Component add(Component comp, int
pos)
    public synchronized Component add(String name,
Component comp)
    public synchronized void remove(Component comp)
    public synchronized void removeAll()
    public LayoutManager getLayout()
    public void setLayout(LayoutManager mgr)
    public synchronized void layout()
    public synchronized void validate()
    public synchronized Dimension preferredSize()
    public synchronized Dimension minimumSize()
    public void paintComponents(Graphics g)
    public void printComponents(Graphics g)
    public void deliverEvent(Event e)
    public Component locate(int x, int y)
    public synchronized void addNotify()
    public synchronized void removeNotify()
    public void list(PrintStream out, int indent)
}

```

(B-heading) Class Usage

Suppose that the following variables are already defined:

```

Component comp, compArray[];
Container cont;
Insets insets;
int ncomponents, i;
String layoutName; // the name of a layout manager.
LayoutManager layMan;
Graphics gc;
int x,y;
PrintStream printStream;
int indent;

```

To get the number of components:

```
ncomponents = comp.countComponents();
```

To get the nth component:

```
comp = cont.getComponent(i);
```

To get all the components:

```
compArray = cont.getComponents();
```

To get the insets (rectangular dimensions, in pixels from the edges):

```

insets = cont.insets();
i = insets.bottom;
i = insets.top;
i = insets.left;
i = insets.right;

```

To add a Component to a Container:

```
cont.add(comp);
```

To add a Component to a Container in a given order (-1 means end of list):

```
cont.add(comp, i);
```

To add a Component to a Container using a layout manager:

```
cont.add(layoutName, i);
```

To remove a Component from a Container:

```
cont.remove(comp);
```

To delete all Components:

```
cont.removeAll();
```

To get and set the layout manager:

```
layMan = cont.getLayout();
```

```
cont.setLayout(layMan);
```

To perform layout on the Container but not the descendents:

```
cont.layout();
```

To perform layout on the Container and all descendents:

```
cont.validate();
```

To get the preferred and minimum dimensions:

```
Dimension dim = cont.preferredSize();
```

```
dim cont.minimumSize();
```

To paint or print the Components onto a Graphics instance:

```
cont.paintComponents(gc);
```

```
cont.printComponents(gc);
```

From within a Container (like a Frame) one may write:

```
printComponents(getGraphics());
```

```
paintComponents(getGraphics());
```

To post an Event to a Component, located at (e.x, e.y):

```
Event e;
```

```
cont.deliverEvent(e);
```

To locate a component at (x, y):

```
comp = cont.locate(x, y); // null returned if component
unfound
```

To send the addNotify message to all descendents and the super class:

```
cont.addNotify(); // this creates peer descendents
```

To send the removeNotify message to all descendents and the super class:

```
cont.removeNotify(); // this deletes peer descendents
```

To print out a list of the descendent:

```
cont.list(printStream, indent);
```

The indent is the indentation used during generation traversal. For example, (BEGIN CDROM) in AudioFrame.java, a MenuItem event is handled by:

```
if (Evt.match(e, print_mi)) {
    validate();
    list(System.out, 5);
    return true;
}
```

This emits a component list, including 4 scrollbars, a Panel, a Label and an IntLabel:

```
lyon.AudioFrame[0,0,550x385,layout=java.awt.BorderLayout,title=/hd/current/Java%20book/code/au/ah.au]
java.awt.Scrollbar[0,0,550x16,val=14,vis=true,min=0,max=24,horz]
java.awt.Scrollbar[0,354,550x16,val=0,vis=true,min=0,max=3469,horz]
```

```

java.awt.Scrollbar[0,23,16x331,val=0,vis=true,min=-
300,max=300,vert]
java.awt.Scrollbar[534,23,16x331,val=8,vis=true,min=0,max=1
1,vert]
java.awt.Panel[0,0,550x23,layout=java.awt.FlowLayout]
java.awt.Label[206,5,101x13,align=left,label=Number of
Samples:]
lyon.IntLabel[312,5,32x13,align=left,label=3469]

```

This can be useful for debugging purposes. The Frame instance may be seen in Figure 3.8, the Digital Oscilloscope.

(A-heading) The Frame Class

The Frame class is descendent of the Window class (which in-turn descends from the Container and Component classes). As an extension of the Window class, it add the features of a title bar, menu bar, border, cursor and an icon image. The Frame implements the MenuContainer and so a Frame may contain MenuComponents.

(B-heading) Class Summary

```

package java.awt;
public class Frame extends Window implements MenuContainer
    public static final int  DEFAULT_CURSOR
    public static final int  CROSSHAIR_CURSOR
    public static final int  TEXT_CURSOR
    public static final int  WAIT_CURSOR
    public static final int  SW_RESIZE_CURSOR
    public static final int  SE_RESIZE_CURSOR
    public static final int  NW_RESIZE_CURSOR
    public static final int  NE_RESIZE_CURSOR
    public static final int  N_RESIZE_CURSOR
    public static final int  S_RESIZE_CURSOR
    public static final int  W_RESIZE_CURSOR
    public static final int  E_RESIZE_CURSOR
    public static final int  HAND_CURSOR
    public static final int  MOVE_CURSOR
    public Frame()
    public Frame(String title)
    public synchronized void addNotify()
    public String getTitle()
    public void setTitle(String title)
    public Image getIconImage()
    public void setIconImage(Image image)
    public MenuBar getMenuBar()
    public synchronized void setMenuBar(MenuBar mb)
    public synchronized void remove(MenuComponent m)
    public synchronized void dispose()
    public boolean isResizable()
    public void setResizable(boolean resizable)
    public void setCursor(int cursorType)
    public int getCursorType()

```

```
}
(B-heading) Class Usage
```

Suppose that

```
Frame f = new Frame("Title of Frame");
```

is predefined. The Frame constructor is overloaded. You may either pass a string (which becomes the title of the frame) or you may leave the frame untitled:

```
Frame foo = new Frame();
Frame titledFrame = new Frame("Here is a title");
```

Frame instances always start life as being invisible and with a layout called BorderLayout. To get and set the title of a frame use:

```
String title = f.getTitle();
f.setTitle("My title");
```

To get and set the frames icon image:

```
Image icon = f.getIconImage();
f.setIconImage(icon);
```

To get and set the frames menu bar:

```
MenuBar mb = f.getMenuBar();
f.setMenuBar(mb);
```

The getMenuBar method returns null if no menu bar was set.

To remove the menu bar:

```
f.remove(m);
```

To dispose of the frame:

```
f.dispose;
```

To see if the frame is resizable:

```
aBoolean = f.isResizable();
```

To set the resizabililty:

```
f.setResizable(aBoolean);
```

The cursor is set using constants that are end with `_CURSOR`. To set the cursor to one of the preset cursors, use one of:

```
f.setCursor(DEFAULT_CURSOR);
f.setCursor(CROSSHAIR_CURSOR);
f.setCursor(TEXT_CURSOR);
f.setCursor(WAIT_CURSOR);
f.setCursor(SW_RESIZE_CURSOR);
f.setCursor(SE_RESIZE_CURSOR);
f.setCursor(NW_RESIZE_CURSOR);
f.setCursor(NE_RESIZE_CURSOR);
f.setCursor(N_RESIZE_CURSOR);
f.setCursor(S_RESIZE_CURSOR);
f.setCursor(W_RESIZE_CURSOR);
f.setCursor(E_RESIZE_CURSOR);
f.setCursor(HAND_CURSOR);
f.setCursor(MOVE_CURSOR);
```

To get the cursor:

```
anInt = f.getCursorType();
```

It is typical for event handling to be performed within a frame. It is also typical for speciality frames to exist that are able to handle specific events. As an example, we show the ClosableFrame.

(B-heading) The ClosableFrame Class

The ClosableFrame is an extension of the Frame class that handles the close window event. We have found that it is often desirable to have windows that respond to the close event and that this event is typically handled with the following code fragment:

```
        if(e.id == e.WINDOW_DESTROY) {
            hide();
            return true;
        }
```

Rather than repeat the same code in every event handler of every Frame instance, we have devised the ClosableFrame class. The ClosableFrame is given in the following code:

```
package lyon.gui;
import java.awt.*;
public class ClosableFrame extends Frame {
    // constructor needed to pass window title to class
    Frame
    public ClosableFrame(String name) {
        // call java.awt.Frame(String) constructor
        super(name);
    }
    // needed to allow window close
    public boolean handleEvent(Event e) {
        // Window Destroy event
        if (e.id == Event.WINDOW_DESTROY) {
            dispose();
            return true;
        }

        // it's good form to let the super class look at
        any unhandled events
        return super.handleEvent(e);
    } // end handleEvent()
} // end class ClosableFrame
```

Any Frame that extends the ClosableFrame will inherit the ability to handle the WINDOW_DESTROY event, provided the subclass returns super.handleEvent.

To take advantage of the ClosableFrame we extend the ClosableFrame as Follows:

```
...
public class AudioFrame extends ClosableFrame {
    ...
    public AudioFrame(String name) {
        super(name);
    }
}
```

(BEGIN NOTE)

The constructor of the super class is called with the invocation of *super(name)*. This is so the call to the *java.awt.Frame(String)* will eventually be made.

(END NOTE)

(A-heading) The Panel Class

The Panel class is a generic extension of the Container class. It is often used to organize the layout of components into structured sub-parts. For example, in Figure 3.10, we see an example of the HTML Generator Panel. The Panel instance has several components, including buttons, pop-up menus and labels. These components are arranged to give the user a functional presentation of the input and output components. The Panel Class represents a low-overhead Container with its own layout manager instance. Further, a Panel can contain its own event handler. This permits a more object oriented approach to the event handling.

(B-heading) Class Summary

```
public class Panel extends Container {
    public Panel()
    public synchronized void addNotify()
}
```

(B-heading) Class Usage

The default LayoutManager for the panel is the FlowLayout. Suppose the following variable is predefined:

```
Panel p = new Panel();
```

To create a peer for the Panel use:

```
p.addNotify();
```

Beyond the addNotify and constructor (with default layout), the Panel inherits all of its methods and properties from the Container Class.

(B-heading) Building a Panel

In the following code we see several components (some of which will be discussed later) being added to a panel. (BEGIN CDROM) These are in the TargetControlPanel.java file, a part of the HTML Generator interface: (END CDROM)

```
package htmlconverter;
import java.awt.*;
public class TargetControlPanel extends Panel {
    Choice c;
    TargetControlPanel() {
        c = new Choice();
        c.addItem("Java");
        c.addItem("C");
        c.addItem("C++");
        c.select("Java");
        setLayout(new GridLayout(1, 3, 10, 10));
        add(new Label("Target:", Label.LEFT));
        add(c);
    }
}
```

(A-heading) The Checkbox Class

The Checkbox class extends the Component class. As such, it is one of the many interface widgets that may be added to any Container subclass. Typically, a Checkbox

instance is added to a Panel or a Frame. The state of the Checkbox instance becomes true if it is checked, otherwise it is false. The nice thing about a Checkbox instance is, you do not have to handle the Checkbox event. Checkbox instances may be incorporated directly into Java programs.

Checkbox instances may be grouped into another instance called a CheckboxGroup. The common name for a CheckboxGroup is a radio button. Radio buttons are a paradigm of the push-button favorite station selection feature available on some car radios. The favorite stations were typically assigned to the buttons. The user would select one station with the push of the button. The basic rule of the radio button interface is that you may select only one button out of the many. Another feature is that the act of selecting one button, deselects the others. Further, there is always one button selected.

(B-heading) Class Summary

```
public class Checkbox extends Component {
    public Checkbox()
    public Checkbox(String label)
    public Checkbox(String label, CheckboxGroup group, boolean
state)
    public synchronized void addNotify()
    public String getLabel()
    public void setLabel(String label)
    public boolean getState()
    public void setState(boolean state)
    public CheckboxGroup getCheckboxGroup()
    public void setCheckboxGroup(CheckboxGroup g)
}
```

(B-heading) Class Usage

Suppose the following variables are predefined:

```
String label;
CheckboxGroup group;
boolean state;
Checkbox cb;
```

To make an instance of an unchecked Checkbox without a label, group:

```
cb = new Checkbox();
```

To make an instance of an unchecked Checkbox with a label and no group:

```
cb = new Checkbox(label);
```

To make an instance of a Checkbox with a label, group and known state:

```
cb = new Checkbox(label, group, state);
```

To make the peer of a Checkbox:

```
cb.addNotify();
```

To get and set the label of a Checkbox:

```
label = cb.getLabel();
cb.setLabel(label);
```

To get and set the state of the Checkbox:

```
state = cb.getState();
cb.setState(state);
```

To get and set the group of the Checkbox:

```
group = cb.getCheckboxGroup();
cb.setCheckboxGroup(group);
```

(B-heading) Adding Checkboxes to Frames

(BEGIN CDROM) In GUI.java, the checkboxes for the DiffCAD Frame are placed into an array of checkboxes. The array is used to manipulate the checkboxes with fewer lines of code. The checkboxes may be see in Figure 3.9, The DiffCAD MainFrame.

(END CDROM)

```

1.  ...
2.  Checkbox ap_cb = new Checkbox("Auto-pan",null,true);
3.  Checkbox order_cb = new Checkbox("-order",null,true);
4.  Checkbox i3_cb = new Checkbox("-i3");
5.  Checkbox r3_cb = new Checkbox("-r3",null,true);
6.  Checkbox x3_cb = new Checkbox("-x3",null,false);

```

(BEGIN NOTE) To use a Checkbox with a known state and no Checkbox group, you must place a null, as a place holder, into the group parameter. (END NOTE)

```

7.  Checkbox checkboxes[] = {
8.      ap_cb,
9.      order_cb,
10.     i3_cb,
11.     r3_cb,
12.     x3_cb};

```

To add the checkboxes to a frame, we use a method called addCheckBoxes:

```

public void addCheckBoxes( Checkbox checkboxes[] ) {
    for (int i=0; i < checkboxes.length; i++) {
        add(checkboxes[i]);
    }
}

```

To perform computation with a Checkbox, you need only get its state. For example:

```

if (x3_cb.getState()) {
    p.x_data[i] = p.x_data[i] * -1;
}

```

It is possible to make the Checkbox event trigger computation. In some cases, this may be the preferred mode; however, if computation is going to lag behind the users input rate, then feedback to the user will be required and the program may seem sluggish [Tog]. Since there are several checkboxes, the approach taken by the DiffCAD program is only to update the screen with a recomputation during a *repaint*. This policy permits the embedding of Checkbox states directly into equations.

(A-heading) The Scrollbar Class

The Scrollbar class is a component subclass. It is generally added to a Container class instance and is used to obtain a numeric quantity from the user. The scrollbar excels at permitting a user to scroll data, such as an image or a graph, in a window of limited size. In the digital oscilloscope example (see Figure 3.8) there are 4 scrollbars, top, bottom, left and right. Their functions will be described in the Class Usage section. The box that slides along the scrollbar is typically called the elevator. Taking the elevator up is the same as clicking and holding on the box while dragging the pointing devices' cursor to the top of the scrollbar.

(B-heading) Class Summary

```

public class Scrollbar extends Component {
    public static final int      HORIZONTAL

```

```

public static final int      VERTICAL
public Scrollbar()
public Scrollbar(int orientation)
public Scrollbar(int orientation, int value, int visible,
int minimum, int maximum)
public synchronized void addNotify()
public int getOrientation()
public int getValue()
public void setValue(int value)
public int getMinimum()
public int getMaximum()
public int getVisible()
public void setLineIncrement(int l)
public int getLineIncrement()
public void setPageIncrement(int l)
public int getPageIncrement()
public void setValues(int value, int visible, int minimum,
int maximum)
}

```

(B-heading) Class Usage

Suppose that the following variables are predefined:

```

int orientation;
int visible;
int minimum, maximum;
int value;

```

To construct a `Scrollbar` instance, you

`Scrollbar sb;`

The constructor defaults to a vertical scrollbar:

```

sb = new Scrollbar();
int orientation;

```

The orientation, one of: `Scrollbar.HORIZONTAL` or `Scrollbar.VERTICAL`.

```

sb = new Scrollbar(orientation);
int value;
int pageSize, minimum, maximum;

```

To make a `Scrollbar` instance with given orientation, value, visibility, minimum and maximum range:

```

sb = new Scrollbar(orientation, value, visible, minimum,
maximum);

```

To make a peer for the `Scrollbar` instance:

```

sb.addNotify();

```

To get the orientation of the `Scrollbar`:

```

orientation = sb.getOrientation();

```

To get and set the value:

```

value = sb.getValue();
sb.setValue(value);

```

Value is clipped to Maximum or Minimum if it is not between them.

To get the minimum and maximum of the scrollbar:

```

value = sb.getMinimum();

```

```

    value = sb.getMaximum();
To get the amount visible, in pixels:
    value = sb.getVisible();
To set and get the elevator increment for an arrow click:
    sb.setLineIncrement(1);
    l = sb.getLineIncrement();
To set and get the elevator increment for a page click:
    sb.setPageIncrement(1);
    l = sb.getPageIncrement();
To set the values (value is clipped if it exceeds the valid range):
    sb.setValues(value, visible, minimum, maximum);
}

```

(B-heading) Adding Four Border Scrollbars

In this section we show how to construct the four border scrollbars shown in Figure 3.8, The Digital Oscilloscope generator. An oscilloscope is a test instrument used to graph wave forms. The digital oscilloscope is intended to have some of the features of the traditional instrument (which can typically be purchased at significant cost). The following code illustrates how to add scrollbars to the AudioFrame (which is used to display the digital oscilloscope):

```

1. public class AudioFrame extends ClosableFrame {
2. ...

```

In lines 3-6 we create the horizontal and vertical scroll bars both (bottom, top) and (left, right).

```

3. Scrollbar sbHorzBottom = new
Scrollbar(Scrollbar.HORIZONTAL);
4. Scrollbar sbHorzTop = new
Scrollbar(Scrollbar.HORIZONTAL);
5. Scrollbar sbVertLeft = new
Scrollbar(Scrollbar.VERTICAL);
6. Scrollbar sbVertRight = new
Scrollbar(Scrollbar.VERTICAL);

```

For our digital oscilloscope, we want sbVertLeft to translate the wave form vertically. The scrollbar, sbVertRight, will scale the wave form in height. The scale factors for the wave form are given in millivolts per division. We want divisions in millivolts to reflect the steps of the physical instrument (from 5 volts to 1 millivolts). These steps become our y-scale factors. We also want the x-scale factors to alter the number of seconds used per division. To reflect the physical instrument we proceed in steps from 0.05 microseconds per division to 5 seconds per division. This models the steps and range of a circa 1969 dual-trace 150 Mhz bandwidth Tektronix model 7704 portable oscilloscope. The 7704 weighs over 50 lbs and is portable by virtue of having two handles.

```

7. private int dx = 0;
8. private int oldDx = 0;
9. private int dy = 0;
10. private int oldDy = 0;
11. private final double xScaleFactors[] = {50000, 25000,
10000, 5000, 2500, 1000,
12. 250, 100, 50, 25, 10,
13. 2.5, 1, .5, .25, .1,

```

```

14. .025, .01, .005, .0025, .001,
15. .0005};
16. private final String xSFLabels[] = {"0.05 u", "0.1 u",
    "0.25 u",
17. "0.5 u", "1 u", "2.5 u",
18. "5 u", "10 u", "25 u",
19. "50 u", "100 u", "250 u",
20. "500 u", "1 m", "2.5 m",
21. "5 m", "10 m", "25 m",
22. "50 m", "100 m", "250 m",
23. "500 m", "1 ", "2.5 ", "5 "};
24. private final int xsfStartIndex = 14;
25. private double xScaleFactor =
xScaleFactors[xsfStartIndex];
26. private double oldXScaleFactor =
xScaleFactors[xsfStartIndex];
27. private String xSFLLabel = new
String(xSFLabels[xsfStartIndex]);

28. private final double yScaleFactors[] = {500, 200, 100,
50, 20, 10,
29. 2, 1, .5, .2, .1};

30. private final String ySFLabels[] = {"1 m", "2.5 m", "5
m", "10 m", "25 m", "50 m",
31. "100 m", "250 m", "500 m", "1 ", "2.5 ", "5 "};

32. private final int ysfStartIndex = 8;
33. private double yScaleFactor = 1;
34. private double oldYScaleFactor = 1;
35. private String ySFLLabel = new
String(ySFLabels[ysfStartIndex]);
36. ...

37. public boolean handleEvent(Event e) {
38. ...

```

During the handling of the event, we may check to see if the event target contains the same reference as one of the components. This is done in line 39.

```

39. if(e.target == sbHorzTop) {
40.     int i = sbHorzTop.getValue();
41.     xScaleFactor = xScaleFactors[i];

```

For line 42, we check to see if the scale factor has changed, if it has not we do not call repaint. Repaint is computationally expensive.

```

42.     if (xScaleFactor != oldXScaleFactor) {
43.         xSFLLabel = xSFLabels[i];
44.         repaint();
45.         oldXScaleFactor = xScaleFactor;

```

```

46.     }
47.     return true;
48. }

```

In order to make the scrollbars border the `AudioFrame`, we use a layout manager called `BorderLayout`:

```

1. public AudioFrame(String name) {
2. ...
3.  setLayout(new BorderLayout());
4.  add("North", sbHorzTop);
5.  add("South", sbHorzBottom);
6.  add("West", sbVertLeft);
7.  add("East", sbVertRight);

8.  // openAudioStream will set the fileName
9.  // and audioStream variables.
10. openAudioStream();
11. // Set top scrollbar characteristics
12. sbHorzTop.setValues(xsfStartIndex, 0, 0, 24);
13. ...

```

(A-heading) The Label Class

The `Label` is a subclass of the `Component` class, and as such is generally added to a container. Labels are typically not generators of useful events. A label may be updated dynamically to reflect the state of an underlying variable. For example, in the (BEGIN CDROM) `DiffCAD` program in the `GUI.java` file there are both static labels and dynamic labels. Dynamic labels are discussed in a later section. (END CDROM)

(B-heading) Class Summary

```

public class Label extends Component {
    public static final int LEFT
    public static final int CENTER
    public static final int RIGHT
    public Label()
    public Label(String label)
    public Label(String label, int alignment)
    public synchronized void addNotify()
    public int getAlignment()
    public void setAlignment(int alignment)
    public String getText()
    public void setText(String label)
}

```

(B-heading) Class Usage

Suppose the following variables are predefined:

```

Label l;
String name;
int alignment;

```

Alignment is set to one of:

```

Label.LEFT, Label.RIGHT, Label.CENTER.

```



```

public class Label extends Component {
To construct an empty Label:
    l = new Label();
To construct a named label:
    l = new Label(name);
To construct a named label with known alignment:
    l = new Label(name, alignment);
To make the label peer
    l.addNotify();
To get and set alignment:
    alignment = l.getAlignment();
    l.setAlignment(alignment);
To get and set the string that the labels contains:
    name = l.getText();
    l.setText(name);
}

```

(B-heading) Adding Labels to Frames

To add labels to a Frame, you need only invoke:

```
add(new Label("Grid:"));
```

No event handler code is required. Also, any extension of the Container class can add components. For example

(A-heading) The Choice Class

The Choice Component class provides a pop-up menu of string choices. The currently selected string becomes the title of the string menu. In many ways, the Choice Component is like a main menu bar selection that continue to read out its last selection. Pop-up menu examples may be seen in Figure 3.9, The DiffCAD Main Frame and Figure 3.10, The HTML Generator Panel.

(B-heading) Class Summary

```

public class Choice extends Component {
    public Choice()
    public synchronized void addNotify()
    public int countItems()
    public String getItem(int index)
    public synchronized void addItem(String item)
    public String getSelectedItem()
    public int getSelectedItemIndex()
    public synchronized void select(int pos)
    public void select(String str)
}

```

(B-heading) Class Usage

There is only a single constructor for the Choice Component. Suppose the following variables are predefined:

```

Choice choice = new Choice();
int i;
String str;

```

To create the peer:

```
choice.addNotify();
```

To find the number of strings in choice:

```
i = choice.countItems();
```

To get the String at location i:

```
str = choice.getItem(i);
```

To add a String:

```
choice.addItem(str);
```

To get the selected String:

```
str = choice.getSelectedItem();
```

To get the location of the String in the list:

```
i = choice.getSelectedIndex();
```

To select a String, by position or name, thereby making it visible:

```
choice.select(i);
```

```
choice.select(str);
```

```
}
```

(B-heading) Adding Choices to a Frame

In this example, we show how to add a Choice menu to a Frame. (BEGIN CDROM) The code is excerpted from the DiffCAD program in GUI.java.(END CDROM) The basic idea is that a graph object may be selected by choosing it from the Choice instance.

```
public class GUI extends Applet implements constants {
    Choice mouseChoice;
    String objectPropertiesStr = new String("Object
Properties");
    String cameraStr = new String("camera");
    String laserStr = new String("laser");
    String laserAngleStr = new String("laser angle");
    String gratingStr = new String("grating");
    String graphPvsXStr = new String("graph p vs x");
    String graphDLvsXStr = new String("graph DL vs x");
    String allStr = new String("all");

    String mouseChoiceItems[] = {
        objectPropertiesStr,
        cameraStr, laserStr, laserAngleStr, gratingStr,
        graphPvsXStr, graphDLvsXStr, allStr
    };
};
```

To assist in the creation of Choice menus, there is graphics utilities class that is in the called Guitils:

```
public class Guitils {
    static public Choice addChoiceMenu(String items[],
Container cont) {
        Choice choice = new Choice();
        for (int i = 0; i < items.length; i++) {
            choice.addItem(items[i]);
        }
        cont.add(choice);
        return choice;
    }
};
```

} ...
 GUI.java makes use of the `Guitils.addChoiceMenu` method in `create_gui`:

```
public void create_gui() {
    setBackground(Color.white);
    Guitils.addCheckBoxes(checkboxes, this);
    mouseChoice = Guitils.addChoiceMenu(mouseChoiceItems,
    this);
```

Since the `Frame` class is a subclass of the `Container` class, the usage of *this* simple passes the `Frame`, with automatic type casting, into the `Guitils.addCheckBoxes` method. The `Guitils.addCheckBoxes` method did not keep a reference to the `Choice` instance, *choice*, because the `Container` instance will store it.

To handle the choice, we have adopted the policy of waiting for the mouse to be released. Once this occurs, we can process the choice:

```
1. public boolean handleEvent(Event e) {
2.     switch (e.id) {
3.         case Event.MOUSE_UP: {
4.             String objectName =
mouseChoice.getSelectedItem();
5.             processMouseUp( objectName, e.x, e.y);
6.             repaint();
7.         }
8.         case Event.MOUSE_DOWN:
9.             anchor = new point( e.x, e.y);
```

In line 9, we store the mouse down event so that we can compute the relative mouse motion when the mouse is released.

```
10.         return super.handleEvent(e);
11.     } // switch
```

The invocation of the `processMouseUp` method causes a sequence of *if-then-else* methods to be fired:

```
1. public void processMouseUp(String object_name, int x,
int y) {
```

The *rmove* point instance in line 2 is used to compute the relative mouse motion. The upside down nature of the coordinate system, imposed by the AWT designers, requires that we negate the $(y - \text{anchor.y})$ displacement. The use of an upside-down coordinate system was probably an unfortunate design choice.

```
2.     point rmove = new point(x-anchor.x, anchor.y - y);
3.     // rmove is the relative motion of the mouse.
```

It is much faster to compare strings using a check between references. This is safe, because the strings were added to the `Choice` instance by reference. It is NOT safe to use:

```
// if (object_name == "camera") // <--- this is NOT safe
```

Also, it is bad software engineering practice to embed string literals throughout the code, since typos can result in hard-to-find bugs:

```
// if (object_name.equals("camera")) // <--- this is bad
engineering!
```

The best way is to allow the compiler a chance to check for type correctness. This way, if there is a typo, the compiler will emit a syntax error, rather than require run-time debugging. Further, the reference comparison, using `==` is very fast.

```

4.         if ( object_name== cameraStr) {
5.         Camera c = (Camera) Geometry.camera;
6.         c.auto_pan = ap_cb.getState();
7.         c.change_config(rmove);

8.         Geometry.compute_rays();

9.     } else if ( object_name== laserStr) {
10.        Laser l = (Laser) Geometry.laser;
11.        l.change_config(rmove);
12.        Geometry.compute_rays();
13.    } else if ( object_name == gratingStr) {
14.        Grating g = (Grating) Geometry.grating;
15.        g.change_config(rmove);
16.        Geometry.compute_rays();
17.    } else if ( object_name == graphDLvsXStr)
18.        make_graph_dl();
19.    else if ( object_name == graphPvsXStr)
20.        make_graph_p();
21. else if (object_name == laserAngleStr) {
22.        Wedge w = (Wedge) Geometry.wedge;
23.        w.change_config(rmove);
24.        Geometry.compute_rays();
25.        return;
26.    } else if (object_name == allStr) {
27.    Xform.rmove(rmove);
28.    return;
29.    }

```

(A-heading) Summary

This chapter introduced some of the basic classes in Java needed to perform basic AWT programming. In the following chapters we will present a series of subclass that extend the component class. (BEGIN CDROM) On the CDROM we have a program, called DiffCAD, that allows us to present several types of components in various configurations. In later chapters we will address the creation of some of these interfaces. As a sample, we show an image of a digital oscilloscope (shown in Figure 3.8), an optical raytracer for designing diffraction rangefinders (shown in Figure 3.9), an HTML , able to read batches of Java, C and C++ files, transforming them into HTML (shown in Figure 3.10).

(END CDROM)

Figure 3.8 The Digital Oscilloscope
 Figure 3.9 The DiffCAD Main Frame
 Figure 3.10 The HTML Generator Panel

(CN) 4. Futil Recipes for Feudal Times

From cooking meals for hungry hired men
 And washing dishes after them
 -Robert Frost
 Resistance is Futile
 - Borg

(BEGIN ON CDROM) This book comes with a package called *futils*. (END ON CDROM) The *futils* package is a collection of file utilities that simplify the writing of user-friendly GUI-based code. The *java.io* package is responsible for enabling all input-output (I/O) operations from Java. Due, in part, to the number and variety of operations that *java.io* is responsible for, *java.io* has grown to 31 different class files. Eventually, the programmer will want to learn them all. However, even a seasoned programmer will want to develop a set of useful tools that are less general and are therefore easier to use. The premise is that simplicity is inversely related to generality and that “simplicity” is a feature! Thus, the *futil* package presents an elementary set of simple tools pragmatically designed to be both easy to use and to make compact code that is easy read.

File utilities are, by their very nature, considered to be “unsafe”. File utilities can rename, copy, list and even delete files! This is all the more reason why there should be a centralized package of well-tested and well-understood classes for file manipulation. Further, due to security manager restrictions, classes in the *futil* package will generally not run within a browser.

The *futil* package has several public final classes, each of which has several static methods. To gain access to these methods, type:

```
import futils.*;
```

at the head of your Java source.

The *Futil* class is a public final class that resides in the *futils* package. The *Futil* class contains a single private constructor that is used to prevent instantiation of the class. The *Futil* class is dependent on the *java.io* package.

(A-heading) The Dialog Class

An instance of a *Dialog* class is like a low-overhead *Frame* instance. A *Dialog* instance may be resizable. A *Dialog* instance has a title, a border and a layout.

A *Dialog* instance has a modal property. If a *Dialog* instance is modal, then the user is forced to interact with it. If a *Dialog* instance is non-modal, then the user may choose other *Window* instances (like *Frame* instances). The default layout is *BorderLayout*.

(B-heading) Class Summary

```
public class Dialog extends Window {
    public Dialog(Frame parent, boolean modal)
    public Dialog(Frame parent, String title, boolean
modal)
    public synchronized void addNotify()
    public boolean isModal()
    public String getTitle()
    public void setTitle(String title)
    public boolean isResizable()
    public void setResizable(boolean resizable)
}
```

(B-heading) Class Usage

Suppose the following variables are predefined:

```
String title;
boolean resizable, modal;
Dialog dialog;
Frame parent;
```

To make an (initially invisible) instance of a Dialog:

```
dialog = new Dialog(parent, modal);
```

To make an (initially invisible) instance of a Dialog with a title:

```
dialog = new Dialog(parent, title, modal);
```

To create a peer:

```
dialog.addNotify();
```

To see if a Dialog instance is modal:

```
modal = dialog.isModal();
```

To get and set the title:

```
title = dialog.getTitle();
dialog.setTitle(title);
```

To get and set the resizable property:

```
resizable = dialog.isResizable();
dialog.setResizable(resizable);
```

(A-heading) The FileDialog Class

The FileDialog class is a subclass of the Dialog class. Instances of the FileDialog class are used by the `futils` package to obtain file and directory information from the user. An instance of the FileDialog class will block the invoking thread when shown. The FileDialog class is always modal.

(B-heading) Class Summary

```
public class FileDialog extends Dialog
{
    public static final int LOAD
    public static final int SAVE
    public FileDialog(Frame parent, String title)
    public FileDialog(Frame parent, String title, int mode)
    public synchronized void addNotify()
    public int getMode()
    public String getDirectory()
    public void setDirectory(String dir)
    public String getFile()
    public void setFile(String file)
    public FilenameFilter getFilenameFilter()
    public void setFilenameFilter(FilenameFilter filter)
}
```

(B-heading) Class Usage

Suppose the following variables are predefined:

```
FileDialog fileDialog;
Frame parent;
String title, path;
int mode;
FilenameFilter filter;
```

Mode must be one of `FileDialog.LOAD` or `FileDialog.SAVE`

To create a `FileDialog` instance for reading a file:

```
fileDialog = new FileDialog(parent, title);
```

To specify a mode during instantiation:

```
fileDialog = new FileDialog(parent, title, mode);
```

To get the mode:

```
mode = fileDialog.getMode();
```

To get and set the directory:

```
path = fileDialog.getDirectory();
```

```
fileDialog.setDirectory(path);
```

To get the file name:

```
path = fileDialog.getFile();
```

To set the default file name before the dialog is shown:

```
fileDialog.setFile(path);
```

To get and set the `FilenameFilter`:

```
filter = fileDialog.getFilenameFilter();
```

```
fileDialog.setFilenameFilter(filter);
```

(B-heading) Futil.getReadFileName

The `Futil.getReadFileName` is a static method that creates a file open dialog box. The dialog box is used by the user to select a file. Once the file is selected, the `Futil.getReadFileName` returns an absolute path name to the file as a `String` instance. An example of the standard file open dialog box is shown in Figure 4.1.

Figure 4.1. The Standard File Open Dialog Box

The appearance of the file open dialog box will be platform dependent.

```
public static String getReadFileName() {
    FileDialog dialog = new FileDialog(new Frame(), "select
    file");
    dialog.show();
    String file_name = dialog.getFile();
    String path_name = dialog.getDirectory();
    String file_string = path_name + file_name;
    System.out.println("Opening file: "+file_string);
    dialog.dispose();
    return file_string;
}
```

(BEGIN NOTE)The parent field of the dialog is set to `new Frame()` so that the user does not need to pass a `Frame` instance into the `getReadFileName` parameter list. This simplifies the `getReadFileName` call at the expense of efficiency. The path name for the file is explicitly concatenated into the `file_string` before the return. (END NOTE)

Typically a programmer would prompt the user to select a file by using:

```
String inputFileName = Futil.getReadFileName();
```

(B-heading) Futil.writeFileName

To get a file name for write, the mode of the `FileDialog` must be set to `FileDialog.SAVE`. A sample of the save file dialog box is shown in Figure 4.2.

Figure 4.2. Save File Dialog Box

This is done in the following code:

```
public static String getWriteFileName() {
    FileDialog dialog = new FileDialog(new Frame(), "Enter
file name",FileDialog.SAVE);
    dialog.show();
    String fs = dialog.getDirectory() + dialog.getFile();
    System.out.println("Opening file: "+fs);
    dialog.dispose();
    return fs;
}
```

To ask the user for a file string, with a fully qualified path name, use:

```
String writeFile = Futil.getWriteFileName();
```

(A-heading) The File Class

The File class resides in the java.io package. An instance of the File class keeps track of several file related properties, including: a path to the file, information on whether the path is relative or absolute and a series of static constants that indicate the path separator. It is unfortunate that on some systems (i.e., DOS, Windows, etc.) the path name separator is represented by a back-slash ('\'), whereas on other systems (i.e., Macintosh and Unix variants) the path name separator is represented by a forward-slash ('/').

The File class makes use of FilenameFilter instance. These are introduced in the following section on FilenameFilters.

(B-heading) Class Summary

```
public class File {
    public static final String separator
    public static final char separatorChar
    public static final String pathSeparator
    public static final char pathSeparatorChar
    public File(String path)
    public File(String path, String name)
    public File(File dir, String name)
    public String getName()
    public String getPath()
    public String getAbsolutePath()
    public String getParent()
    public boolean exists()
    public boolean canWrite()
    public boolean canRead()
    public boolean isFile()
    public boolean isDirectory()
    public native boolean isAbsolute();
    public long lastModified()
    public long length()
    public boolean mkdir()
    public boolean renameTo(File dest)
    public boolean mkdirs()
    public String[] list()
    public String[] list(FilenameFilter filter)
```



```

    public boolean delete()
    public int hashCode()
    public boolean equals(Object obj)
    public String toString()
}

```

(B-heading) Class Usage

Suppose the following variables are predefined:

```

String absPath = Futil.getReadFileName();
// The fileName is relative and
// does not include absPath
String fileName;
String dirName; // absolute path to directory
File dirFile; // dir File instance
File destFile; // a destination file
boolean aboolean, successful;
int i;
long timeInMilliseconds; // relative time.
long bytes; // size of the file
String path; // relative or absolute
String fileNames[];
FileNameFilter filter;

```

To make an instance of the File class:

```

file = new File(absPath);
file = new File(dirName,fileName);
file = new File(dirFile,fileName);
fileName = file.getName();

```

To get the path to the file:

```

path = file.getPath();

```

To get the absolute path to the file:

```

absPath = file.getAbsolutePath();

```

To get the name of the parent directory:

```

dirName = file.getParent();

```

To see if a file exists:

```

aboolean = file.exists();

```

To see if a file is writable:

```

aboolean = file.canWrite();

```

To see if a file is readable:

```

aboolean = file.canRead();

```

To see if a File instance is a file or a directory:

```

aboolean = file.isFile();
aboolean = file.isDirectory();

```

To see if getPath is relative:

```

aboolean = file.isAbsolute();

```

To get the relative time since modification:

```

timeInMilliseconds = file.lastModified();

```

To get the size of the file in bytes:

```

bytes = file.length();

```

To invoke mkdir and return true if successful:

```

    successful = file.mkdir();
To rename to a destination file:
    successful = file.renameTo(destFile);
To make all directories in this path:
    successful = file.mkdirs();
To list the files in a directory:
    fileNames = file.list();
To use a filter to list the files in a directory:
    fileNames = file.list(filter);
To delete a file:
    success = file.delete();
To compute a hashCode:
    i = file.hashCode();
To see if two files are equal:
    aboolean = file.equals(destFile);
To get the path string:
    path = file.toString();

```

(B-heading) Ls.getDirName

In Unix type operating systems there is a command called *ls*. We have modeled the *ls* command in the *futils* package using a class called *Ls*. The *Ls* class has static methods in it and is declared as *public* and *final*. *Ls.getDirName* will open a standard file dialog box and ask the user to select a file (there is no way to select a directory, as far as we know). Once the file is selected, a directory name is returned (it is the directory that contains the file)

```

    static public String getDirName() {
        FileDialog fileDialog =
            new FileDialog(new Frame(), "select
file");
        fileDialog.show();
        String dirName = fileDialog.getDirectory();
        dialog.dispose();
        return dirName;
    }

```

(B-heading) Ls.deleteFile

The *Ls.deleteFile* method takes an absolute path name (a string) converts it to a *File* instance and then deletes the file, with error reporting to the console.

```

    static public void deleteFile(String absPath) {
        File fileToDelete = new File(absPath);
        System.out.print("deleting file " + absPath);
        if (fileToDelete.exists()) {
            System.out.println(" deleted!");
            fileToDelete.delete();
        }
        else
            System.out.println(" does not exist");
    }

```

(B-heading) Futil.getReadFile Futil.getWriteFile and Futil.getDirFile

The static method, `Futil.getReadFile()` opens a dialog box for the user and returns a `File` instance, whereas `Futil.getDirFile` works by getting a file based on the `Ls.getDirName` method:

```
public static File getReadFile() {
    return new File(getReadFileName());
}
public static File getWriteFile() {
    return File(getWriteFileName(fs));
}
public static File getDirFile() {
    return new File(Ls.getDirName());
}
```

(A-heading) The FilenameFilter interface

The `FilenameFilter` is an interface reference data type that requires that an `accept` method be implemented. Once the `FilenameFilter` instance is created, it may be passed to utilities that will determine if a file name should be allowed on a list.

(B-heading) Class Summary

```
public interface FilenameFilter
    boolean accept(File dir, String name);
}
```

(B-heading) Class Usage

The best way to illustrate the usage of the `FilenameFilter` is to show some examples. All the examples are taken from the `futils` package.

Typically, a more efficient approach is to keep all file names in `File` instances. It is the case, however, that there are reasons for maintaining lists of files as arrays of `String` instances. This is particularly efficient when attempting to trade space for time. For example, the storage of a `String` instance is based on a dictionary that assumes the `String` is immutable. This permits the reuse of substrings. Therefore redundancy in absolute path name storage should not be too memory inefficient, as the `String` storage facility has been highly optimized. By emphasizing the storage of absolute path names in `String` instances, we have gained a space efficiency. However, having to create and dispose of many `File` instance creates a CPU inefficiency. Questions concerning the quantitative aspects of this trade-off remain open.

(B-heading) DirFilter

The `DirFilter` takes a file name and returns true if the file is a directory. It does this by first making an instance of the `File` class, using the file name, then targeting the instance with an `isDirectory()` invocation.

```
package futils;
import java.io.*;
import java.util.*;
public class DirFilter implements FilenameFilter {
    public boolean accept(File dir, String name) {
        return new File(dir, name).isDirectory();
    }
}
```

(B-heading) The FileFilter Class

The FileFilter class, just like the DirFilter Class, is used to determine if the dir+name constitutes a valid file by creating a temporary File instance.

```
package futils;
import java.io.*;
import java.util.*;
public class FileFilter implements FilenameFilter {
    public boolean accept(File dir, String name) {
        return new File(dir, name).isFile();
    }
}
```

(B-heading) The WildFilter Class

The WildFilter class contains the suffix string of the file name string that the programmer is looking for. Once the WildFilter is instanced, the suffix is unchangable (due to the private visibility of the suffix class variable).

```
package futils;
import java.io.*;
import java.util.*;
public class WildFilter implements FilenameFilter {
    private String suffix;
    WildFilter (String suffix_) {
        suffix = suffix_;
    }

    public boolean accept(File dir, String name) {

        return name.endsWith(suffix);
    }
}
```

(B-heading) Ls.getWildNames

The Ls.getWildNames() static method brings up a standard file open dialog box. After a user selection, all the names are returned in a String array, with the absolute path name.

```
static public String[] getWildNames(String wild) {
    File dir = Futil.getDirFile();
    String absPath = dir.getAbsolutePath();
    String[] fileNames = dir.list(new WildFilter(wild));
    System.out.println("getWildNames:"+absPath);
    for (int i=0; i < fileNames.length; i++) {
        fileNames[i] = absPath+fileNames[i] ;
    }
    return fileNames;
}
```

(B-heading) Ls.wildToConsole

The Ls.wildToConsole provides the service of listing all the files to the standard output device. This is equivalent to the Unix 'ls *.wild', or the DOS 'dir *.wild'.

```
static public void wildToConsole(String wild) {
    String[] files = getWildNames(wild);
    System.out.println(files.length + " file(s):");
    for (int i=0; i < files.length; i++)
```

```
        System.out.println("\t" + files[i]);
    }
}
```

(B-heading) Ls.deleteWildFile

Now for the really dangerous stuff. A method that deletes all the files in a directory, without confirmation!

(BEGIN WARNING) Use this method with caution.

(END WARNING)

```
static public void deleteWildFiles(String wild) {
    String[] files = getWildNames(wild);
    System.out.println(files.length + " file(s):");
    for (int i=0; i < files.length; i++)
        deleteFile(files[i]);
}
```

(B-heading) Ls.WordPrintMerge

The Ls.WordPrintMerge gets all the files that end with a "PICT" suffix and creates Microsoft Word print merge commands that will include the pict files into a single word document. In fact, this is the utility that helped create the figure manuscript for chapter 1 of this book.

```
static public void WordPrintMerge() {
    String wild = "PICT";
    String[] files = getWildNames(wild);
    System.out.println(files.length + " file(s):");
    int fileNumber;
    for (int i=0; i < files.length; i++) {
        fileNumber = i + 1;
        System.out.print("Figure *." +
            fileNumber +
            ".    «INCLUDE hd:current:Java
book:chapter I:batch 1 rev1:picts:");
        System.out.println(files[i]+"»");
    }
}
```

(B-Heading) Ls.lowerFileNames

(BEGIN WARNING)

Now for the really dangerous (and fun!) stuff. The Ls.lowerFileNames takes a File instance and *recursively traverses the file system converting ALL file names to lower case!*(END WARNING). There is no single command in most computer operating systems for doing this (for obvious reasons). The purpose of Ls.lowerFileNames is to maintain web sites that have file names that do not match the case of the file names in the hypertext references. This is particularly painful for web masters who port web sites from one file system to another. For example, on a DOS system, the file names may appear as upper case. On a Mac or Unix file system, files are mixed case.

```
public void lowerFileNames( File thePath ){
    String[] fileNames = thePath.list();
    String pathstr = thePath.getPath();
    for( int i=0;
        fileNames != null && i < fileNames.length; i++ ) {
```

```

String aFileName = fileNames[ i ];
String newFileName = aFileName.toLowerCase();
File theFile = new File( pathstr, aFileName );
if( theFile.isFile() ){
    //rename theFile to lowewr cases
    System.out.print( i+":" + aFileName );
    theFile.renameTo( new File( pathstr, newFileName ) );
    System.out.println( "\t==>\t"+newFileName );
}else{
    //case theFile is Dir, in the Dir,
    // repeat same procedure
    System.out.println( "Dir:"+aFileName );
    lowerFileNames( new File( pathstr+aFileName ) );
}
}
return;
} //lowerFileNames

```

(A-heading) The FileOutputStream Class

The FileOutputStream class resides in the java.io package. Instances of the FileOutputStream class are used as targets of the close method. When an instance of a FileOutputStream is made, the file is opened for write.

Since the creation and closing of a FileOutputStream instance can throw an IOException, operations involving the use of a FileOutputStream are surrounded by a try and catch.

(B-heading) Class Summary

```

public class FileOutputStream extends OutputStream {
    public FileOutputStream(String name) throws IOException
    public FileOutputStream(File file) throws IOException
    public FileOutputStream(FileDescriptor fdObj)
    public native void write(int b) throws IOException;
    public void write(byte b[]) throws IOException
    public void write(byte b[], int off, int len) throws
        IOException
    public native void close() throws IOException;
    public final FileDescriptor getFD() throws IOException
}

```

(B-heading) Class Usage

To open a file, using a standard file save dialog box, define a FileOutputStream instance by using the Futil.getWriteFileName()

```

FileOutputStream output_stream = new
    FileOutputStream(getWriteFileName());

```

Suppose the following variables are predefined:

```

String fileName;
File file;
FileDescriptor fd;
byte b, bytes[];
int offset, length;

```

Then to open a file for write (remember to use try and catch):

```

    fos = new FileOutputStream(name);
To open a file using a File instance:
    fos = new FileOutputStream(file);
To open a file using a FileDescriptor instance:
    fos = new FileOutputStream(fd);
To write a byte of data:
    fos.write(b);
To write an array of bytes:
    fos.write(bytes);
To write a subarray of bytes:
    fos.write(bytes, offset, length);
To close the FileOutputStream:
    fos.close();
To get the file descriptor:
    fd = fos.getFD();

```

(B-heading) Futil.getFileOutputStream

A simple way to get a file output stream, by prompting the user with a file dialog box and intercepting the possible resulting exceptions, is given below. The Futil.getFileOutputStream method will return null if the output file is unable to be made.

```

public static FileOutputStream getFileOutputStream() {
    FileOutputStream fos = null;
    try {fos =
        new FileOutputStream(getWriteFileName());
    }
    catch (IOException e) {
        System.out.println("futil:Could not create
file");
    }
    return fos;
}

```

(B-heading) Futil.closeOutputStream

As with the getFileOutputStream, there is a closeOutputStream method. (BEGIN NOTE) Any subclass of the OutputStream (i.e., the FileOutputStream) will be automatically cast into an OutputStream instance during the call. Thus, there is no need for a Futil.closeFileOutputStream method (END NOTE)

```

public static void closeOutputStream(OutputStream os) {
    try {os.close();} // end try
    catch (IOException exe)
    {System.out.println(
    "futil: could not close outputstream");}
}

```

(A-heading) The PrintStream Class

The PrintStream class resides in the java.io package. An instance of the PrintStream class exists in System.out and has been introduced, informally using:

```

System.out.println("Hello World");

```

A `PrintStream` instance may be explicitly flushed. When flushed, the buffered output will be written. A newline can be used to trigger a flush.

(B-heading) Class Summary

```
public class PrintStream extends FilterOutputStream {
    public PrintStream(OutputStream out)
    public PrintStream(OutputStream out, boolean autoflush)
    public void write(int b)
    public void write(byte b[], int off, int len)
    public void flush()
    public void close()
    public boolean checkError()
    public void print(Object obj)
    synchronized public void print(String s)
    synchronized public void print(char s[])
    public void print(char c)
    public void print(int i)
    public void print(long l)
    public void print(float f)
    public void print(double d)
    public void print(boolean b)
    public void println()
    synchronized public void println(Object obj)
    synchronized public void println(String s)
    synchronized public void println(char s[])
    synchronized public void println(char c)
    synchronized public void println(int i)
    synchronized public void println(long l)
    synchronized public void println(float f)
    synchronized public void println(double d)
    synchronized public void println(boolean b)
}
```

(B-heading) Class Usage

Suppose the following variables are predefined:

```
OutputStream os;
PrintStream ps=new PrintStream(os);
boolean autoflush;
byte b, bytes[];
int offset, length;
boolean aboolean;
Object object;
String string;
char aChar, charArray[];
int i;
long l;
float f;
double d;
```

To make a `PrintStream` instance from any `OutputStream` instance:

```
ps = new PrintStream(os);
```


To specify the automatic flush:

```
ps = new PrintStream(os, autoflush);
```

To write a byte:

```
ps.write(b);
```

To write a subarray:

```
ps.write(bytes, offset, length);
```

To flush the stream:

```
ps.flush();
```

To close the stream:

```
ps.close();
```

To flush the print stream with output stream error messages and a return true if an error ever occurred during the life of the PrintStream:

```
aboolen = ps.checkError();
```

To print a newline:

```
ps.println();
```

To print an object or an object+newline:

```
ps.print(object);
```

```
ps.println(object);
```

To print a String or a string+newline:

```
ps.print(string);
```

```
ps.println(string);
```

To print a char or an array of char:

```
ps.print(aChar);
```

```
ps.print(charArray);
```

To print an int, long, float or double:

```
ps.print(i);
```

```
ps.print(l);
```

```
ps.print(f);
```

```
ps.print(d);
```

To print an int, long, float or double + newline:

```
ps.println(i);
```

```
ps.println(l);
```

```
ps.println(f);
```

```
ps.println(d);
```

To print a boolean or boolean + newline:

```
ps.print(aboolen);
```

```
ps.println(aboolen);
```

(B-heading) Futil.makeToHtml

In this section we show a method for reading all the documents in a directory and creating a table of contents, in HTML. The user is prompted for the input directory and for the location of the output file. It is beyond the scope of this book to explain HTML.

The Futil.makeToHTML uses Futil.getDirFile to create a standard file open dialog that prompts the user for an HTML file.

```
1. public static void makeToHtml() {
2.     File dir = getDirFile();
3.     String[] files = dir.list(new FileFilter());
4.     System.out.println(files.length + " file(s):");
```

In line 5, the `Futil.getFileOutputStream` is used (without exception handling, since this has been handled at a lower level).

```

5.   FileOutputStream fos = getFileOutputStream();
6.   PrintStream ps = new PrintStream(fos);
7.   ps.println("<HTML>");
8.   ps.println("<BODY>");
9.   ps.println("<ul>");

```

On lines 10 and 11, the file names are written as a part of the HTML. Note that file instances were never needed, as this is a simple text output program. The *href* is a hypertext reference that shows the file with a relative path name.

```

10.  for (int i=0; i < files.length; i++)
11.      ps.println("<LI><a href = \"\" + files[i]+\"\">"+
12.          files[i]+"</a><P>");
13.  ps.println("</ul>");
14.  ps.println("</BODY>");
15.  ps.println("</HTML>");
16.  closeOutputStream(fos);
17.  }

```

The browser output of `makeTocHtml` (after being run on the `futils` package) appears below:

```

Cat.java
DirFilter.java
FileFilter.java
Find.java
Futil.java
Ls.java
WildFilter.java

```

When any of the underlined links are clicked on, a plain-text file of the source code is shown.

(A-heading) The `FileInputStream` Class

The `FileInputStream` class resides in the `java.io` package. It may be constructed from a file name or `File` instance. The `FileInputStream` is a subclass of the `InputStream` class.

The `FileInputStream` keeps a private copy of the `FileDescriptor` reference.

The read methods always block if there are no more bytes to read. When a read method blocks, it will stop execution of the thread. This indicates that I/O bound tasks should probably be placed into their own threads. If, on the other hand, a file will open quickly, then placing the file I/O in its own thread may needlessly complicate the program. Thus, the decision to use threaded I/O depends upon the application.

(B-heading) Class Summary

```

public class FileInputStream extends InputStream {
    public FileInputStream(String name) throws
        FileNotFoundException
    public FileInputStream(File file) throws
        FileNotFoundException
    public FileInputStream(FileDescriptor fdObj)
    public int read() throws IOException;
}

```

```

    public int read(byte b[]) throws IOException
    public int read(byte b[], int off, int len) throws
        IOException
    public long skip(long n) throws IOException
    public int available() throws IOException
    public void close() throws IOException
    public final FileDescriptor getFD() throws IOException
}

```

(B-heading) Class Usage

Suppose the following variables are predefined:

```

String fileName;
FileInputStream fis;
File file;
FileDescriptor fd;
byte b bytes[];
int length, offset;
int amountRead;
int n, numberSkipped;

```

To make an instance of the `FileInputStream` class from a file name, `File` instance or `FileDescriptor` instance:

```

fis = new FileInputStream(fileName);
fis = new FileInputStream(file);
fis = new FileInputStream(fd);

```

To read a byte of data (-1 returned at end of stream):

```
b = fis.read();
```

To read an array of bytes (-1 returned at end of stream):

```
amountRead = fis.read(bytes);
```

To read a subarray of bytes:

```
amountRead = fis.read(bytes, offset, length);
```

To skip `n` bytes:

```
numberSkipped = fis.skip(n);
```

To find out how many bytes can be read (used to test the length of the file):

```
n = fis.available();
```

To close the stream

```
fis.close();
```

To get the file descriptor:

```
fd = fis.getFD();
```

(B-heading) Futil.getFileInputStream

In order to localize the exception handling within the `Futil` class, a static public method has been devised called `Futil.getFileInputStream`. As seen below, `getFileInputStream` is overloaded to either take an absolute path name, or no argument at all. When no argument is passed, a standard file dialog box is presented to the user for file selection. After a valid file is selected the `getFileInputStream` calls the overloaded version of itself that takes a string. Should the user cancel out of the dialog box selection, an exception causes an error message to be printed and program execution continues.

```

public static FileInputStream getFileInputStream() {
    return getFileInputStream(getReadFileName());
}

```

```

public static FileInputStream getFileInputStream(String
name) {
    FileInputStream fis = null;
    try
        {fis = new FileInputStream(name);}
    catch (IOException e)
        {System.out.println("futil:Could not open file");}
    return fis;
}

```

Recall that `FileInputStream` also has a constructor that works with a `File` instance:

```

public static FileInputStream getFileInputStream(File file)
{
    FileInputStream fis = null;
    try
        {fis = new FileInputStream(file);}
    catch (IOException e)
        {System.out.println("futil:Could not open file");}
    return fis;
}

```

(B-heading) Futil.available

The `Futil.available` method is a static public method that permits a fast file check to see how many bytes are in a file. This is a very specific operation that was developed for the recursive `futils.DirList` class (which appears in the next section).

```

// Open the file, return -1 if file cannot be opened
// otherwise return the size in bytes
public static int available(File file) {
    FileInputStream fis = null;
    int sizeInBytes = -1;
    try {
        fis = new FileInputStream(file);
        sizeInBytes = fis.available();
        fis.close();
    }
    catch (IOException e)
        {System.out.println("Futil:Could not open file");}
    return sizeInBytes;
}

```

(B-heading) The `futils.DirList` class—Recursive File Lister (`ls -al */* >foo`)

The `DirList` class resides in the `futils` package. If `DirList.main` is invoked, a standard open dialog box is presented to the user. The user selects a file and `DirList` builds a list of all files in the directory and all subdirectories. Invocation of `DirList.main` is like the Unix command `'ls -al */* >foo'` in that `DirList` opens each file, prints the size, in bytes, and stores the `File` instances, the total number of files visited and total number of bytes. A class instance variable called *history* is of `Vector` type and holds instances of all files visited.

(BEGIN WARNING) The power of recursive file management is fraught with danger. Do not attempt to write recursive file deletes as this is a serious time bomb. (END WARNING)

```

package futils;
import java.awt.*;
import java.io.*;
import java.util.*;
import java.lang.*;
public class DirList {
    String startDir;
    public Vector history = new Vector();
    int totalBytes = 0;
    int totalFiles = 0;
    public static void main(String args[]) {
        DirList dl = new DirList();
        dlprintStats();
    }
    DirList() {
        startDir = Ls.getDirName();
        startAtThisDir(startDir);
    }

    public void printStats() {
        System.out.println("Saw " +
            totalFiles +
            " files with a total size of " +
            totalBytes +
            " bytes");
    }
    //-----
-
    // Recursive function that given an anchor directory
    // will walk directory tree
    //
    //-----
-
    public void startAtThisDir(String anchorDir)
    {
        FileFilter files = new FileFilter();
        DirFilter dirs = new DirFilter();
        File f1 = new File(anchorDir);
        File f2;
        FileInputStream fis;
        int i;
        String[] ls;
        int bytes;
        System.out.println("Selected -> " + anchorDir);
        System.out.println("Files in this Directory: ");
    }

```

```

-----
//-----
-----
// Loop through all the filenames in the current
directory
// and store them in history:
//-----
-----
for (ls = f1.list(files), i=0;
     ls != null && i < ls.length; i++) {
    totalFiles++;
    f2 = new File(f1, ls[i]);
    bytes = Futil.available(f2);
    totalBytes += bytes;
    System.out.println(f2 +
        " has "+
        bytes + " bytes");
    history.addElement(f2);
}
//-----
-----
// This loop recurses on all directory names in
// the current working directory.
//-----
-----
for (ls = f1.list(dirs),
     i=0; ls != null && i < ls.length; i++)
    startAtThisDir(anchorDir + ls[i] + f1.separator);
}
}

```

(BEGIN NOTE) The use of the `Ls.getDirName` and `Futil.available` methods enabled the elimination of the try-catch structure that generally characterizes Java I/O code. As a result, the code for the `DirList` class has fewer nested code blocks. Since the exception handling is occurring at a lower level, some of the complexity is hidden from the programmer and the code appears easier to read. (END NOTE)

(A-heading) The `DataInputStream` Class

The `DataInputStream` class resides in the `java.io` package. The `DataInputStream` class is a byte stream reader. The `DataInputStream` provides high-level methods that supply reading and casting services from a stream of bytes into various primitive data types. This is useful when attempting to decode binary files (like the audio files in the following chapter).

The `DataInputStream` is a subclass of the `FilterInputStream` and implements the `DataInput` interface. When a read is performed and no bytes are available, the thread will block (cease to execute).

(B-heading) Class Summary

```

public class DataInputStream extends FilterInputStream
implements DataInput {
    public DataInputStream(InputStream in)

```

```

public final int read(byte b[]) throws IOException
public final int read(byte b[], int off, int len) throws
IOException
public final void readFully(byte b[]) throws IOException
public final void readFully(byte b[], int off, int len)
throws IOException
public final int skipBytes(int n) throws IOException
public final boolean readBoolean() throws IOException
public final byte readByte() throws IOException
public final int readUnsignedByte() throws IOException
public final short readShort() throws IOException
public final int readUnsignedShort() throws IOException
public final char readChar() throws IOException
public final int readInt() throws IOException
public final long readLong() throws IOException
public final float readFloat() throws IOException
public final double readDouble() throws IOException
public final String readLine() throws IOException
public final String readUTF() throws IOException
public final static String readUTF(DataInput in) throws
IOException
}

```

(B-heading) Class Usage

```

InputStream is;
byte b, bytes[];
int length, offset;
char c;
int i;
long l;
float f;
double d;
String string;

```

Suppose that the following variables are predefined:

```

InputStream is;
DataInputStream dis;
int numberRead;

```

To create an instance of the `DataInputStream` class:

```
dis = new DataInputStream(is);
```

To read data into a byte array or subarray:

```
numberRead = dis.read(bytes);
numberRead = dis.read(bytes, offset, length);
```

To read `bytes.length` into bytes from `bytes[0]`:

```
dis.readFully(bytes);
```

To read bytes into a subarray:

```
dis.readFully(bytes, offset, length);
```

To skip bytes:

```
numberSkipped = dis.skip(numberToSkip);
```

To read a boolean (a single byte that is non-zero for true to be returned):

```

    aboolean = dis.readBoolean();
To read a byte:
    b = dis.readByte();
To read the byte into an int:
    i = dis.readUnsignedByte();
To read a 16 bit signed or unsigned short:
    s = dis.readShort();
    i = dis.readUnsignedShort();
To read a 16 bit char:
    c = dis.readChar();
To read a 32 bits into an int:
    i = dis.readInt();
To read 64 bits into a long:
    l = dis.readLong();
To read 32 bits into a float:
    f = dis.readFloat();
To read 64 bits into a double:
    d = dis.readDouble();
To read a line, stopping at the end of the stream, '\n', '\r' or '\r\n'
    string = dis.readLine();
To read a Unicode Transfer Format (UTF string):
    string = dis.readUTF();
To read UTF from an InputStream:
    string = DataInputStream.readUTF(is);

```

(B-heading) Cat.fileToStream and Cat.javasToFile

The `futils` package has a facility that works like the `cat` command of Unix. The basic idea is that we would like Java to perform the ‘`cat *.java >file`’ sequence. That is, list all the files in the present directory into a single file.

On line 2 we build a list of files with the ‘`.java`’ suffix. Lines 6 and 7 pass the file name and printstream of each file to the `Cat.fileToStream` method.

```

1.static public void javasToFile() {
2.String[] files = Ls.getWildNames("java");
3.    FileOutputStream fos =
4.    Futil.getFileOutputStream();
5.    PrintStream ps = new PrintStream(fos);
6.    for (int i=0; i < files.length; i++)
7.        fileToStream(files[i], ps);
8.    Futil.closeOutputStream(fos);
9. }
10.static public void fileToStream(String fileName,
PrintStream output) {
11.System.out.println("cat: "+fileName);
12.FileInputStream fis =
Futil.getFileInputStream(fileName);
13.    String line;

```

The mapping is that there may be many input files, but only a single output stream. This requires that we open and close the `FileInputStream` many times....on several different

files. (BEGIN NOTE) There is a try and catch surrounding lines 15-17. This is due to the IOException that line 16 might throw. (END NOTE)

```

14.     try {
15.         DataInputStream dis = new DataInputStream(fis);
16.         while ((line = dis.readLine()) != null)
17.             output.println(line);
18.     } // try
19.     catch (Exception exe)
20.         {System.out.println("cat:Error on input file");}
21.     Futil.closeInputStream(fis);
22. }
```

(A-heading) The DataOutputStream Class

Like the DataInputStream class, the DataOutputStream class resides in the java.io package. The DataOutputStream class is a byte stream writer. The DataOutputStream provides high-level methods that supply writing services from various primitive data types into a stream of bytes. This is useful when attempting to encode binary files (like the audio files in the following chapter).

The DataOutputStream is a subclass of the FilterOutputStream and implements the DataOutput interface.

A DataOutputStream instance keeps track of the number of bytes that it has written. This is kept in protected storage. All output performed with blocking is blocked until the data is finally written. If the output does not say "with blocking" then the output may be buffered and flushed either manually or automatically.

A UTF-8 format is used to write strings in a machine independent manner. Chars from 1-127 are written as a single byte. Chars from 128-2047 and 0 are written by a pair of bytes. Chars from 2048-65535 are written by 3 bytes. The actual bit format is given in [Gosling and Yellin].

(B-heading) Class Summary

```

public class DataOutputStream extends FilterOutputStream
implements DataOutput
public DataOutputStream(OutputStream out)
public synchronized void write(int b) throws IOException
public synchronized void write(byte b[], int off, int len)
public void flush() throws IOException
public final void writeBoolean(boolean v) throws
IOException
public final void writeByte(int v) throws IOException
public final void writeShort(int v) throws IOException
public final void writeChar(int v) throws IOException
public final void writeInt(int v) throws IOException
public final void writeLong(long v) throws IOException
public final void writeFloat(float v) throws IOException
public final void writeDouble(double v) throws IOException
public final void writeBytes(String s) throws IOException
public final void writeChars(String s) throws IOException
public final void writeUTF(String str) throws IOException
public final int size()
```

}

(B-heading) Class Usage

Suppose that the following variables are predefined:

```
char c;
String string;
OutputStream os;
short s;
int i, offset, length;
long l;
float f;
double d;
byte b;
byte bytes[];
```

To construct a new `DataOutputStream` instance:

```
DataOutputStream dos = new DataOutputStream(is);
```

To write a byte, with blocking:

```
dos.write(b);
```

To write a sub-array of bytes:

```
dos.write(bytes, offset, length);
```

To flush the output stream:

```
dos.flush();
```

To write a boolean (as a 0 or a 1 byte):

```
dos.writeBoolean(aboolean);
```

To write a byte to an underlying 8 bit representation:

```
dos.writeByte(b);
```

To write a short to an underlying 16 bit representation with the high-byte first:

```
dos.writeShort(s);
```

To write a char to an underlying 16 bit representation with the high-byte first:

```
dos.writeChar(c);
```

To write an int to an underlying 32 bit representation with the high-byte first:

```
dos.writeInt(i);
```

To write a long to an underlying 64 bit representation with the high-byte first:

```
dos.writeLong(l);
```

To write a float to an underlying 32 bit representation with the high-byte first:

```
dos.writeFloat(f);
```

To write a double to an underlying 64 bit representation with the high-byte first:

```
dos.writeDouble(d);
```

To write a `String` as a sequence of bytes (this casts each char as a byte, then writes the byte). This will write `s.len` bytes:

```
dos.writeBytes(s);
```

To write a `String` as a sequence of 16 bit chars. This will output $2 * s.len$ bytes:

```
dos.writeChars(s);
```

To write a string in a machine independent UTF-8 format:

```
dos.writeUTF(s);
```

To get the number of bytes written:

```
i = dos.size();
```

(B-heading) Java, C, C++ -> HTML, The HTML Converter

Formats! You want formats? We got formats! Formats for browsers (HTML). Formats for Word processors (RTF). Formats for computer languages (Java, C, C++). Formats for images (JPEG, GIF, PPM). Formats, you can't live with 'em, you can't live without 'em! In this section we address the conversion of Java, C and C++ into formatted and colorized HTML (Hyper Text Markup Language).

(BEGIN ON CD) One of the packages included with the DiffCAD program is called the htmlconverter package. (END ON CD) The htmlconverter package contains the support systems for the HtmlGenerator class, a Java, C and C++ source code conversion program that generates formatted and colored HTML files. The following code was converted to HTML by the HtmlGenerator. The HTML was then converted into RTF (Rich Text Format). RTF is the format that Microsoft Word is able to read. The conversion from HTML to RTF was performed using a program called MacLinkPlus [DataViz].

```
1. package htmlconverter;
2. import java.io.*;
```

The `JavaStream` extends the `DataInputStream` and, as such, is able to inherit the `DataInputStream` methods.

```
3. public class JavaStream extends DataInputStream
4.     implements JavaText, CText, CplusplusText {
5.     JavaHtmlString mainText = new JavaHtmlString();
6.     JavaHtmlString comments = new JavaHtmlString();
7.     JavaHtmlString strings = new JavaHtmlString();
8.     JavaHtmlString keywords = new JavaHtmlString();
9.     DataOutputStream dos;
10.    public static String[] reservedWords =
        javaReservedWords;
```

The `JavaStream` constructor takes an `InputStream`

```
11.    JavaStream(DataInputStream s0, DataOutputStream s1)
12.    {
13.        super(s0);
14.        dos = s1;
15.    }
16.    public void convertToHtml() {
17.        int index;
18.        boolean isCommentedOut = false;
19.        boolean isQuoted = false;
20.        boolean isSingleQuoted = false;
```

The basic idea behind the `convertToHtml` method is that a string of characters should be read using the `readLn` method. Since `JavaStream` is a subclass of the `DataInputStream`, `JavaStream` inherits the `readLn` method.

```
1. String line;
2. String str1, str2;

3. try {
4.     while (true) {
```

Line 5 reads until the end of a line and places the characters into the *line* String.

```
5.         line = readLine(); // get a line
```

```

6.             if (line == null) {           // if line is
null
7.                 break;                   // break
this loop
8.             }

```

After a line of text is read, a series of search and replaces are performed. For example:

```

1.  for (int counter = 0; counter < line.length();
counter++) {

```

Recall, from Chapter 2, that to find anString starting from anInt we use:

```
anInt = anotherString.indexOf(aString, anInt);
```

So in line 2, we start from the counter position and look for the less than character.

```

2.  index = line.indexOf("<", counter);
3.  if (index < 0) break;
4.  // if < is not found then break this loop

```

We then insert the string at the index point, replacing the '<' with '<'. This replacement is performed because HTML represents the '<' character as '<' and hold '<' as reserved.

```

5.  line = line.substring(0, index) + "&lt;";
6.      + line.substring(index + 1, line.length());
7.  // replace < to &lt;;
8.  counter = index + 4;
9.  // update counter
10. }

```

This search and replace is continued until all the replacements are performed. At that point the line is written to the DataOutputStream instance using a write bytes.

(A-heading) StreamTokenizer

The StreamTokenizer class resides in the java.io package. It is used to convert an InputStream instance into a stream of tokens. We refer the reader to [Gosling and Yellin] for a more complete description of the StreamTokenizer. Bytes are read from the InputStream instance and are treated as unsigned shorts that range from 0-255.

A StreamTokenizer instance is an instance of a parser. The parser settings may be altered to recognize C-style comments, C++ style comment, line terminators, and to convert tokens to lower case.

The StreamTokenizer is a higher-level Stream than the DataInputStream.

(B-heading) Class Summary

```

public class StreamTokenizer {
    public int ttype
    public static final int TT_EOF
    public static final int TT_EOL
    public static final int TT_NUMBER
    public static final int TT_WORD
    public String sval
    public double nval
    public StreamTokenizer (InputStream I)
    public void resetSyntax()
    public void wordChars(int low, int hi)
    public void whitespaceChars(int low, int hi)

```

```

    public void ordinaryChars(int low, int hi)
    public void ordinaryChar(int ch)
    public void commentChar(int ch)
    public void quoteChar(int ch)
    public void parseNumbers()
    public void StreamTokenizer:eolIsSignificant(boolean
    flag)
    public void StreamTokenizer:slashStarComments(boolean
    flag)
    public void slashSlashComments(boolean flag)
    public void lowerCaseMode(boolean fl)
    public int nextToken() throws IOException
    public void pushBack()
    public int lineno()
    public String toString()
}

```

(B-heading) Class Usage

Suppose the following variables are predefined:

```

String sval;
double nval;
InputStream is;
int low, hi, ch, token, lineNumber;
String s;
StreamTokenizer st;

```

To make an instance of a StreamTokenizer:

```
st = new StreamTokenizer(is);
```

The StreamTokenizer has several constants held as public static final ints. They are used by case statements to determine the token type. They are the end-of-file and end-of-line token:

```
StreamTokenizer.TT_EOF
StreamTokenizer.TT_EOL

```

To determine if the token is a number (value in nval):

```
StreamTokenizer.TT_NUMBER
```

To determine if the token is a word (value in sval):

```
StreamTokenizer.TT_WORD
```

To make all characters ordinary:

```
st.resetSyntax();
```

To cumulatively specify the Unicode characters to be used for words:

```
st.wordChars(low, hi);
```

To spec the white space chars:

```
st.whitespaceChars(low, hi);
```

Ordinary chars are returned by nextToken. To spec the ordinary chars:

```
st.ordinaryChars(low, hi);
```

To add an ordinaryChar:

```
st.ordinaryChar(ch);
```

To add a single line comment char (all chars to end of line are comment chars and tokenizer skips them:

```
st.commentChar(ch);
```

To spec the quote char to delimit a string:

```
st.quoteChar(ch);
```

To spec that numbers should be parsed:

```
st.parseNumbers();
```

To make TT_EOL be returned by nexttoken:

```
flag = true;
```

```
st.eolIsSignificant(flag);
```

To select '/*' comments:

```
st.slashStarComments(flag);
```

To select '/' comments:

```
st.slashSlashComments(flag);
```

To select if TT_WORD are converted to lower case:

```
lowerCaseMode(fl);
```

To parse a token, returning ttype:

```
ttype = st.netToken();
```

To push a token back into the stream:

```
st.pushBack();
```

To get the current line number:

```
i = st.lineno();
```

To convert the StreamToken to a string:

```
s = st.toString();
```

(B-heading) Futil. readDataFile

In order to save the state of our DiffCAD program, we write several key parameters into a file. These parameters are stored with keywords and values so that a human with a text editor (perhaps a non-programmer) is able to alter the data. A sample data file follows:

```
Data format is order dependent
lyon.Laser      Rotation=
34.4198         6.47311    -36.7812
lyon.Camera     rho=
18.4931 pc=     125.3      -102.35
A= 4.8 F=
3.6  lyon.Wedge p1 = -403.685
591.244  lyon.Grating P1= 6651 P2= 2261 L= 81
```

The readDataFile method is in the futils package. It is used to take a file name and read data into an array of doubles. To perform this task we use a StreamTokenizer instance called tokens.

```
1. public static void readDataFile(String file, double
data[]) {
2.     System.out.println("processing:\t" + file);
3.     FileInputStream inputFile =
4.         getFileInputStream(file);
5.     StreamTokenizer tokens = new
StreamTokenizer(inputFile);
6.     int next = 0;
7.     int num = 0;
```

Note that the tokens.nextToken() must be nested in a try-catch block.

```
8.     try {
9.         while ((next = tokens.nextToken()) !=
tokens.TT_EOF) {
10.            switch (next) {
11.                case tokens.TT_WORD:
```

12. `break;`
 Lines 11 and 12 indicate that words are ignored (they are for humans!!). Only numbers matter, and since people are warned not to change the order of the parameters, future versions of the data file may only add numbers to the end of the file. They may not insert numbers!

```

13.             case tokens.TT_NUMBER:
14.                 data[num] = (double)
tokens.nval;
15.                 System.out.println(num+": "+
data[num]);
16.                 num = num + 1;
17.                 break;
18.             case tokens.TT_EOL:
19.                 break;
20.         }
21.     }
22. }
23.     catch (Exception exe)
24.
    {System.out.println("listFilteredHrefFile:er!");}
25.     closeInputStream(inputFile);
26. }
```

Creation of the file is a simple matter. In the following code, we open a file as a `PrintStream` instance, write out the data, then close the file:

```

public static void save() {
    FileOutputStream os =
Futil.getWriteFileOutputStream();
    PrintStream output = new PrintStream(os);
    output.println("Data format is order dependent");
    Geometry.print(output);
    Futil.closeOutputStream(os);
}
```

The `Geometry.print` is a public static method that uses the `Print` class in the `futils` package, which is described in the following section.

(B-heading) Futil.Print

To ease the implementation of output to files and the console, the `futils` package provides its own `Print` class. The `Print` class is a public final class with a private constructor, to prevent instantiation. The key point about the `Print` class is that it stores an instance of the `PrintOutputStream` in a settable variable. Thus, output may be directed to a file or, by default, to the console. The trick is that the change of the `PrintOutputStream` is centralized, and all methods that make use of the `Print` class benefit from this centralization. For example, in the `Geometry` class we write:

```

1. static public void print(PrintStream out) {
We setPrintStream in the Print class, execute our print, then reset the stream.
2. Print.setPrintStream( out);
3.     print();
4. Print.setPrintStream( System.out);
5. } // print
```

Every shape knows how to print (using the Print class) and so will make use of the new output stream when printing. If the print method of line 6 is invoked without line 2 being executed then the output will be directed to the console.

```

6. static public void print() {
7.     laser.print();
8.     camera.print();
9.     wedge.print();
10.    grating.print();
11.    System.out.println("Lambda = "+
lambda.getValue());
12.    System.out.println("s = " + s());
13. } // print

```

The code for the `futils.Print` follows:

```

1. package futils;
2. import java.util.*;
3. import java.io.*;
4. public final class Print {
5. // prevent instantiation
6. private Print() {}
7. private static PrintStream output = System.out;
8. public static void setPrintStream(PrintStream ps) {
9.     output = ps;
10. }
11. public static PrintStream getPrintStream() {
12.     return output;
13. }
14. public static void d(double i)
15.     {Print.output.print(i+"\t");}
16. public static void ln(double d)
17.     {Print.output.println(d);}
18. public static void ln(String str)
19.     {Print.output.println(str);}
20. public static void print(String str)
21.     {Print.output.print(str);}
22. public static void print(double d)
23.     {Print.output.print(d);}
24. public static void print(int d)
25.     {Print.output.print(d);}
26. public static void className(Object o)
27.     {Print.output.print(o.getClass().getName() +
"\t");}
28. }

```

(B-heading) Futil.writeFilteredHrefFile

A buggy web authoring tool (like Netscape 3.01 Gold), will produce HTML that has hrefs (hyper-text references) with embedded spaces. The API documentation shows an href on line 8. The space between “API” and “Documentation” is a bug. Browsers will stop reading the href name after the first space. To get browsers to recognize the space, we must replace spaces with their hexadecimal equivalent, %20.


```

1. <html>
2.   <head>
3.     <title>
4.     API User's Guide
5.   </title>
6. </head>
7. <body>
8.   <a href= "API Documentation/packages.html" > Java
API</a>

```

The corrected version of line 8 follows:

```
<a href= "API%20Documentation/packages.html" > Java API</a>
```

To perform this transform, we build a custom tokenizer that looks for quoted strings and replaces all spaces that they contain with a %20. This has the bug of transforming the spaces in non-href strings to %20 too (a bug we have been able to live with).

```

1. public static void writeFilteredHrefFile(String
inputName, String outputName) {
2. System.out.println("Filtering:\t" + inputName
+"\t>\t"+outputName);
3. try {

```

Lines 4-7 make an input stream and a StreamTokenizer instance.

```

4.   FileInputStream is = new FileInputStream(inputName);
5.   StreamTokenizer tokens = new StreamTokenizer(is);
6.   FileOutputStream os = new FileOutputStream(outputName);
7.   PrintStream output = new PrintStream(os);
8.   int i;
9.   int next = 0;

```

The interesting bit is line 10, recall that `resetSyntax` will make all characters ordinary. This that there are no comment characters.

```
10. tokens.resetSyntax();
```

Line 11 says that all characters are word characters. Line 12 identifies the only character of interest, the quote character. Recall that `nextToken` will read until the `quoteCharacter`, setting `sval` to the value of the body of the string contained in quotes.

```

11. tokens.wordChars(0,255);
12. tokens.quoteChar(' ');
13. while ((next = tokens.nextToken()) != tokens.TT_EOF) {
14.     switch (next) {
15.         case ' ':
16.             output.print(' ');

```

We now output a quote, then we scan the string for a space. If a space is found, output a "%20" otherwise output the character.

```

17.         for (i=0;i<tokens.sval.length();i++)
18.             if (tokens.sval.charAt(i) == ' ')
19.                 output.print("%20");
20.             else
21.                 output.print(tokens.sval.charAt(i));
22.             output.print(' ');
23.             break;

```

In all other cases we simply output the string read. This is really nice because we only read up to the point at which a delimiting character occurs.

```

24.         case tokens.TT_WORD:
25.             output.print(tokens.sval+" ");
26.             break;
27.         case tokens.TT_NUMBER:
28.             output.print(tokens.nval+" ");
29.             break;
30.         case tokens.TT_EOL:
31.             output.println();
32.             break;
33.         } // end switch
34.     } // end while
35. is.close();
36. os.close();
37. } // end try
38. catch (Exception exe)
39.     {System.out.println("writeFilteredHrefFile:er!");}
40. }
```

(A heading) Exercises

The astute reader will note that we did not perform local handling of exceptions for the StreamTokenizer. We suggest the development of a StreamTokenizer with local handling of exceptions as an exercise for the reader.

Another topic for exploration includes the adding of features onto the `futils.DirList` class. One nice feature might be to include a constructor that takes a String instance for a suffix. `DirList` could then construct a `WildFilter` instance to form a list of only those files that have an ending that matches the suffix string.

A topic of exploration, that we have not seen in any book (yet) is the creation of a Unix like command line interface to Java. Then commands like “`ls *.java > foo`” could be typed into the command line reader. This could be an excellent way to employ piping and to write a series of data processing type tools.

(A-heading) Summary

In this chapter we covered the `Dialog` and `FileDialog` classes as the widgets to use to get file names from the user. The `Futils.getReadFileName` and `Futils.getWriteFileName` both had embedded calls to the `FileDialog` constructor. As a result, there is no reason to embed a file name into a file in the Java source code. The `futils` package contained a list of classes that are used on a daily basis and save several line of source code each time that they are used.

From a design viewpoint, the programmer might object to handling exceptions locally, as the `futils` classes attempt to do. This generally does not appear to be a problem. In short, we have found the local handling of exceptions to be the correct approach in all cases that we have encountered so far.

This chapter introduced a number of new classes that are unique to this book. The `Ls` class has the ability to list batches of files using the “`ls *.<suffix>`” and place the results into internal data structures. The `Ls` class provides the ability to recursively traverse the directory tree structure using a form that is like the UNIX “`ls -al */*`”. The `Cat` class takes

the output of *Ls* and lists the files into a single file. Batch processing was also used by the *HtmlGenerator* class to input C, C++ or Java and output colored and formatted HTML, with a hyper-linked HTML index.

(CN) 5 Digital Audio Processing

Thy voice sounds like a prophet's word;
And in its hollow tones are heard
- Fitz-Greene Halleck. 1790-1867.

(A-heading) What is Digital Signal Processing?

Sound is a pressure wave that traverses a medium. Sound pressure waves in air are the objective cause of human hearing. Sound will not travel through a vacuum, but it will travel through various phases of matter (solid, liquid and gas).

A transducer is a device that takes power from one system and supplies power to another. For example, a microphone is a transducer that takes sound power and supplies electrical power. The electrical power supplied by the microphone forms a signal that is analog. An analog signal is continuous.

Digitization is a process that converts a continuous signals into a digital form.

Digitization (also known as analog to digital conversion) is performed by sampling and quantization. Sampling is the process of converting a continuous signal into a set of voltages. Quantization is the process of converting the sampled voltages into a countable set of digital values. Analog data that is converted to digital data is said to be PCM encoded. PCM stands for Pulse Code Modulation and is a broad term that can refer to any type of digital encoding of analog data. Figure 5.1 depicts a PCM encoder.

Figure 5.1. Block diagram of a PCM encoder

A low-pass filter (called an anti-aliasing filter) is typically set to attenuate frequencies at or above one-half the analog to digital converters' sampling rate (this is known as the Nyquist frequency).

To transform the PCM signal back into the analog domain, we couple a digital-to-analog converter with another low-pass filter. A block diagram of the PCM decoder is shown in Figure 5.2.

Figure 5.2. Block diagram of a PCM decoder

Digital signal processing is a kind of data processing that operates on PCM data. Thus, broadly speaking, audio, image and image sequence processing are 1-D, 2-D and 3-D digital signal processing.

In common usage, the term digital signal processing refers to one dimensional signals, $V(t)$. In image processing we often speak about two dimensional signals, $I(x, y)$. This chapter deals only with one dimensional digital signal processing in Java.

(A-heading) Why do we need digital signal processing?

A digital signal stream may come from any energy (i.e., sound, measurement, temperature, speed, pressure, radiation, etc.). There also exist non-physical phenomena that can produce a digital stream of data (i.e., financial data, statistical data, network traffic, etc.).

In short, digital signal processing may be performed on any recordable event. Digital signal processing is just a kind of data processing.

In this chapter we treat only the restricted domain of audio digital signal processing in Java. There are several reason for this;

1. Java can already play audio files.
2. The techniques may be extended to other types of data.
3. We can hear the results.
4. It is fun!

(A-heading) What is the spectrum of a signal?

The harmonic content of a signal is called the *spectrum* of the signal. The spectrum of a signal consists of a series of sin and cosine waves. *Spectra* is the plural form of spectrum. A French mathematician, Jean Baptist Joseph de Fourier (1768-1830), showed that harmonic waves (i.e., sine and cosine waves) may be summed in a series to form any periodic waveform. The summation (called the superposition principle) fails to approximate a waveform when the equations governing the waveform are non-linear (i.e., shock waves, turbulence, chaos, etc.) [Halliday]. The series was first formulated by, and is used in, harmonic analysis (also called Fourier analysis). Harmonic analysis is the process that determines the harmonic components of a complex wave. The series may be written as

$$v(t) = a_0 + (a_1 \cos t + b_1 \sin t) + (a_2 \cos 2t + b_2 \sin 2t) + \dots \quad (5.1)$$

where $a_0, a_1, b_1, a_2, b_2, \dots$ are constants called Fourier coefficients.

For example, a sawtooth wave may be computed by letting k in

$$f(x) = \frac{2}{\pi} \sum_{n=1}^k \frac{(-1)^{(n-1)} \sin(n\pi x)}{n}$$

go to infinity. When $k=5$, the waveform of Figure 5.3 results:

Figure 5.3. Sawtooth waveform with $k=10$

When $k=100$, the waveform of Figure 5.4 is produced.

Figure 5.4. Sawtooth waveform with $k=100$

When the waveform to be approximated is not periodic, the summation is replaced by the Fourier transform:

$$V(f) = F[v(t)] = \int_{-\infty}^{\infty} v(t) e^{-2\pi jft} dt \quad (5.2)$$

$$v(t) = F^{-1}[V(f)] = \int_{-\infty}^{\infty} V(f) e^{2\pi jft} dt \quad (5.3)$$

Where $e^{i\theta}$ is given by Euler's identity:

$$e^{i\theta} = \cos \theta + i \sin \theta \quad (5.4)$$

Euler's identity can lead to several equivalent representations for the Fourier series. For example there is the Sine-Cosine Representation

$$x(t) = \sum_{n=0}^{\infty} a_n \cos(2\pi n f_0 t) + \sum_{n=1}^{\infty} b_n \sin(2\pi n f_0 t)$$

where $f_0 = \text{frequency}$ and $nf_0 = \text{nth harmonic of } f_0$. The constants, known as Fourier coefficients, are found by correlating the time dependent function, $x(t)$, with a Nth harmonic sine-cosine pair:

$$a_0 = \frac{1}{T} \int_0^T x(t) dt$$

$$a_n = \frac{2}{T} \int_0^T x(t) \cos(2\pi n f_0 t) dt .$$

$$b_n = \frac{2}{T} \int_0^T x(t) \sin(2\pi n f_0 t) dt$$

Another common representation of the Fourier series is the amplitude-phase representation. This is also a result of the Euler's identity:

$$x(t) = c_0 + \sum_{n=1}^{\infty} c_n \cos(2\pi f_0 t + \theta_n)$$

$$c_0 = \frac{1}{T} \int_0^T x(t) dt$$

$$c_n = \sqrt{a_n^2 + b_n^2}$$

$$\theta_n = -\tan^{-1}\left(\frac{b_n}{a_n}\right)$$

In general the usage of the representation of the Fourier transform is a matter of preference, as the various representations are equivalent. There are some interesting properties of the Fourier transform, and while it is beyond this scope to state them all (or to prove any of them), we do give some of them here.

A Fourier transform representation of an a periodic signal has discrete spectral components at f_0 and nf_0 , as shown by (5.1). An aperiodic signal as a continuous and infinite spectrum, as shown by (5.2) and (5.3). This means that time-limited signals (which are, by definition, aperiodic) have infinite bandwidth.

The effective bandwidth of a signal is the width of the spectra which contains the most power. The average power in a given interval of time is computed by $P = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} |x(t)|^2 dt$.

To compute the average power in a periodic signal, whose period is T , we use

$P = \frac{1}{T} \int_0^T |x(t)|^2 dt$. The PSD (Power Spectral Density) is the power at a specific

frequency, $S(f)$. Chapter 6 will discuss the computation of the PSD in more detail, but a sample PSD may be seen in Figure 5.6.

The Fourier transforms are important because they permit computation in either the time-domain or in the frequency domain. Some relations with Fourier transforms follow:

Superposition state that linear combinations in the time domain become linear combinations in the frequency domain:

$$a_1 V_1(f) + a_2 V_2(f) = F[a_1 v_1(t) + a_2 v_1(t)] \quad (5.5)$$

Delay in the time domain causes a phase shift in the frequency domain:

$$V(f)e^{-2\pi if} = F(v(t - t_d)) \quad (5.6)$$

Scale change in the time domain causes a reciprocal scale change in the frequency domain:

$$\frac{1}{|\alpha|} V\left(\frac{f}{\alpha}\right) = F(v(\alpha t)), \alpha \neq 0 \quad (5.7)$$

The convolution theorem states that multiplication in the time domain causes convolution in the frequency domain:

$$V * W(f) = F(v(t)w(t)) \quad (5.8)$$

Where convolution between two functions of the same variable is defined by:

$$V * W(f) \equiv \int_{-\infty}^{\infty} V(\lambda)W(f - \lambda)d\lambda \quad (5.9)$$

For proofs of these results, see [Carlson].

(A-Heading) What does sampling do to the spectrum of a signal?

The digitization process, as depicted in Figure 5.1, starts with the process of sampling. The sampling-reconstruction process does not have to perform digitization, and yet is still responsible for the introduction of additional harmonic content into the reconstructed signal. The sampling-reconstruction process is shown in Figure 5.5.

Figure 5.5. The Sampling-Reconstruction process

Figure 5.5 shows a continuous signal, $v(t)$ being sampled at a rate of f_s using an electronically controlled switch. The output of the switch is fed into an amplifier that has a gain of R . Mathematically, the sampling function may be expressed in terms of a Dirac delta function multiplied by $v(t)$ in the time domain. The Dirac delta function, $\delta(t)$, is a generalized function that is defined by:

$$\int_{-\varepsilon}^{\varepsilon} \delta(t)dt = 1 \quad (5.10)$$

Where ε is arbitrarily small. Thus the Dirac delta function is a unit impulse that occurs at time $t=0$. We can make the Dirac delta function fire at any time, t_d , by using the form

$$\delta(t - t_d). \quad (5.11)$$

Further, we can model the sampling function as an infinite sum of Dirac delta functions:

$$s(t) = \sum_{n=-\infty}^{\infty} \delta(t - n / f_s) \quad (5.12)$$

The switching function multiplies $s(t)$ by $v(t)$. The sampled function is represented by

$$v_s(t) = v(t)s(t) = v(t) \sum_{n=-\infty}^{\infty} \delta(t - n / f_s) \quad (5.13)$$

Recall that multiplication in the time domain is convolution in the frequency domain:

$$V_s(f) = F[v_s(t)] = V(f) * S(f) \quad (5.14)$$

The Fourier transform of an impulse train is also an impulse train

$$V_s(f) = V(F) * \sum_{n=-\infty}^{\infty} f_s \delta(f - nf_s) \quad (5.15)$$

Finally we see that sampling a signal at a rate of f_s causes the spectrum to be reproduced at f_s intervals.

$$V_s(f) = f_s \sum_{n=-\infty}^{\infty} V(f - nf_s) \quad (5.16)$$

A sample of this spectrum replication is shown in Figure 5.6.

Figure 5.6 Spectrum replication due to sampling
Equation (5.16) is called the aliasing formula.

(A-heading) Audio Files

In this section we describe how to read, play, graph and write audio files. The current Java API can open an audio file and play it. There is no support for decoding the audio file. Thus, the code that we present in this chapter extends and builds upon the existing Java API.

The reader will find a collection of sound utilities for the Mac at:

<http://wwwhost.ots.utexas.edu/mac/pub-mac-sound.html>. There are also a collection of sound utilities on the book CD-ROM. These are quite useful for recording and resampling audio files.

(A-heading) The sun.audio Package

AudioData, AudioDataStream, AudioPlayer, AudioStream, AudioStreamSequence and ContinuousAudioDataStream are public classes that reside in the sun.audio package.

(BEGIN WARNING) Code that makes use of classes in the Java sun.audio package does so in an undocumented and unsupported way. When the Java sun.audio package changes, code that depends on the classes in the sun.audio package may not work. (END WARNING).

(BEGIN NOTE) A trick of the Java trade is to take a library of files (like the classes.zip file that comes with some applet viewers) and unzip the file. Unzipping creates a large number of class files (446 for JDK 1.0.2) and it is a very good idea to perform this operation on a copy of the classes.zip (don't break your applet viewer!). Using a class browser (like the class dumper included with DiffCAD) you can list the fields of the class. This is called hacking.(END NOTE)

A clip of audio data, contains μ -law encoded 8-bit, 8000 sample rate data. This data can be used to construct an AudioDataStream instance. The getAudioData method gets an audio clip out of the cache. The AudioStream is an input stream used to play AudioData. For more documentation on the sun.audio package see <http://www.cdt.luth.se/java/doc/sun/shared/Package-sun.audio.html>.

(BEGIN NOTE)

Current implementations have been tested with the book software on Mac OS System 7.5x, Windows 95/NT and Sun Solaris operating systems. The sound player software cannot make audio output on a wintel platform that lacks a sound card. Further, some sound cards do not get driven properly by the current Java sound drivers. Sound blaster compatible cards seem to work with this books software.

(END NOTE)

We have *guessed* how classes in the sun.audio package might work. There is no documentation upon which to base a solid code foundation. However, the code that we present in the following section does work, at present.

(A-heading) The AudioStream Class

The AudioStream class resides in the sun.audio package. It is used to Convert an InputStream to an AudioStream. The AudioStream extends the FilterInputStream class. As a subclass of the FilterInputStream, the AudioStream class inherits all the methods and attributes of the FilterInputStream.

(B-heading) Class Summary

Since we cannot reverse engineer the software that makes up the AudioStream, we reproduce the raw class dumper output from the DiffCAD program

```
package sun.audio
import java.lang.Thread;
import java.io.IOException;
/*
 * This class has 1 optional class attributes.
 * These attributes are:
 * Attribute 1 is of type SourceFile
 * SourceFile : AudioStream.java
 */

public class AudioStream extends FilterInputStream {
public void AudioStream(java.io.InputStream a);
public int read(byte a[], int b, int c);
public sun.audio.AudioData getData();
public int getLength();
}
```

(B-heading) Class Usage

Suppose the following variables are pre-defined:

```
InputStream is;
int position, length;
byte bytes[];
AudioData ad;
```

To construct an Instance of the AudioStream class (throws IOException):

```
AudioStream as = new AudioStream(is);
```

To construct an instance of an AudioStream from a URL:

```
AudioStream as = new AudioStream(url.openStream());
```

To read an array of bytes from the input stream starting at position and proceeding for length bytes (throws IOException):

```
as.read(bytes, position, length);
```

To find the length of the AudioStream instance:

```
length = as.getLength();
```

To get the AudioData from an AudioStream instance:

```
ad = as.getData();
```


(A-heading) The AudioData Class

The AudioData class resides in the sun.audio package. An instance of the AudioData class is often called a *clip* of audio data. The AudioData instance consists of 8-bit μ -law encoded, 8000 Hz sample rate data. An instance of the AudioData stream is used to construct an instance of an AudioDataStream. An instance of an AudioDataStream can be played to the speaker of audio capable machines.

(BEGIN WARNING)

As a resident of the sun.audio package, the interface is subject to change without notice. The class summary was derived from the DiffCAD class dumper.

(END WARNING)

(B-heading) Class Summary

```
public class AudioData extends java.lang.Object {
    public void AudioData(byte a[]);
}
```

(B-heading) Class Usage

Suppose the following variables are predefined:

```
AudioStream as;
byte bytes[];
```

To make an instance of the AudioData:

```
ad = new AudioData(bytes);
```

To get data from an AudioStream:

```
ad = as.getData();
```

(A-heading) The AudioDataStream Class

The AudioDataStream resides in the sun.audio package. It extends ByteArrayInputStream. An AudioDataStream instance provides an input stream to play AudioData.

(BEGIN WARNING)

As a resident of the sun.audio package, the interface is subject to change without notice. The class summary was derived from the DiffCAD class dumper.

(END WARNING)

(B-heading) Class Summary

```
public class AudioDataStream extends ByteArrayInputStream {
    public void AudioDataStream(sun.audio.AudioData a);
}
```

(B-heading) Class Usage

The AudioDataStream has only one public method, its constructor.

Suppose the following variables are predefined:

```
byte ulawData[];
AudioData audioData = new AudioData ( ulawData);
```

To make an instance of the AudioDataStream:

```
audioDataStream = new AudioDataStream (audioData);
```

(B-heading) Reading and Playing an AU File

The following code fragment, from the (BEGIN ON CDROM) UlawCodec class (END ON CD ROM) shows how to read data from a Sun AU file. This is used to make and play an AudioDataStream instance:

```
package lyon.audio;
```

```

import java.io.*;
import sun.audio.*;
import futils.Futil;
public class UlawCodec implements Runnable {
private byte ulawData[];
private double doubleArray[];
private  AudioDataStream audioDataStream = null;
private  AudioStream audioStream = null;
...
// -----
-----
//
// public ReadAUFile(String name);
//
// This constructor can be used to acquire the audio
samples
// from an AU file
//
// -----
-----
public void readAUFile(String fileName){
    // force recomputation
    // of the doubleArray, as it is invalid
    doubleArray = null;
    try {
        audioStream = new AudioStream(new
FileInputStream(fileName));
        // AudioStream constructor expects data stream from AU
file as input
        ulawData = new byte[audioStream.getLength()];
        audioStream.read(ulawData, 0, audioStream.getLength());
    }catch(Exception e){}
}
public void readAUFile() {
    String fileName = Futil.getReadFileName();
    readAUFile(fileName);
}
}

```

(A-heading) The AudioStreamSequence Class

The AudioStreamSequence resides in the sun.audio package. An instance of the AudioStreamSequence can be used to convert a sequence of input streams into a single InputStream. This is typically used to play two audio clips in sequence. (BEGIN WARNING) As a resident of the sun.audio package, the interface is subject to change without notice. The class summary was derived from the DiffCAD class dumper. (END WARNING)

(B-heading) Class Summary

```

public class AudioStreamSequence extends
java.io.InputStream {

```

```

    public void AudioStreamSequence(java.util.Enumeration a);
    public int read();
    public int read(byte a[], int b, int c);
}

```

(B-heading) Class Usage

Suppose the following variables are predefined:

```

    AudioStream as1, as2;
    Vector v = new Vector();
    v.addElement(as1);
    v.addElement(as2);
    int lengthRead, length, position;
    byte bytes[];

```

To make a new instance of the AudioStreamSequence:

```

    AudioStreamSequence ass = new
    AudioStreamSequence(v.elements());

```

To play the AudioStreamSequence instance:

```

    AudioPlayer.player.start(audiostream);

```

To read from an instance of the AudioStreamSequence and flip to the next stream if an EOF is encountered (throws IOException):

```

    lengthRead = ass.read(bytes, position, length);
    lengthRead = ass.read();

```

(A-heading) The AudioPlayer Class

The AudioPlayer class resides in the sun.audio package. This class provides an interface to play and instance of the AudioStream.

(BEGIN WARNING) As a resident of the sun.audio package, the interface is subject to change without notice. The class summary was derived from the DiffCAD class dumper.
(END WARNING)

(B-heading) Class Summary

```

import java.io.PrintStream;
import java.lang.System;
public class AudioPlayer extends java.lang.Thread {
    public static final sun.audio.AudioPlayer player;
    private void AudioPlayer();
    public synchronized void start(java.io.InputStream a);
    public synchronized void stop(java.io.InputStream a);
    public void run();
}

```

(B-heading) Class Usage

Suppose the following variables are pre-defines:

```

    AudioStream as;
    AudioData ad = as.getData();

```

To play an audio stream use:

```

    AudioPlayer.player.start(as);

```

To stop playing an audio stream use:

```

    AudioPlayer.player.stop(as);

```

(A-heading) The μ -law CODEC concept

In this section we show how to use a software CODEC (COder-DECoder) to decode a Sun AU audio file and play it. The Java API uses a μ -law (pronounced mu-law) compression technique for playing audio files. The supported μ -law format consists of logarithmically companded, 8 khz sample rate, byte-quantized, voice-grade audio. The word *compandor* is a contraction of “compressor” and “expander” [BTL]. The sample time for an 8000 samples per second system is $1/8000 = 0.000125$ second = $125 \mu s$. Such a format generates an $8000 \text{ sample/second} * 8 \text{ bits / sample} = 64 \text{ kbps}$ data stream and is common in telephony. Also, the peak bandwidth of such a compression format is

$$\text{Bandwidth} = \frac{\text{Sampling Rate}}{2} = 4\text{khz} \quad (5.17)$$

The International Telecommunication Union (ITU) formerly CCITT, has created a specification called, G.711. There are two PCM algorithms defined within the G.711 standard, “A-Law” and μ -law. In both the “A-Law” and μ -law format, the sample rate is 8 khz. In a linear PCM system there are uniform voltage quantization steps [Bates]. A copy of the G.711 specification is available for sale at http://www.itu.int/itudoc/itu-t/rec/g/g700-799/g711_27434.html (it is 20 Swiss francs, last we checked). The Sun reference implementation is on the CD, along with a wealth of other audio information. The μ -law encoding formula is given by:

$$y = F(x) = \text{sign}(x)V_{\max} \frac{\ln\left(1 + \frac{\mu x}{V_{\max}}\right)}{\ln(1 + \mu)} \quad (5.18)$$

where

$$-V_{\max} \leq x \leq V_{\max}$$

The “A-law formula is given by:

$$y = \frac{A}{1 + \ln A} \left(\frac{x}{V_{\max}} \right), \left| \frac{x}{V_{\max}} \right| < \frac{1}{A} \quad (5.19).$$

and

$$y = \frac{\text{sign}(x)}{1 + \ln A} \left(1 + \ln \left(\frac{Ax}{V_{\max}} \right) \right), \frac{1}{A} \leq \left| \frac{x}{V_{\max}} \right| \leq 1$$

Typical values of the compression parameters used in (5.18) and (5.19) are:

$$\mu = 100 \text{ and } 255 \quad (5.20)$$

$$A = 87.6$$

A graph of (5.18), with $V_{\max} = 1$, and $\mu = 255$ is shown in Figure 5.7. The value of $\mu = 255$ is used for North American telephone transmission. The value of $A=87.6$ is used for European telephone transmission. The rationale for companding is that the human ear has an inability to differentiate between amplitudes of sound waves as the amplitude increases [Embree].

Figure 5.7 Graph of the μ -law transfer function, with $\mu=255$.

One metric of PCM performance is the ratio of the signal power to quantization noise power (signal-to-quantizing distortion ratio). The basic idea behind “A-law” and μ -law companding is that a logarithmic curve may be used to improve the signal-to-quantizing distortion ratio at low signal levels.

Binary PCM will have a number of quantization levels given by:

$$N_q = 2^{N_b} \tag{5.21}$$

where

N_q = the number of quantization levels

N_b = the number of bits per sample

The signal power varies from 0 to 1, inclusive, and is given by:

$$S \in [0...1]$$

The signal voltage, x , varies from:

$$-V_{\max} \leq x \leq V_{\max} \tag{5.22}.$$

For simplicity we assume that

$$V_{\max} = 1 \tag{5.23}$$

The uniform quantizer divides the signal voltage evenly among the number of quantization levels. The uniform quantizers’ quantization error voltage is given by:

$$-\frac{1}{2^{N_b}} \leq \varepsilon \leq \frac{1}{2^{N_b}} \tag{5.24}$$

where

ε = quantization error voltage

The average quantization error is 0, but the root-mean-square (RMS) value of the quantization error is the mean square quantization noise power. The term “root-mean-square” refers to the fact that the computation is performed by taking the square-root of the mean of error voltage squared [Carlson and Gisser]. Recall that for a continuous

probability distribution function, the expectation is taken by $E[X] = \int_{-\infty}^{\infty} xf_x(x)dx$ and the

variance is taken by $\sigma_x^2(t) = E[|X(t)|^2]$. The variance expands to $\sigma_x^2(t) = \int_{-\infty}^{\infty} x^2 f_x(x)dx$.

The reader may also recall that for a discrete random variable, $E[X] = mean = \frac{1}{N} \sum_{i=1}^N x_i$

and $\sigma_x^2(t) = variance = E[|X(t)|^2] = \frac{1}{N} \sum_{i=1}^N (x_i - E[X])^2$.

Thus, we compute the RMS error by integrating the square of the quantization error voltage over the range in (5.24), assuming a zero mean:

$$\sigma^2 = \frac{1}{(2 / N_q)} \int_{-1/N_q}^{1/N_q} \varepsilon^2 d\varepsilon = \frac{1}{3N_q^2} \tag{5.25}.$$

When the maximum signal voltage is constrained, as in (5.22) and (5.23) we compute the signal-to-quantization noise power as:

$$\begin{aligned} SNR_D &= 10 \log_{10} (3 \times 2^{2N_b} S_x) \\ SNR_D &= 10 \log_{10} 3 + 20N_b \log_{10} 2S_x \end{aligned} \quad (5.26)$$

Where the signal power is given by S_x . If the signal power is equal to 1, then the range on the upper bound for the signal-to-quantization noise power is:

$$SNR_D \leq 4.8 + 6N_b \quad (5.27).$$

With the 8-bit PCM system and uniform quantization, the best we can hope for is a SNR of 52.8 dB. Note that the SNR (in dB) falls off linearly as a function of the power in (5.26). It can be shown that the companding equations of (5.18) and (5.19) will provide an improvement in the SNR when the signal power falls below -20 dB.

It must also be mentioned that for signal powers above -20 dB, companding degrades performance, relative to uniform quantization, assuming that the PDF (Probability Distribution Function) of a voice signal has a *Laplace* distribution of the form

$$p(x) = \frac{1}{2} \alpha e^{-\alpha |x|}$$

[Carlson].

The compression parameter values given in (5.20) are based on an assumed PDF of an input signal. The PDF assumption is required for telephony applications. However, in the instance of audio files that are stored on static media (such as CD ROM, or web server hard drive) the computation of the PDF can be performed off-line. For such a system, the SNR is

$$SNR_D = \frac{S_x}{\sigma^2} = \frac{3N_q^2 S_x}{K_z} \quad (5.27)$$

where

$$K_z = 2 \int_0^1 \frac{p(x)}{[y']^2}$$

computing the derivative of (5.18) with respect to x , and substituting into (5.27) yields

$$K_z = 2 \int_0^1 \frac{p(x) (V_{max} + \mu x)^4 \ln(1 + \mu)^4}{\mu^4 V_{max}^4} dx \quad (5.28)$$

Once the PDF, $p(x)$, is computed, the companding parameter can be precomputed using a criterion of optimality based on the bit rate budget. For example, a bit rate budget of 16 kbps might require a 4 bit sample with a 4 khz sampling rate. Such a system might be used to stream audio (transmit in real time) via a low data-rate phone connection. Ideally, a server should be able to resample and compand the audio before transmission, in response to the bit rate budget.

(A-heading) The UlawCodec Class

The UlawCodec class is a public class that resides in the lyon.audio package. (BEGIN ONCDROM) The UlawCodec class performs the μ -law CODEC function, in addition to providing file save and open services.(END ON CD ROM) Java can, in principle, process any file format. The basic problem, however, is that Java's present API is only able to play an AudioDataStream that is μ -law encoded. Thus, while Java is able to read, process and write any audio data format file, an intermediary μ -law encoding is required to play the audio, at present.

(B-heading) Class Summary

```
public class UlawCodec implements Runnable {
    public UlawCodec()
    public UlawCodec(String name)
    public UlawCodec(short linearArrayOfShort[])
    public UlawCodec(double linearArrayOfDouble[])
    public UlawCodec(byte ulawArrayOfByte[])
    public void readAUFile(String fileName)
    public void readAUFile()
    public void writeAUFile(String fileName)
    public void writeAUFile()
    public void playSync()
    public void playAsync()
    public byte [] getUlawData()
    public void setUlawData(byte ulawArrayOfByte[])
    public double[] getDoubleArray()
    public int getLength()
    public double getDuration()
    public void reverseUlaw()
    public static void main(String argc[])
}
```

(B-heading) Class Usage

The UlawCodec has several constructors, each has, as its main goal, to construct a μ -law encoded byte array in a private storage area. The only way to obtain access to this storage area is via the getUlawData and setUlawData methods. This is due, in part, to a series of parallel data structures that must maintain their consistency. For example, when you invoke the getDoubleArray method, a check is performed to see if the internal DoubleArray is null. If the array is null, it is set using computations involving the ulawData. The consistency maintenance mechanism is invisible to the programmer.

Suppose the following variables are pre-defined:

```
UlawCodec ulc;
String fileName;
byte ulawArrayOfByte[];
short linearArrayOfShort[];
Double linearArrayOfDouble[];
int length;
double timeInSeconds;
String args[];
```

To read in a Sun AU file, using a standard file open dialog box:

```

    ulc = new UlawCodec();
To read in a Sun AU file, using a file name:
    ulc = new UlawCodec(fileName);
To construct a UlawCodec instance from a 16 bit linear data array:
    ulc = new UlawCodec(linearArrayOfShort);
To construct a UlawCodec instance from a linear double array:
    ulc = new UlawCodec(linearArrayOfDouble);
To overwrite the internal data and read in a new AU file, given a file name:
    ulc.readAUFile(fileName);
To prompt the user for a read file name and overwrite the internal data:
    ulc.readAUFile();
To write the internal data as new AU file, given a file name:
    ulc.writeAUFile(fileName);
To prompt the user for a file name, then write a Sun AU file:
    ulc.writeAUFile();
To play synchronously, returning only after the sound is played:
    ulc.playSync();
To play asynchronously, returning right away and playing the sound in the background:
    ulc.playAsync();
To get the raw companded byte data:
    ulawArrayOfByte = ulc.getUlawData();
To set the raw companded byte data:
    ulc.setUlawData(ulawArrayOfByte);
To get the data as an array of linear doubles:
    linearArrayOfDouble = ulc.getDoubleArray();
To get the number of samples:
    length = ulc.getLength();
To get the play time in seconds:
    timeInSeconds = ulc.getDuration();
To reverse the Ulaw data, forcing recomputation of the DoubleArray:
    ulc.reverseUlaw();
To test the read play and write methods:
    UlawCODEC.main(args);

```

(B-heading) Reading and writing μ -law

The following example, excerpted from the UlawCodec.java file, shows how the main method is implemented:

```

1. public static void main(String argc[]){
2.   UlawCodec ulc = new UlawCodec();
3.   ulc.playSync();
4.   ulc.writeAUFile();
5. }

```

(BEGIN ON CDROM) Line 2 shows the default constructor for the CODEC. The default constructor opens the standard file dialog box in order for the user to select a AU file.

(END ON CDROM) Line 3 plays the sound and does not return until the sounds has completed playing. The writeAUFile opens a dialog box and the user must type in a file name to save the AU file.

(A-heading) The Oscillator Class

The Oscillator class is a public class in the `lyon.audio` package and is independent of all other packages. Several instances of the Oscillator class may be made to create banks of Oscillators. The Oscillator class makes use of double precision data arrays and can make very low-distortion waveforms. The weak link in the chain is the Java API's voice grade audio realization. To combat this limitation (before Sun does) would require the development of Java native methods for playing audio on all platforms. Such a development effort requires many hours of skilled labor.

(B-heading) Class Summary

```
public class Oscillator {
    public Oscillator(double frequency, int length)
    public double[] getSineWave()
    public double[] getSquareWave()
    public double[] getSawWave()
    public double[] getTriangleWave()
    public double getDuration()
    public int getSampleRate()
    public double getFrequency()
    public void setModulationIndex(double I)
    public void setModulationFrequency(double fm)
    public double[] getFM()
    public double[] getAM()
}
```

(B-heading) Class Usage

The Oscillator class has a number of private properties that are accessed via `get` and `set` methods. An Oscillator instance is created for a fixed carrier frequency and number of samples. All waveforms vary from -1 to 1. Suppose the following variables are predefined:

```
double frequency = 440;
double length = 2000; // the total number of samples
Oscillator osc;
double audioData[];
double timeInSeconds;
int sampleRate;
double indexOfModulation;
```

Then to make an instance of an Oscillator:

```
osc = new Oscillator(frequency, length);
```

To get sine, square, saw tooth and triangle waves:

```
audioData = osc.getSineWave();
audioData = osc.getSquareWave();
audioData = osc.getSawWave();
audioData = osc.getTriangleWave();
```

To get the time the wave form will last, in seconds:

```
timeInSeconds = osc.getDuration();
```

To get the sample rate, in Hz:

```
sampleRate = osc.getSampleRate()
```

To get the frequency, in Hz:

```
frequency = osc.getFrequency();
```

To set the index of modulation of the FM oscillator:

```
osc.setModulationIndex(indexOfModulation);
```

To set the modulation frequency of both the AM and FM oscillators:

```
osc.setModulationFrequency(frequency);
```

```
audioData = osc.getFM();
```

```
audioData = osc.getAM();
```

The index of modulation and the modulation frequency are topics that are covered in more detail in Chapter 6.

(B-heading) Class Examples

The `AudioFrame` class is able to generate a series of waveforms using the `Oscillator` class and the `UlawCodec` class.

```
public class AudioFrame extends ClosableFrame {
    private UlawCodec ulc;
    private Oscillator osc =
        new Oscillator(440,4000);
    ...
public class AudioFrame extends ClosableFrame {
    private UlawCodec ulc;
    private Oscillator osc =
        new Oscillator(440,4000);
    ...
public void play() {
    ulc.playSync();
}
public void sineWave() {
    ulc = new UlawCodec(
        osc.getSineWave());
    play();
}
public void squareWave() {
    ulc = new UlawCodec(
        osc.getSquareWave());
    play();
}
public void sawWave() {
    ulc = new UlawCodec(
        osc.getSawWave());
    play();
}
public void triangleWave() {
    ulc = new UlawCodec(
        osc.getTriangleWave());
    play();
}
public void am() {
```

```

        osc.setModulationIndex(0.5d);
        osc.setModulationFrequency(200d);
        ulc = new UlawCodec(
            osc.getAM());
        play();
    }

    public void fm() {
        osc.setModulationIndex(0.5d);
        osc.setModulationFrequency(200d);
        ulc = new UlawCodec(
            osc.getFM());
        play();
    }
}

```

(B-heading) Class Implementation

An Oscillator is able to generate repeated waveforms by constructing a single cycle of the waveform into a wavetable. The wavetable is copied repeatedly into an array of double data known, internally as `audioData`.

The Oscillator class is implemented with a series of private class variables:

```

1.  package lyon.audio;
2.  import futils.utils.Computation;

3.  public class Oscillator {
4.  private double audioData[];
5.  private double waveTable[];

```

Lines 4 and 5 show that the `audioData` `waveTables` are unallocated until the constructor is invoked. Line 6 shows the `sampleRate`. The constructor could be overloaded to take other sample rates, but 8000 Hz is the default for Java's current API.

```

6.  private int sampleRate = 8000;

```

Line 7 shows the frequency, in Hz. For a sine wave, this is the number of wave table cycles that must be clocked out, per second. `lambda` is the number of seconds in the period of one cycle. Line 9 shows the number of samples in a single cycle of the wave table, if this were computed with precision. Keep in mind that the length of a wave table is always an integer and the `samplesPerCycle` must be converted as a result.

```

7.  private double frequency;
8.  private double lambda;
9.  private double samplesPerCycle;

```

Oscillator construction requires that the carrier frequency and number of cycles be known. Line 3 show the memory allocation for the `audioData`.

```

1.  public Oscillator(double frequency_, int length) {
2.  frequency = frequency_;
3.  audioData = new double[length];

```

Once the period of the waveform is computed, on line 5, we are able to compute the number of samples in a cycle of the wave table. This is the `samplesPerCycle` variable, cast into an integer. Given the integral approximation, the computation for the actual frequency at which the wave table is clocked out is:

$$f_{actual} = sampleRate / waveTable.length \quad (5.29)$$

With the wave table length being:

$$waveTable.length = round(sampleRate / f) \quad (5.30)$$

To compute the error in the digital oscillator instance, we subtract the frequency that we wanted from the rate cycles are clocked out of the wave table.

$$f_e = f - f_{actual}$$

For example, for a frequency of 440 Hz, `waveTable.length = 18` `sampleRate = 8000` `audioData.length = 4000` and the actual frequency is 444.444 Hz. Exact frequencies may be had when the frequency desired is an exact multiple of the `sampleRate`. For example, 400 will be reproduced with precision because $8000/400 = \text{a } waveTable.length \text{ of } 20$.

```

4. //the period of the wave form is
5. lambda = 1/frequency;
6. //The number of samples per period is
7. samplesPerCycle = sampleRate * lambda;

8. delta_freq = 1/samplesPerCycle;
9. waveTable =
10.     new double[(int) samplesPerCycle];

11. }
```

(B-heading) Building the WaveTable

The `AudioDataFromTable` method in the `Oscillator` class is used to turn a single cycle of the `WaveTable` into a long array of audio data. A constraint on the `audioData` array is that it must have an absolute value that is strictly less than 1 (due to the companding formulas).

```

1. private double[] AudioDataFromTable() {
2.     int k = 0;
3.     for (int i = 0; i < audioData.length; i++) {
Line 4 builds the audioData from the waveTable. While the indexes, i and k both begin at
0, lines 6 and 7 reset k, while index i increments on.
4.         audioData[i] = waveTable[k];
5.         k++;
6.         if (k >= waveTable.length)
7.             k = 0;
8.     }
9.     System.out.println("\nlambda="+lambda+
10.         "\nfrequency = "+frequency+
11.         "\nwaveTable.length = "+waveTable.length+
12.         "\nsampleRate = "+sampleRate+
13.         "\naudioData.length = "+audioData.length+
14.         "\nactual frequency = "+actualFrequency());
15.     return audioData;
16. }
```

In the `getSineWave` method, the `waveTable` is computed for a single cycle. To make sure that the absolute value of the amplitude of the sine wave is always less than 1, it is first multiplied by 0.98, on line 20.

```

17. public double[] getSineWave() {
18.     for (int i=0; i<waveTable.length; i++)
19.         waveTable[i] =
20.             0.98*Math.sin(twopi * i/waveTable.length);
21.     return AudioDataFromTable();
22. }

```

To build a wave table for a saw wave, we set the initial voltage to -1, then compute a change in voltage, dv using the length of the wave table, L so that $V_0 = -1, dv = \frac{2}{L}$. then, after $L-1$ intervals, the final voltage will reach a value of $1-dv$. The following code implements the `getSawWave` method:

```

1. public double[] getSawWave() {

```

In line 2, the initial voltage is set to a value that is a little higher than 1.0. This is due to the constraint on the CODEC's input. Also, in line 4, the check is `i<waveTable.length` rather than `i <= waveTable.length`. Thus the saw wave will end at $(v-dv)$, rather than at 1.0 volts.

```

2.     double v = -0.99;
3.     double dv = 2.0 / (double) waveTable.length;
4.     for (int i=0; i<waveTable.length; i++){
5.         waveTable[i] = v;
6.         v += dv;
7.     }
8.     System.out.println("Sawwave ends at:"+(double)(v-dv));
9.     return AudioDataFromTable();
10. }

```

The saw wave output is shown in Figure 5.8.

Figure 5.8 The saw wave output

(A-heading) The `DoubleDataProducer` Interface

The `DoubleDataProducer` interface is a public interface that resides in the `lyon.audio` package. The purpose of the `DoubleDataProducer` interface is to isolate the `OscopeFrame` class (see next section) from application specific code needed to obtain an array of data. Classes that implement the `DoubleDataProducer` interface must provide methods whose signature is consistent with lines 3 and 4:

```

1. package lyon.audio;
2. public interface DoubleDataProducer {
3.     double [] getDoubleData();
4.     void openDataFile();
5. }

```

For an example of usage, see the following section.

(A-heading) The OscopeFrame Class

The OscopeFrame class is a public abstract class in the lyon.audio package. The OscopeFrame may be extended in order to add an oscilloscope type feature to any frame. The OscopeFrame has its own paint method, so the user must be careful to call super.paint if a paint method is implemented in the subclass. The OscopeFrame has four private scrollbars whose events are handled locally. Thus, any subframe that handles events must be sure to call super.handleEvent at the end of the handleEvent method. Also, the OscopeFrame has a main menu bar setting. This may be overridden by the subclass, though doing so will remove the main menu bar features. The OscopeFrame extends the PictFrame class. The PictFrame class implements a feature in the main menubar that enables the OscopeFrame to save a copy of itself as a pict file. The vector part of pict file format is used so that the OscopeFrame has a vector output that may be edited with any draw program (including MS Word's draw utility). Figure 5.10 is an example of a vector output that the OscopeFrame is able to generate. An example of the main menu bar that the OscopeFrame sets up is shown in Figure 5.9.

Figure 5.9. The OscopeFrame SaveMenu Permits the saving of Vector Output

Figure 5.10. An example of the vector output from the OscopeFrame
The OscopeFrame implements the DoubleDataProducer interface. As a result, the programmer must implement the getDoubleData and openDataFile methods, or the subclass will be abstract. When openDataFile is invoked, the subclass must respond by invoking setDoubleData.
An example of the OscopeFrame is shown in Figure 5.11.

Figure 5.11. The Triangle Wave in the OscopeFrame

Scroll bar adjustments permit translation and scaling of the waveform, both in the time domain and in the amplitude domain. Another view of the triangle wave of Figure 5.11 is shown in Figure 5.12.

Figure 5.12. Another View of the Triangle Wave in the OscopeFrame

From Figure 5.12 we can visually measure ten 250 micro second divisions cycles in the waveform. This corresponds to a frequency of $1/(2500 \text{ micro seconds}) = 400 \text{ Hz}$. This is exactly what was specified for the oscillator (a multiple of the 8000 Hz sampling rate).

(B-heading) Class Summary

```
package lyon.audio;
import java.awt.*;
import futils.utils.Draw;
import futils.Futil;
import lyon.dclap.PictFrame;
```

```
public abstract class OscopeFrame
    extends PictFrame implements DoubleDataProducer {
public OscopeFrame(String title)
public void setDoubleData(double [] d)
public void setGridColor(Color c)
public void paint(Graphics g)
```

```

    public boolean handleEvent(Event e)
    }

```

(B-heading) Class Usage

Since the `OscopeFrame` implements the `DoubleDataProducer` interface, the subclass must implement the `getDoubleData` and `OpenDataFile` methods. The `AudioFrame` subclasses the `OscopeFrame`, as shown in the following code:

```

package lyon.audio;
import java.awt.*;
import gui.*;
import grapher.*;
import observers.*;
import futils.utils.*;
import futils.bench.*;
import futils.Futil;
import lyon.dclap.PictFrame;
public class AudioFrame extends OscopeFrame {
    UlawCodec ulc;

```

The `AudioFrame` constructor shows the name of the Frame being passed to the `OscopeFrame`

```

    public AudioFrame(String name) {
        super(name);

```

The `getDoubleData` and `OpenDataFile` methods are defined, to keep the `AudioFrame` class from being abstract:

```

    public double [] getDoubleData() {
        return ulc.getDoubleArray();
    }
    public void openDataFile() {
        ulc = new UlawCodec();
        setDoubleData(ulc.getDoubleArray());
        ulc.playAsync();
    }

```

While no paint method was defined in the `AudioFrame`, there were events. Thus, at the end of the event handler, we are careful to call `super.handleEvent`, so that the `OscopeFrame` events are processed:

```

    public boolean handleEvent(Event e) {
        if (Evt.match(e,saveSound_mi)) {
            saveAs();
            return true;
        }....
        return super.handleEvent(e);
    }

```

(B-heading) Modifying the `OscopeFrame`

Once the user starts to use the `OscopeFrame`, there may be a need to add features to the `OscopeFrame` class, without altering the existing API. One way this can be accomplished is by adding to the pre-existing menubar. In the following code, we demonstrate how to

add the feature of turning the `OscopeFrame` grid on and off using a menu item in the main menu bar. The `OscopeFrame` extends the `PictFrame`, and, in doing so, inherits an existing menu bar. To add an item to the menu bar, we add the following instance variables to the `OscopeFrame` class:

```
private boolean drawGrid = true;
private MenuBar mb = getMenuBar();
private Menu m = new Menu("Scope Menu");

private MenuItem drawGrid_mi =
    new MenuItem("[d]raw grid");
```

The idea is that, when the “d-key” is depressed, or when the user selects the “[d]raw grid” item from the main menu bar, we want to turn off the drawing of the grid. This is accomplished in two places. The first place is the `handleEvent` method:

```
public boolean handleEvent(Event e) {
    if(Evt.match(e,drawGrid_mi)) {
        drawGrid = ! drawGrid;
        return true;
    }....
```

The second place that requires modification is in the `drawTrace` method:

```
private void drawTrace(Graphics g) {
    doubleData = getDoubleData();

    double limit = (double)(doubleData.length *
xScaleFactor);

    dim = size();
    height = dim.height;
    width = dim.width;
```

In the following line, we check the status of the `drawGrid` variable to see if a grid should be draw.

```
if (drawGrid)
    Draw.grid(20, g, gridColor);
```

....

(B-heading) How does the `OscopeFrame` do the scaling labels?

One of the unique features of the `OscopeFrame` is that it automatically switches to a next logical range when the user clicks in the time-scale or voltage-scale scrollbars. This is an interface hack that is clever enough to warrant some explanation.

The `OscopeFrame` has a series of pre-defined scale factors, with corresponding labels, for both the time-scaling (y-scale) and the voltage-scaling (x-scale). These are held as private class variables and are described as follows:

```
1. private final double
2. xScaleFactors[] = {50000, 25000, 10000, 5000, 2500,
1000,
3. 250, 100, 50, 25,
10,
4. 2.5, 1, .5,
.25, .1,
```



```

5.    .025,    .01, .005, .0025,
    .001,
6. ;

7.    private final String xSFLabels[] = {"0.05 u", "0.1 u",
    "0.25 u",
8.            "0.5 u",    "1 u",
    "2.5 u",
9.            "5 u",    "10 u",
    "25 u",
10.           "50 u", "100
u",    "250 u",
11.           "500 u",    "1
m",    "2.5 m",
12.           "5 m",    "10 m",
    "25 m",
13.           "50 m", "100
m",    "250 m",
14.           "500 m",    "1
",    "2.5 ",
15.           "5 "};

```

Note that the `xsfStartIndex` indicates the default start point for both the scale factor and the label for the corresponding scale factor. This has a direct correspondence with the y-scale factors, shown in line 32.

```

16. private final int xsfStartIndex = 14;
17. private double xScaleFactor =
xScaleFactors[xsfStartIndex];
18. private double oldXScaleFactor =
xScaleFactors[xsfStartIndex];
19. private String xSFLabel = new
String(xSFLabels[xsfStartIndex]);

20. private final double
21. yScaleFactors[] = {
22.     200, 100, 50, 20, 10,
23.     2, 1, .5, .2, .1};

24. private final String
25.     ySFLabels[] = {
26.         "1 m", "2.5 m",
27.         "5 m", "10 m",
28.         "25 m", "50 m",
29.         "100 m", "250 m",
30.         "500 m", "1 ",
31.         "2.5 ", "5 "};

32. private final int ysfStartIndex = 8;

```

```

33. private double yScaleFactor = 1;
34. private double oldYScaleFactor = 1;
35. private String ySFLabel = new
String(ySFLabels[ysfStartIndex]);

```

Now, the clever bit is that the top and bottom scrollbar instances return integers that can be used to index into the label and scale arrays. This is shown in the following code, excerpted from the `handleEvent` method:

```

if(e.target == sbHorzTop) {
    int i = sbHorzTop.getValue();
    xScaleFactor = xScaleFactors[i];

    if (xScaleFactor != oldXScaleFactor) {
        xSFLabel = xSFLabels[i];
        repaint();
        oldXScaleFactor = xScaleFactor;
    }
    return true;
}

```

The `Scrollbar` characteristics are set so that they will only return a valid range for an index into the scale factors and scale labels:

```

// Set top scrollbar characteristics
sbHorzTop.setValues(xsfStartIndex, 0, 0, 24);
// Set right scrollbar characteristics
sbVertRight.setValues(ysfStartIndex, 0, 0, 11);

```

(A-heading) The `DoubleGraph` Class

Some programmers will not want to bother extending the `OscopeFrame` class, mentioned in the previous section. There are a few reasons for this. Some programmers may find the implements interface difficult to implement properly. Other may wish for a simpler API, which, though less general, does make the code easier to both write and understand. The `DoubleGraph` class will make an `OscopeFrame` and handle all the scrollbar events and interface implementations for you.

The `DoubleGraph` class resides in the `lyon.audio` package, but is so generally useful, that it could appear in a utilities class.

(B-heading) Class Summary

```

public class DoubleGraph extends OscopeFrame {
    DoubleGraph(double d[],String title)
    public double[] getDoubleData()
    public void openDataFile()
}

```

(B-heading) Class Usage

The `AudioFrame` class has a public method called `graphSound`. It shows the typical usage for the `DoubleGraph` class instance:

```

public void graphSound() {
    new DoubleGraph(ulc.getDoubleArray(), "Sound");
}

```

The act of instantiation displays the `OscopeFrame`, with scrollbars. (BEGIN NOTE) To avoid added complexity, there are no features for altering an existing `DoubleGraph`

frame. Just make another. The old frame will be collected by the garbage collector when all references to it are removed. You may note that the DoubleGraph instance has a life that ends after the graphSound method returns. The DoubleGraph instance will thus be a temporary one. It would be hard to build a simpler API for graphing an array of doubles.
(END NOTE)

(A-heading) Summary

This chapter has covered a lot of ground. Much of it was covered quickly and without rigor. The basic objective is to give the reader just enough background to understand the basic results of Fourier analysis so that the code of Chapter 6 may be better understood. A great deal of complexity is hidden from the user in the classes just presented. Code is reproduced, where appropriate to the exposition of the book. However, to get a real look at the code, the programs on the CD-ROM can make excellent reading.

Adding features to the classes presented in this chapter can be a fruitful source of exercises for the reader. For example, turning off the grid is just one feature that may be added to the OscopeFrame. There are so many others. Coloring a grid, extending the OscopeFrame to make a DualTraceOscopeFrame, creating an x vs. y plot in the OscopeFrame, etc.

(CN) 6 Digital Audio Transform Recipes

...then thus he cried,
When I the process have in memory,
How thou hast wearied me on every side...
William Wordsworth, 1888

This chapter covers some of the finer points of one-dimensional digital signal processing. The last chapter showed some of the mathematical basis for the Fourier transform and pointed out some of its properties. This chapter addresses how to implement the Fourier transform using uniformly sampled audio. We start with a description of the DFT (Discrete Fourier Transform) and IDFT (Inverse Discrete Fourier Transform). We show how the computation of the DFT and IDFT may be performed in Java and show why such operations are typically held as slow. A speed up of the DFT and IDFT is discussed using a class of algorithms known as the FFT (Fast Fourier Transform) and the IFFT (Inverse Fast Fourier Transform).

As the DFT and FFT are covered, we demonstrate the computation of the psd (Power Spectral Density) mentioned in the previous chapter. The remaining portion of the chapter is dedicated to applications of these transforms. The applications include filtering, windowing, pitch-shifting and the spectral analysis of resampling.

(A-heading) The Discrete Fourier Transform

Let

$$v_j, j \in [0 \dots N - 1] \quad (6.1)$$

be the sampled version of the waveform, $v(t)$ where N is the number of samples.

(BEGIN NOTE) Equation (6.1) numbers from zero, rather than one, to reflect the start point of arrays in Java.

(END NOTE) Recall that the Fourier transform of $v(t)$ was given by

$$V(f) = F[v(t)] = \int_{-\infty}^{\infty} v(t)e^{-2\pi ft} dt \quad (6.2).$$

(BEGIN NOTE)

$V(f)$ can only exist if $v(t)$ is absolutely integrable, i.e.

$$\int_{-\infty}^{\infty} |v(t)| dt < +\infty$$

(END NOTE)

Recall also, that the inverse Fourier transform of $V(f)$ was given by

$$v(t) = F^{-1}[V(f)] = \int_{-\infty}^{\infty} V(f)e^{2\pi ft} dt \quad (6.3).$$

In order to compute (6.2) for the sampled waveform of (6.1), we must perform a discrete time Fourier transform called the DFT (Discrete Fourier Transform). The DFT is given by

$$V_k = \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi ijk/N} v_j \quad (6.4).$$

Recall, from Chapter 5, Euler's relation,

$$e^{i\theta} = \cos \theta + i \sin \theta \quad (6.4a).$$

This permits the alternate formulation of the kernel of the transform in (6.4) as:

$$e^{-2\pi ijk/N} = \cos(-2\pi jk/N) + i \sin(-2\pi jk/N) \quad (6.4b).$$

Recall that an even function is one that has the property that $f(-x) = f(x)$. Also, that an odd function has the property that $f(-x) = -f(x)$. Since $\sin(-\theta) = -\sin \theta$, sine is an odd function. Also, since $\cos(-\theta) = \cos \theta$, cosine is an even function. Using the odd-even function properties of sine and cosine, we rewrite (6.4b) as:

$$e^{-2\pi ijk/N} = \cos(2\pi jk/N) - i \sin(2\pi jk/N) \quad (6.4c).$$

Substituting (6.4c) into (6.4) we obtain the form of the DFT used in the Java implementation:

$$V_k = \frac{1}{N} \sum_{j=0}^{N-1} (\cos(2\pi jk/N) - i \sin(2\pi jk/N)) v_j \quad (6.5).$$

When implementing the Java program we compute the real and imaginary parts of (6.5) using:

```
for(int j = 0; j < N; j++) {
    twoPijkOnN = twoPikOnN * j;
    r_data[k] += v[j] * Math.cos( twoPijkOnN );
    i_data[k] -= v[j] * Math.sin( twoPijkOnN );
}
r_data[k] /= N;
i_data[k] /= N;
```

The complete routine for the DFT is given at the bottom of the section. Recall that the above loop is executed N times. The index, k is varied from $0 \dots N-1$. Why? Because the

DFT divides the spectrum into N parts. Each spectral division is accessed using the *frequency index*, k . The frequency, f_k , is computed from k using:

$$f_k = \frac{k}{N\Delta t} \quad (6.6),$$

where the sampling period, Δt , is given by:

$$\Delta t = 1 / f_s \quad (6.7).$$

An examination of (6.5) and (6.6) shows that the spectrum is described by integral harmonics of f_s / N . For example, suppose that the sampling rate, f_s , is 8000 Hz and that the number of points is 2048; then the smallest change in frequency that can be detected is given by $f_1 = \frac{1}{2048} 8000 \approx 4 \text{ Hz}$. Therefore, integral harmonics of 4 Hz will be used to approximate $v(t)$.

The psd (Power Spectral Density) gives the power at a specific frequency index, psd_k . We compute the power at any f_k by summing the square of the real and imaginary components of the amplitude:

$$psd_k = \text{real}^2(V_k) + \text{imaginary}^2(V_k) \quad (6.8).$$

The amplitude spectral density is given by the square root of the power spectral density.

(BEGIN ON CDROM)

An implementation of the DFT is shown below.

```

1. package lyon.audio;
2. import java.io.*;
3. import java.awt.*;
4. import grapher.Graph;
5. public class FFT extends Frame {

6.     double r_data[] = null;
7.     double i_data[] = null;
8.     ...
   public double[] dft(double v[]) {
       int N = v.length;

       double t_img, t_real;

       double twoPiOnN;
       double twoPiijkOnN;

       // how many bits do we need?
       N=log2(N);
       //Truncate input data to a power of two
       // length = 2**(number of bits).
       N = 1<<N;

       double twoPiOnN = 2 * Math.PI / N;
```

```

// We truncate to a power of two so that
// we can compare execution times with the FFT.
// DFT generally does not need to truncate its input.

r_data = new double [N];
i_data = new double [N];
double psd[] = new double[N];

System.out.println("Executing DFT on "+N+" points...");

for(int k=0; k<N; k++) {

    twoPikOnN = twoPiOnN *k;

    for(int j = 0; j < N; j++) {
        twoPijkOnN = twoPikOnN * j;
        r_data[k] += v[j] * Math.cos( twoPijkOnN );
        i_data[k] -= v[j] * Math.sin( twoPijkOnN );
    }
    r_data[k] /= N;
    i_data[k] /= N;

    psd[k] =
        r_data[k] * r_data[k] +
        i_data[k] * i_data[k];

}
return(psd);
}
(END ONCDROM)

```

(B-heading) Bit Computations and a Log Review

We have notice that some students are confused by the use of logs in the following code:

1. // how many bits do we need?
2. $N = \log_2(N)$;
3. //Truncate input data to a power of two
4. // length = $2^{**}(\text{number of bits})$.
5. $N = 1 < N$;

If lines 2 and 5 are obvious, the reader should probably skip to the next section.

Otherwise, please read on! Logs are pretty basic things, but people do seem to forget them.

The log function is defined by:

$$\text{if } x = a^y \text{ then } y = \log_a x$$

For example, if $x = 2^{10}$ then $10 = \log_2 x$.

The following are known as the laws of logarithms:

$$\log_a(xy) = \log_a x + \log_a y$$

$$\log_a(x/y) = \log_a x - \log_a y$$

$$\log_a x^n = n \log_a x$$

For example, to find $\log_2 4096$ use:

$$4096 = 2^y$$

$$\ln 4096 = y \ln 2$$

$$\frac{\ln 4096}{\ln 2} = y = 12$$

so,

$$\log_2 x = \frac{\ln x}{\ln 2}$$

Also, to find $\log_B x$ use:

$$\log_B x = \frac{\ln x}{\ln B} = \frac{\log_{10} x}{\log_{10} B} \quad (6.9)$$

So, based on (6.9), line 2 computes the number of bits needed, rounding down, via truncation, the integral number, N . To compute (6.9) to the base 2, we use:

```
public static int log2(int n) {
    return (int) (Math.log(n)/Math.log(2.0));
}
```

Thus, line 2 replaces N , the number of items in the list, plus one, with the log of N to the base 2, rounded *down* (also known as the floor function) to the nearest integer.

```
2. N=log2(N);
```

Finally, line 5 creates a number which is an integral power of two:

```
5. N = 1<<N;
```

(BEGIN NOTE)

DFT does not need to truncate the length of the data to be a power of two. We will see, however, that there are implementations of the FFT (Fast Fourier Transform) that do require this truncation. To fairly compare the performance of the DFT implementation against the FFT implementation, we must perform the same truncation for both algorithms.

(END NOTE)

(A-heading) The `futils.Timer` Class, Benchmarking the DFT

Benchmarking is a craft that permits the measurement of hardware and software performance. One of the remarkable things about Java is that the compile-once-run-anywhere attribute enables the same byte codes to be executed on different implementations of the Java machine. In addition, we have similar Java *virtual* machines that are implemented on different hardware platforms. This enables a base-line comparison of different hardware platforms when executing the byte codes. One use of benchmarking is to measure the effect of the implementation of an algorithm on the execution time. Also of interest is the measurement of two different algorithm's execution time for the same data.

(BEGIN NOTE) There is a difference between the performance measurement of various algorithmic *implementations* and the performance measurement of various *algorithms*. Better algorithms are generally combined with better implementations to yield performance improvement.

(END NOTE).

(B-heading) Class Summary

```
package futils.bench;
import java.io.*;
public class Timer {
    public Timer()
    public void mark()
    public void record()
    public float elapsed()
    public void report(PrintStream pstream)
    public void report()
}
```

(B-heading) Class Usage

Suppose the following variables are pre-defined:

```
Timer t1;
PrintStream pstream;
```

To make an instance of a Timer class (with the elapsed time set to zero) use:

```
t1 = new Timer();
```

To mark the time (like resetting a stop-watch):

```
t1.mark();
```

To add the elapsed time to the total elapsed time since the record method was last invoked:

```
t1.record();
```

To return the elapsed time in seconds:

```
t1.elapsed();
```

To print the elapsed time, in seconds, to the PrintStream instance pstream:

```
t1.report(pstream);
```

To print the elapsed time, in seconds, to System.out:

```
t1.report();
```

(B-heading) BenchMarking the DFT method

In this section we show how to use the Timer class to measure the execution time of the DFT. This serves both as an example of how to use the DFT and how to benchmark a method. Benchmarking showed that executing the DFT on 2048 points took 55 seconds on a 100 Mhz PPC 601, without a JIT compiler. The following example of how to use the DFT in the lyon.audio package is contained in the AudioFrame class;

```
public void dft() {

    FFT f = new FFT();
    double [] doubleData = ulc.getDoubleArray();
    double [] psd;
```



```

// Time the fft
Timer t1 = new Timer();
t1.mark();

psd=f.dft(doubleData);

// Stop the timer and report.
t1.record();
System.out.println("Time to perform DFT:");
t1.report();
f.graphs();
new DoubleGraph(psd, "psd");
}

```

(A-heading) The Inverse DFT

Recall also, that the inverse Fourier transform of $V(f)$ was given by

$$v(t) = F^{-1}[V(f)] = \int_{-\infty}^{\infty} V(f)e^{2\pi ft} dt \quad (6.3).$$

In order to compute (6.3) for the sampled waveform of (6.1), we must perform an inverse discrete time Fourier transform called the IDFT (Inverse Discrete Fourier Transform).

The IDFT is given by

$$V_k = \sum_{j=0}^{N-1} e^{2\pi jk/N} v_j \quad (6.10).$$

(Begin NOTE)

Recall that in (6.4):

$$V_k = \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi jk/N} v_j \quad (6.4).$$

the summation result is multiplied by $1/N$. This is not the case in (6.10). In some expositions, both (6.10) and (6.4) are multiplied by $1/\sqrt{N}$, in order to keep the DFT and IDFT symmetric. We abandoned such an approach during development because it both complicates the presentation of the PSD and requires slightly more computation.

(END NOTE)

Recall, from Chapter 5, Euler's relation,

$$e^{i\theta} = \cos \theta + i \sin \theta \quad (6.4a).$$

Substituting (6.4a) into (6.10) results in:

$$v_k = \sum_{j=0}^{N-1} [\cos(2\pi jk/N) + i \sin(2\pi jk/N)] v_j \quad (6.11).$$

Recall that when two complex numbers are multiplied together, $z_1 z_2$ the result may be expressed as:

$$z_1 z_2 = (x_1 + iy_1)(x_2 + iy_2) = x_1 x_2 - y_1 y_2 + i(x_1 y_2 + y_1 x_2) \quad (6.11a)$$

Based on (6.11a) we conclude that the real part of $z_1 z_2$ is given by:

$$\text{real}(z_1 z_2) = x_1 x_2 - y_1 y_2 \quad (6.11b)$$

and the imaginary part of $z_1 z_2$ is given by:

$$\text{imaginary}(z_1 z_2) = x_1 y_2 + y_1 x_2 \quad (6.11c).$$

We are interested in a reconstruction of a real signal. Therefore we do not need to compute the result in (6.11c). Substituting (6.11b) into (6.11) yields:

$$\text{real}(v_k) = \sum_{j=0}^{N-1} [\cos(2\pi jk / N) \text{real}(V_j) - \sin(2\pi jk / N) \text{imaginary}(V_j)] \quad (6.12).$$

Computing only the real part of the IDFT saves $2N$ multiplies and N additions for each frequency index, k computed in (6.12) than in (6.11). A comparison of (6.12) with (6.5),

$$V_k = \frac{1}{N} \sum_{j=0}^{N-1} (\cos(2\pi jk / N) - i \sin(2\pi jk / N)) v_j \quad (6.5)$$

shows that both take about the same amount of time to compute (something that our experiments confirm).

The following code implements the IDFT shown in (6.12):

```
// assume that r_data and i_data are
// set. Also assume that the real
// value is to be returned
public double[] idft() {
    int N = r_data.length;
    double twoPiOnN = 2 * Math.PI / N;

    double twoPikOnN;
    double twoPijkOnN;

    double v[] = new double[N];

    System.out.println("Executing IDFT on "+N+" points...");

    for(int k=0; k<N; k++) {
        twoPikOnN = twoPiOnN *k;
        for(int j = 0; j < N; j++) {
            twoPijkOnN = twoPikOnN * j;
            v[k] += r_data[j] * Math.cos(twoPijkOnN )
                - i_data[j] * Math.sin(twoPijkOnN );
        }
    }
    return(v);
}
```

(A-heading) Numeric Check of the DFT and IDFT

A numeric check should be an integral part of every class. The FFT class contains a method called *testDFT* whose sole job it is to verify the correctness of the DFT and IDFT implementation. With the number of samples set to 8, the testing method is able to print the input and output points for human comparison.

```
public static void testDFT() {

    int N = 8;
    FFT f = new FFT(N);

    double v[];

    double x1[] = new double[N];
    for (int j=0; j<N; j++)
        x1[j] = j;

    // take dft
    f.dft(x1);

    v = f.idft();
    System.out.println("j\tx1[j]\tre[j]\tim[j]\t v[j]");
    for (int j=0; j < N; j++)
        System.out.println(
            j+"\t"+
            x1[j)+"\t"+
            f.r_data[j)+"\t"+
            f.i_data[j)+"\t"+
            v[j]);
}
```

(Begin NOTE) We print the intermediate DFT results to permit a detailed check against variations in implementation. Full data disclosure allows a base-line comparison between different implementations of the DFT. As we shall see in the following section, this is a comforting data result, particularly when compared with the FFT data. (END NOTE)

Executing IDFT on 8 points...

j	x1[j]	re[j]	im[j]	v[j]
0	0	3.5	0	-3.10862e-15
1	1	-0.5	1.20711	1
2	2	-0.5	0.5	2.00000
3	3	-0.5	0.207107	3
4	4	-0.5	0	4
5	5	-0.500000	-0.207107	5
6	6	-0.500000	-0.5	6
7	7	-0.5	-1.20711	7

While the input is not quite the same as the output, it is quite close.

(A-heading) The FFT

The Fast Fourier Transform (FFT) is a family of algorithms designed to speed the computation of the DFT:

$$V_k = \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi ijk/N} v_j \quad (6.4).$$

Direct computation of the DFT takes $O(N^2)$ complex multiplications while the FFT takes $O(N \log N)$ complex multiplications. In this section we show an FFT algorithm known as the decimation in time, radix 2 FFT algorithm (also known as the Cooley-Tukey algorithm). The Cooley-Tukey algorithm is probably one of the most widely used of the FFT algorithms. The radix 2 property means that the number of samples to be transformed must be a power of two. The decimation in time means that the algorithm performs a recursive subdivision of the input sequence into its odd and even members. We are able to perform this subdivision as a result of the Danielson-Lanczos Lemma:

$$V_k = \frac{1}{N} [V_k^e + W^k V_k^o] \quad \forall_{k \in [0 \leq k < N-1]}$$

Proof of the Danielson-Lanczos Lemma:

Let

$$W = e^{-2\pi i/N} \quad (6.13)$$

so that

$$W^{jk} = W^j W^{j(k-1)}$$

Substitute (6.13) into (6.4) to obtain

$$V_k = \frac{1}{N} \sum_{j=0}^{N-1} W^{jk} v_j \quad (6.14).$$

Separate (6.14) into its odd and even components. This is done by altering how the samples are indexed.

$$V_k = \frac{1}{N} \left[\sum_{j=0}^{N/2-1} W^{2jk} v_{2j} + \sum_{j=0}^{N/2-1} W^{(2j+1)k} v_{2j+1} \right] \quad (6.15)$$

Where (6.15) shows summations operating over the odd and even indices. For example, if $j = 0, 1, 2, 3, \dots$,

then

$$2j = 0, 2, 4, 6, \dots \text{ and } 2j + 1 = 1, 3, 5, \dots$$

Factoring the exponents in (6.15) yields

$$V_k = \frac{1}{N} \left[\sum_{j=0}^{N/2-1} W^{2jk} v_{2j} + \sum_{j=0}^{N/2-1} W^{2jk} W^k v_{2j+1} \right]$$

The W^k term in the right most summation is not a function of the index, so that:

$$V_k = \frac{1}{N} \left[\sum_{j=0}^{N/2-1} W^{2jk} v_{2j} + W^k \sum_{j=0}^{N/2-1} W^{2jk} v_{2j+1} \right] \quad (6.16).$$

To reflect the odd and even summations, (6.16) is rewritten as

$$V_k = \frac{1}{N} [V_k^e + W^k V_k^o] \quad \forall_{k \in [0 \leq k < N-1]} \quad (6.17).$$

Q.E.D.

The implications of (6.17) are that we can divide the sequence into odd and even numbered samples. Thus the Danielson-Lancos lemma enables a divide and conquer algorithm to recursively split the sample sequence in half. The computational result of the Danielson-Lancos lemma is that the $O(N^2)$ DFT may be computed in $O(N \log N)$ time. The Danielson-Lancos lemma shows that a sequence must be divided up into its odd and even subsets. That these subsets must in-turn be divided into their subsets. This continues until we have only two members per subset. An illustration of this subdivision, for $N=8$, is shown in Figure 6.1. This is called decimation in time.

Figure 6.1 Decimation in time.

It is natural to implement the decimation in time using recursive calls with odd and even sets. It has been shown, however, that a recursive implementation is six times slower than a non-recursive implementation [Gonzalez et al.]. To avoid a recursive approach to decimation, a bit-reversal algorithm is used to perform the decimation in time. This is called the Cooley-Tukey algorithm.

The Cooley-Tukey FFT algorithm performs decimation in time by using a bit-reversal, this is shown in Figure 6.2.

Figure 6.2. An Example of how to decimate by bit reversal

To arrive at the bit reversal, we implement a Java method in the FFT class:

```
int bitr(int j) {
    int ans = 0;
    for (int i = 0; i < nu; i++) {
        ans = (ans << 1) + (j & 1);
        j = j >> 1;
    }
    return ans;
}
```

The bitr method works by linking together two software shift-registers, as shown in Figure 6.3.

Figure 6.3. The j and a registers are linked with the + operator.

After the decimation in time is performed, the balance of the computation is optimization hacks and house-keeping. For example, a simplification results from V_k being periodic in N so that $V_{k+N} = V_k$.

Proof:

Recall that the DFT is given by:

$$V_k = \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi i j k / N} v_j \quad (6.4)$$

so that

$$V_{k+N} = \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi j(k+N)/N} v_j$$

expanding the exponents and simplifying using

$$V_{k+N} = \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi jk/N} e^{-2\pi jN/N} v_j$$

with $e^{-2\pi j} = \cos(-2\pi j) + i \sin(-2\pi j) = 1$. Thus

$$V_{k+N} = \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi jk/N} v_j$$

with

$$V_{k+N} = V_k \quad (6.18)$$

Q.E.D.

In addition, it can be shown that

$$W^{k+N/2} = -W^k \quad 0 \leq k \leq N/2 \quad (6.19)$$

Proof:

Recall (6.13) is given by

$$W = e^{-2\pi i/N} \quad (6.13)$$

So that

$$e^{-2\pi i(k+N/2)/N} = \cos(-2\pi(k+N/2)/N) + i \sin(-2\pi(k+N/2)/N)$$

with

$$\begin{aligned} \cos(-2\pi(k+N/2)/N) &= \cos(2\pi k/N + \pi) = -\cos(2\pi k/N) \\ \sin(-2\pi(k+N/2)/N) &= \sin(2\pi k/N + \pi) = -\sin(2\pi k/N) \end{aligned} \quad (6.20)$$

this naturally leads to:

$$W^{k+N/2} = -W^k \quad 0 \leq k \leq N/2$$

Q.E.D.

A further efficiency may be had by the use of the recurrence relation

$$W^{jk} = W^j W^{j(k-1)} \quad (6.21).$$

Proof:

$$\begin{aligned} W^{jk} &= e^{-2\pi jk/N} = \cos(-2\pi jk/N) + i \sin(-2\pi jk/N) \\ W^{jk} &= \cos(-2\pi jk/N) + i \sin(-2\pi jk/N) \\ W^{jk} &= [\cos(-2\pi j/N) + i \sin(-2\pi j/N)] \\ &\quad * [\cos(-2\pi j(k-1)/N) + i \sin(-2\pi j(k-1)/N)] \\ W^{jk} &= W^j W^{j(k-1)} \end{aligned} \quad (6.22)$$

Q.E.D.

The real and imaginary parts of (6.22) are given by

$$\text{real}(z_1 z_2) = x_1 x_2 - y_1 y_2 \quad (6.11b)$$

so that

$$W_r^{jk} = W_r^j W_r^{j(k-1)} - W_i^j W_i^{j(k-1)} \quad (6.23)$$

and the imaginary part of (6.22) is given by:

$$\text{imaginary}(z_1 z_2) = x_1 y_2 + y_1 x_2 \quad (6.11c).$$

so that

$$W_i^{jk} = W_r^j W_i^{j(k-1)} + W_i^j W_r^{j(k-1)} \quad (6.24).$$

Equations (6.23) and (6.24) form the basis of the recurrence relationships that enables the quick computation of the next W^{jk} based on the previous W^{jk} . An implementation of (6.24) follows:

```

1.          // (eq 6.23) and (eq 6.24)
2.          wtemp = Wjk_r;
3.          Wjk_r = Wj_r * Wjk_r - Wj_i * Wjk_i;
4.          Wjk_i = Wj_r * Wjk_i + Wj_i * wtemp;

```

Line 2 shows the introduction of wtemp, a temporary variable that enables the computation of the multiplication of the two complex numbers.

(A-heading) The FFT Class

The FFT class is a public class that resides in the lyon.audio package. It depends on the grapher.Graph package for performing graph function, as well as the futils.bench.Timer class, for performing the benchmarking functions. If these packages are not needed, they may be removed, along with their invocations. Such a situation might arise if this code were to be used in another program.

The grapher package provides a simple interface to make an automatically scaled graph. Generally only a single method is invoked. This is best shown by the following example:

```

public void makeHanning() {
    double window[];
        window = makeHanning(256);
        Graph.graph(window,
            "The Hanning window", "f");
}

```

Where the “The Hanning window” string appears along the x-axis and “f” appears on the y-axis. The Graph.graph may be invoked directly because the graph method is static. Also, it only graphs an array of type double.

(B-heading) Class Summary

```

package lyon.audio;
import java.io.*;
import java.awt.*;
import grapher.Graph;
import futils.bench.Timer;
public class FFT extends Frame {
    public FFT(int N)
    public FFT()
    public void graphs()
    public void graphs(String t)
    public void setTitle(String t)
    public static double getMaxValue(double in[])
    public static int log2(int n)
    public static double[] arrayCopy( double [] in)
    public double [] computePSD ()
}

```

```

    public double[] dft(double v[])
    public double[] idft()
    public double [] getReal()
    public double [] getImaginary()
    public void forwardFFT(double in_r[], double in_i[])
    public void reverseFFT(double in_r[], double in_i[])
    public void printArray(double[] v,String title)
    public void printArrays(String title)
    public void printReal(String title)
    public static void main(String args[])
    public static void timeFFT()
    public static void testFFT()
    public static void testDFT()
}

```

(B-heading) Class Usage

The FFT class maintains internal data arrays that are stored as doubles. These arrays are private and are used to assist computations. Further, the in-place Cooley-Tukey algorithm employed for the fast transform is destructive for the original data. The FFT class in the lyon.audio package uses doubles for all computations. This class is for 1-D (audio) transforms.

Suppose the following variables are predefined:

```

FFT f;
int N = 8;
double inputArray[];
String title = "My data title";
double aDoubleArray[];
double in_r[];
double in_i[];

```

To make a new instance of the FFT class, and allocate two internal arrays of double, each of length N:

```
f = new FFT(N);
```

To make a new instance of the FFT class, with no memory allocation:

```
f = new FFT();
```

To graph the real and imaginary data arrays:

```
f.graphs();
```

To graph the real and imaginary data arrays with a title:

```
f.graphs(title);
```

To set the title for the graphs:

```
f.setTitle(title);
```

To get the maximum value of an inputArray:

```
FFT.getMaxValue(inputArray);
```

To compute the floor of the log of an int to base 2:

```
int numberOfBits = FFT.log2(N);
```

To copy an array of double:

```
aDoubleArray = FFT.arrayCopy(inputArray);
```

To compute the psd (power spectral density) of the last dft or fft:

```
aDoubleArray = f.computePSD();
```


To non-destructively compute the dft of an input array and return the psd:

```
aDoubleArray = f.dft(inputArray);
```

(BEGIN NOTE) DFT, IDFT, FFT and IFFT alter the internal data structures in an instance of the FFT class.(END NOTE)

To get the real part of the last transform:

```
aDoubleArray = f.getReal();
```

To get the imaginary part of the last transform:

```
aDoubleArray = f.getImaginary();
```

To take the idft of the internal data and return the real part:

```
aDoubleArray = f.idft();
```

To take the forward fft on two input arrays, destructively:

```
f.forwardFFT(in_r, in_i);
```

To take the inverse FFT on two input arrays, destructively

```
f.reverseFFT(in_r, in_i);
```

To print an array of double, with a title:

```
f.printArray(aDoubleArray, title);
```

To print the internal real and imaginary arrays, with a title:

```
f.printArrays(title);
```

To print the internal real array, with a title:

```
f.printReal(title);
```

To test the DFT, IDFT, FFT and IFFT:

```
FFT.main();
```

To time the FFT:

```
FFT.timeFFT();
```

To test the FFT:

```
FFT.testFFT();
```

To test the DFT:

```
FFT.testDFT();
```

(B-heading) Testing the FFT and IFFT

The FFT class has a static method that permits the testing of the DFT, IDFT, FFT and IFFT. It also performs timing for a transform of 2048 doubles. To run this test, you must invoke

```
FFT.main();
```

The code for the FFT.main method follows:

```
public static void main(String args[]) {
    testDFT();
    timeFFT();
    testFFT();
}
```

The test methods are run on an 8 point input array consisting of a linear ramp. This is to provide a short sequence of input data that can be verified by printing. The timing is performed on 2048 samples stored in two arrays of 2048 doubles each (real and imaginary). The output of the main method follows:

```
Executing DFT on 8 points...
Executing IDFT on 8 points...
j   x1[j]   re[j]   im[j]   v[j]
0   0       3.5     0      -3.10862e-15
1   1      -0.5    1.20711 1
```

```

2      2      -0.5      0.5      2.00000
3      3      -0.5      0.207107      3
4      4      -0.5      0      4
5      5      -0.500000 -0.207107      5
6      6      -0.500000 -0.5      6
7      7      -0.5      -1.20711      7

```

```
fft: bit reversal
```

```
Time for 2048point fftTime 0.178000 sec
```

```
fft: bit reversal
```

```
Time for 2048point ifftTime 0.164000 sec
```

```
Starting 1D FFT test...
```

```
fft: bit reversal
```

```
fft: bit reversal
```

```

j      x1[j]      re[j]      im[j]      v[j]
0      0      3.5      0      0
1      1      -0.5      1.20711      1.00000
2      2      -0.5      0.5      2.00000
3      3      -0.500000 0.207107      3.00000
4      4      -0.5      0      4
5      5      -0.5      -0.207107      5
6      6      -0.5      -0.5      6
7      7      -0.500000 -1.20711      7

```

The reader will see that the input and output are highly correlated for both the DFT and FFT. The surprising thing is how accurate these two radically different algorithms and implementations are. Also, recall that the execution times for the DFT was benchmarked at 55 seconds. The FFT implementation is run in 0.178 seconds, a 308 times speed up. Keep in mind, at 8000 samples per second, the 2048 samples represent 0.256 seconds of data. Also, on a limited data rate connection (such as a 28.8 kbps modem) the time to transmit the data is $2048 * 8 \text{ bits} / 28800 \text{ bits/sec} = 0.56 \text{ seconds}$. We suggest that many dial-up users experience a slower connection than the maximum their modem permits. Thus, there is a window of opportunity for devising a real-time codec (IN JAVA!!) able to perform FFT based compression algorithms. An algorithm based on transform compress typically takes the original data, performs the forward transform, selects coefficients, quantizes and then transmits. Data is recovered by taking the coefficients and performing an inverse transform. Very Low Bit Rate Voice Compression (VLBRVC) is a rich and growing field that lies beyond the scope of this book. See http://www.bdti.com/faq/dsp_faq.htm for an FAQ that relates to this and other DSP topics.

(B-heading) Implementing the FFT.testFFT

The following code shows how to use the FFT class to perform a forward and inverse FFT. The static nature of the testFFT method indicates that invocation may be performed without making an instance of the FFT class.

Line 3 makes an instance of the FFT class, without performing any allocation for the internal data structures. Thus the allocation and copying of arrays is performed outside of the forwardFFT methods. This is due, in part, to the destructive nature of the in-place Cooley-Tukey FFT algorithm. The trade-off is that the programmer must keep track of the data that is being processed by the forwardFFT. The alternative is to automatically copy arrays, perform the in-place forwardFFT, then return the copies. Our findings

indicate that the dynamic allocation of memory (particularly during the image processing, seen later in this book) can slow performance by up to 100 times! Thus, the house keeping chores performed by the programmer are warranted by a leap in performance.

```

1.  public static void testFFT()  {
2.  System.out.println("Starting 1D FFT test...");
3.  FFT f = new FFT();

```

Line 4 may be altered to any number of samples, N, but a large N will result in a large printout.

```

4.  int N = 8;
5.  int numBits = f.log2(N);

```

Lines 6-8 set up the input data to be a ramp that varies from 0 to N.

```

6.  double x1[] = new double[N];
7.  for (int j=0; j<N; j++)
8.  x1[j] = j;

```

Now the housekeeping. The programmer, interested in keeping copies of the original data, the result of the forward FFT and the result of the inverse FFT, must allocate four arrays! This is an unusual case, as it requires that all intermediate results be kept for checking purposes. Normally, production code would not have to keep all intermediate results.

```

9.  double[] in_r = new double[N];
10. double[] in_i = new double[N];

```

The `in_r` and `in_i` arrays are copies of the input data, with the imaginary component equal to zero. Real data (like audio data) often has a zero imaginary component. There are algorithms that can save significant time by taking advantage of the zero imaginary part of the input data. This requires a different FFT implementation.

```

11. double[] fftResult_r = new double[N];
12. double[] fftResult_i = new double[N];

```

```

13. // copy test signal.
14. in_r = arrayCopy(x1);

```

Line 14 copies the input data into `in_r`.

```

15. f.forwardFFT(in_r, in_i);

```

Line 15 replaces `in_r` and `in_i` with the forward FFT results.

```

16. // Copy to new array because IFFT will
17. // destroy the FFT results.
18. fftResult_r = arrayCopy(in_r);
19. fftResult_i = arrayCopy(in_i);
20. f.reverseFFT(in_r, in_i);
21. System.out.println("j\tx1[j]\tre[j]\tim[j]\tv[j]");
22. for(int i=0; i<N; i++) {
23.     System.out.println(
24.         i + "\t" +
25.         x1[i] + "\t" +
26.         fftResult_r[i] + "\t" +
27.         fftResult_i[i] + "\t" +
28.         in_r[i]);
29. }

```

30. }
(A-heading) PSD Computations

To compute the psd of the FFT output, we use the computePSD method in the FFT class. The psd is computed by squaring the real and imaginary parts of the output of the FFT and placing them into a new array. The code for the computePSD method follows:

```
public double [] computePSD () {
    double [] psd = new double[r_data.length];
    for (int k = 0; k < r_data.length; k++) {
        psd[k] =
            r_data[k] * r_data[k] +
            i_data[k] * i_data[k];
    }
    return psd;
}
```

To test the psd, there is a private method in the FFT class called testPSD

```
private static void testPSD() {
    FFT f = new FFT();
    int N = 8;
    int numBits = f.log2(N);
    double x1[] = new double[N];
    for (int j=0; j<N; j++)
        x1[j] = j;
    double[] in_r = new double[N];
    double[] in_i = new double[N];
    // copy test signal.
    in_r = arrayCopy(x1);
    f.forwardFFT(in_r, in_i);
    f.printArrays("After the FFT");
    double psd[] = f.computePSD();
    FFT.printArray(psd, "The psd");
}
```

The output of the test method appears below:

```
After the FFT
[0]=(3.5,0)
[1]=(-0.5,1.20711)
[2]=(-0.5,0.5)
[3]=(-0.5,0.207107)
[4]=(-0.5,0)
[5]=(-0.500000,-0.207107)
[6]=(-0.500000,-0.5)
[7]=(-0.500000,-1.20711)
The psd
v[0]=12.25
v[1]=1.70711
v[2]=0.5
```

```

v[3]=0.292893
v[4]=0.25
v[5]=0.292893
v[6]=0.500000
v[7]=1.70711
Completed(0)

```

The reader can readily verify that the psd is indeed the sum of the squares of the real and the imaginary parts of the spectra.

Figure 6.4. The psd of a 2048 Sampled Waveform

The implementation of the psd computation and the graphing are shown in the following section.

(B-heading) Implementation of the Transforms in the AudioFrame

In the AudioFrame class, we assume that we will be taking the FFT of a real signal. A real signal, like audio, has no imaginary part, only a single value that varies from sample to sample. Thus, we construct a complex input to the FFT, setting the imaginary part of the input equal to zero.

When taking the IFFT, a signal that starts as being real will, end up as a real signal. The exception to this occurs when a spectral modification introduces terms that do not null out in the imaginary plane. This could happen, for example, if we add the spectra from real and imaginary signals.

Figure 6.5 shows the part of the Audio menu in the MainMenuBar in the AudioFrame that contains the transform fragments for performing the FFT manipulations. The keyboard shortcuts are shown in brackets (i.e., '[1]').

Figure 6.5. The Transform Fragment of the AudioFrame

The menu items are initialized at the head of the AudioFrame class using the following code fragment:

```

MenuItem fft_mi      = addItem(m, "[1] FFT" );
MenuItem ifft_mi     = addItem(m, "[2] IFFT" );
MenuItem dft_mi      = addItem(m, "[3] DFT" );
MenuItem idft_mi     = addItem(m, "[4] IDFT" );
MenuItem graphPSD_mi = addItem(m, "[5] Graph PSD, R and
I" );

```

The event handling is performed in the handleEvent method, using the Evt class to detect the keyboard shortcuts. This enables rapid processing by the user:

```

if (Evt.match(e,fft_mi)) {
    fft();
    return true;
}
if (Evt.match(e,ifft_mi)) {
    ifft();
    return true;
}

if (Evt.match(e, dft_mi)) {
    fftInstance = dft();
    return true;
}

```

```

    }
    if (Evt.match(e, idft_mi)) {
        idft(fftInstance);
        return true;
    }

    if (Evt.match(e, graphPSD_mi)) {
        graphPSD();
        return true;
    }
}

```

A class variable, called the `fftInstance` is used to pass the complex data results from method to method. The `fftInstance` is declared right after the `Oscillator` and `UlawCodec` instances

```

public class AudioFrame extends OscopeFrame {

    private Oscillator osc =
        new Oscillator(400,4000);

    UlawCodec ulc;
    FFT fftInstance;
}

```

In the following DFT and IDFT code, we include facilities for benchmarking the execution of the DFT on long sequences. This is because we find the DFT and IDFT to be so slow as to be of interest only for timing purposes. Also, as a twist on the implementation possibilities, we show the `dft` method returning an instance of the `FFT` class. This is used to replace the `fftInstance` class variable in the `handleEvent` method.

```

    public FFT dft() {
        double doubleArray[] = ulc.getDoubleArray();
        double [] psd;
        FFT f = new FFT();
        Timer t1 = new Timer();
        t1.mark();
        psd=f.dft( doubleArray);
        // Stop the timer and report.
        t1.record();
        System.out.println("Time to perform DFT:");
        t1.report();

        f.graphs();
        return f;
    }
}

```

The `handleEvent` passes the `fftInstance` to the `idft` method for processing. Thus, this method is overwriting the `fftInstance` variable (since it is passed, like all instances, by reference). The IDFT method follows:

```

    public void idft(FFT f) {
        double doubleArray[];
        Timer t1 = new Timer();
        t1.mark();
    }
}

```

```

        doubleArray=f.idft( );
        // Stop the timer and report.
        t1.record();
        System.out.println("Time to perform IDFT:");
        t1.report();
        f.graphs();
    }

```

In the following fft method, the fftInstance is overwritten. Data is obtained from a getTruncatedDoubleData method that truncates the input data to the nearest integral power of two. We have elected to perform this truncation explicitly, as some users might prefer to pad their input data with extra samples. Consider, if the input data consisted of 2049 samples, such a practice would pad the input data to 4096 samples. This is a doubling of the processing time. As the loss of data may well be objectionable to some, we have elected to perform the truncation, or padding, outside of the FFT class. The programmer must be careful (BEGIN WARNING) if the input to the FFT is not an integral power of two, the forwardFFT and reverseFFT method will produce WRONG results as no check is performed! (END WARNING)

```

    public void fft() {

        fftInstance = new FFT();
        double[] r_d = getTruncatedDoubleData();
        double[] i_d = new double[r_d.length];
        fftInstance.forwardFFT(r_d, i_d);
        ulc.play();
    }

```

(BEGIN NOTE) The UlawCodec instance, ulc, is played after the fft. (END NOTE) The audio data AudioFrame class is not shortened or altered by the fft invocation. The play simply signals when the fft is complete.

The audio data is copied before truncation occurs. It is not overwritten until after the ifft method is invoked. The getTruncatedDoubleData truncation method follows:

```

    double [] getTruncatedDoubleData() {
        double[] temp = FFT.arrayCopy(getDoubleData());

        int trunc = 1 << FFT.log2(temp.length);
        System.out.println("Truncated size: " + trunc);
        double[] truncArray = new double[trunc];

        for (int i=0; i < trunc; i++)
            truncArray[i] = temp[i];

        return truncArray;
    }

```

Performing an FFT is sometimes called analysis. After the analysis, there may be a frequency domain based pattern recognition (i.e., what note was played) or a spectral modification (i.e., filtering, pitch shifting, etc.). The IFFT is sometime called synthesis

because the time-domain waveform is resynthesized from the frequency domain description. The implementation of the `ifft` method in the `AudioFrame` follows:

```
public void ifft() {
    double realInput[] =
        FFT.arrayCopy(fftInstance.getReal());
    double imaginaryInput[] =
        FFT.arrayCopy(fftInstance.getImaginary());
    fftInstance.reverseFFT(realInput, imaginaryInput);
    ulc = new UlawCodec(realInput);
    ulc.play();
}
```

(BEGIN NOTE) The `ifft` method throws away the imaginary part of the `reverseFFT` methods output. For real input signals, the imaginary part is zero, in theory. We have found, however, that there is some small near-zero imaginary part that has been attributed to round-off error. (END NOTE)

The graph of the `psd` is performed using `graphPSD`

```
public void graphPSD() {
    Graph.graph(fftInstance.computePSD(),
        "PSD of truncated waveform", "a^2");
};
```

For example, suppose that a saw wave is used as an input.

The sawwave and `psd` are shown using the `OscopeFrame` and `graph` methods on the left and right in Figure 6.6.

Figure 6.6. SawWave and Spectral output from the `graphPSD` method.

(BEGIN NOTE) Real signals always have a spectra that is symmetric about its center. (END NOTE)

(B-heading) A Noise filter using the FFT

The basic idea of providing a noise filter is that you take a signal, with added noise, perform an FFT on the signal, remove all spectral harmonics that have a `psd` below some threshold, then take the IFFT. Selecting the `psd` threshold for noise can be tricky. What works well on a synthetic sound might turn a sampled sound into silence. There are limits to the amount of noise that can be filtered out. A block diagram of the process appears in Figure 6.7. The code for adding noise to the waveform stored in the `AudioFrame` instance is shown below:

```
public void addNoise() {
    double r_d[] = getDoubleData();
    for (int i = 0; i < r_d.length; i++)
        r_d[i] = r_d[i] + 0.1*(Math.random() - 0.5);
    ulc = new UlawCodec(r_d);
}
```

Figure 6.7. The Noise Filter

The `Math.random` method returns a random value between zero and one. Thus, the sampled data is summed with time-domain uniformly distributed noise (also known as white noise) that varies from -0.10 to 0.10. The following code performs a `psd` based cutoff, after taking the FFT of the sound samples:

```
public void removeNoise() {
```



```

double noisePowerCutoff = 0.05;

fftInstance = new FFT();
double r_d[] = getTruncatedDoubleData();
double i_d[] = new double[r_d.length];
fftInstance.forwardFFT(r_d, i_d);
double psd[] = fftInstance.computePSD();
for (int i = 0; i < psd.length; i++) {
    if (psd[i] < noisePowerCutoff) {
        r_d[i] = 0;
        i_d[i] = 0;
    }
}
fftInstance.reverseFFT(r_d, i_d);
ulc = new UlawCodec(r_d);
ulc.play();
}

```

The `removeNoise` and `addNoise` methods are placed under interactive user control via the main-menu bar entries:

```

MenuItem addNoise_mi      = addMenuItem(m, "[A] Add Noise");
MenuItem removeNoise_mi  = addMenuItem(m, "[R] Remove
Noise");

```

The initial waveform is a sine wave of 400 Hz. A graph of the sinewave with psd is shown in Figure 6.8.

Figure 6.8. Graph of the Sine Tone at 400 Hz with psd

Figure 6.9 shows the sinewave after noise is added.

Figure 6.9. Sinewave after the addition of noise

The psd of the sinewave plus noise is shown in Figure 6.10.

Figure 6.10. The psd of the Sinewave plus Noise

(BEGIN NOTE) Figure 6.10 shows the clean spectral break between the noise and the sinewave. The removal of noise from such a waveform is performed with a trivial frequency-based amplitude detector. (END NOTE)

Figure 6.11. The Reconstructed Waveform and its psd

(A-heading) Spectral Leakage of the DFT

Recall that in equation (6.6)

$$f_k = \frac{k}{N\Delta t} \quad (6.6),$$

we may compute the frequency of the k th sample of a spectrum. For example, when the number of samples, $N = 2048$, the smallest change in frequency is $8000/2048 = 3.9$ Hz. For the spectrum shown in Figure 6.5, the central point is given by $N/2 = 2048/2 = 1024$. By (6.6), therefore, we compute the highest frequency that may be represented by 2048

samples (with 8kHz sampling rate) as $(8000/2048) * 1024 = 4000$ Hz. Also, we may solve for any sample using (6.25):

$$k = Nf_k / f_s \quad (6.25)$$

Thus, for the 400 Hz sawwave spectrum shown in Figure 6.8, we expect the maximum amplitude to occur at $k = Nf_k / f_s = 2048 * 400 / 8000 = 102.4$. How can the FFT (or DFT, for that matter) represent energy at $k = 102.4$? The answer is that the energy is spread around $k=102$, and 103. Often the k th array element in the frequency domain is referred to as a frequency *bin*. Since the 400 Hz tone is in both bins 102 and 103, the phenomenon is called *spectral leakage*. The problem with spectral leakage is that it is different for different frequencies. For example, at 390.625 Hz, $k = 2048 * 390.625 / 8000 = 100$, *exactly*. Thus, there is no spectral leakage for some frequencies, where there may be a great deal of spectral leakage for others. This is the frequency-domain rationale for spectral leakage.

The time-domain rationale for spectral leakage is that, for a waveform that is not periodic in time, the temporal effect of the sample window becomes visible in the Fourier transform [Walker].

A procedure known as *windowing* predistorts the input samples so that the spectral leakage is evened out (spreading on-bin signals more and off-bin signals less).

Windowing is a common procedure. It is typically performed with one of a variety of possible filters (Cesáro, hanning, Hamming, Parzen, Welch, etc.). It is beyond the scope of this book to cover all the filters. The hanning filter is a popular windowing filter, and is applied in the sample domain by a complex multiplication. In the case when the input signal is real (as in audio) only a single multiplication is required. The equation for the hanning filter is:

$$w_j = \frac{1}{2} \left[1 - \cos \left(\frac{2\pi j}{N-1} \right) \right] \quad j \in [0 \dots N-1] \quad (6.27)$$

Figure 6.12 shows a 256 sample version of the hanning window, in the frequency domain, generated by the DiffCAD program.

Figure 6.12. The hanning window

The main point of windowing is that it reduces the amplitude of the samples at the beginning and end of the window. The following code, from the AudioFrame class, will make and graph the hanning window:

```
public void makeHanning() {
    double window[];
    window = makeHanning(256);
    Graph.graph(window,
        "The Hanning window", "f");
}
```

The method returns an array of doubles that may be stored for later use. This permits reuse of the window for production code.

```
public double[] makeHanning(int n) {
    double window[] = new double[n];

    double arg = 2.0 * Math.PI / (n - 1);
```

```

    for (int i = 0; i < n; i++)
        window[i] = 0.5 - 0.5 * Math.cos(arg*i);

    return window;
}

```

To apply the window to the samples, we have devised a `windowArray` method in the `AudioFrame` class:

```

public void multHanning() {
    double[] r_d = getTruncatedDoubleData();
    double[] window = makeHanning(r_d.length);

    windowAudio(r_d, window, "hanning");
}

public void windowArray(double window[], double r_d[]) {
    for (int i = 0; i < window.length; i++) {
        r_d[i] *= window[i];
    }
}

```

The primary difference between one window and the next is the way it tapers off at the ends of the samples. One window, called the Bartlett window, is a linear window that is described by:

$$w_j = \begin{cases} \frac{2j}{N-1} & j \in \left[0 \dots \frac{N-1}{2}\right] \\ 2 - \frac{2j}{N-1} & j \in \left[\frac{N-1}{2} \dots N-1\right] \end{cases} \quad (6.28)$$

Figure 6.13. The Bartlett Window

Figure 6.13 shows a graph of the Bartlett. The Bartlett window is generated by the method `makeBartlett`, shown below:

```

public double[] makeBartlett(int n) {
    double window[] = new double[n];
    double a = 2.0 / (n - 1);
    for (int i = 0; i < n/2; i++)
        window[i] = i * a;
    for (int i = n/2; i < n; i++)
        window[i] = 2.0 - i * a;
    return window;
}

```

We propose a window that is zero at the sample end-points and also has zero first and second derivatives at both the end-points and the center. Such a window may be formulated with two fifth-order polynomials (called quintics). We present, for your interest, some Maple code for deriving a quintic-based window.

Maple is a symbolic manipulator that can help with some math problems. In Maple, we let `dy` and `ddy` be the first and second derivatives of the polynomial with respect to the

dilation parameter u . The dilation parameter will be varied between zero and one, inclusive. The amplitude of the quintic will vary from y_0 to y_1 . Maple's optimize routine can output a procedure for computing quintics in minimum CPU time. What follows is a Maple procedure for generating the Lyon window. See [Lyon 91] for more information and an application of the quintic to maneuvering.

```
restart;y:=a5*u^5+a4*u^4+a3*u^3+a2*u^2+a1*u+a0:
> dy:=diff(y,u):
> ddy:=diff(dy,u):
> a0:=y0:
> a1:=solve(y1=subs(u=1,y),a1):
> a2:=solve(0=subs(u=0,dy),a2):
> a3:=solve(0=subs(u=1,dy),a3):
> a4:=solve(0=subs(u=0,ddy),a4):
> a5:=solve(0=subs(u=1,ddy),a5):
> readlib(C):
> C(y,optimized);
      t2 = u*u;
      t3 = t2*t2;
      t11 = (6.0*y1-6.0*y0)*t3*u+(-
15.0*y1+15.0*y0)*t3+(10.0*y1-10.0*y0)*t2*u+y0;
```

The following code makes the Lyon window

```
public double y5(double y0, double y1, double u) {
    double t2 = u*u;
    double t3 = t2*t2;
    return
        (6 * y1 - 6 * y0) * t3 * u +
        (-15 * y1 + 15 * y0) * t3 +
        (10 * y1 - 10 * y0) * t2 * u + y0;
}
```

$$w_j = \begin{cases} 6u^5 - 15u^4 + 10u^3 & u = \frac{2j}{N-1}, j \in \left[0 \dots \frac{N-1}{2}\right] \\ -6u^5 + 15u^4 - 10u^3 + 1 & u = \frac{2j+1-N}{N-1}, j \in \left[\frac{N-1}{2} \dots N-1\right] \end{cases} \quad (6.29)$$

The easier way to formulate (and compute) the dilation parameter, u , is by incrementing it by $2/N$ as in:

```
public double[] makeLyon(int n) {
    double window[] = new double[n];

    double u = 0.0;
    double du = 2.0/n;
    for (int i = 0; i < n/2; i++) {
        window[i] = y5(0,1.0,u);
        u += du;
    }
    u=0;
```

```

    for (int i = n/2; i < n; i++) {
        window[i] = y5(1.0,0.0,u);
        u += du;
    }
    return window;
}

```

Figure 6.14. The Lyon and hanning windows compared

The Lyon and hanning windows are shown in Figure 6.14. (BEGIN NOTE) More study is needed to elaborate on the design trade-offs between the Lyon window and other popular windows. About the only remarkable thing about it is that it has a zero first and second derivatives at the center and end-points. The Lyon window also has more taper at the side-lobes than the hanning window and so may reduce aliasing error more. (END NOTE)

(A-heading) The Hi-pass filter

One very simple way to design a hi-pass filter is to take an FFT on the input samples, then multiply the spectrum by the filter envelope. For example, to make a hi-pass filter, we may zero out the low frequencies using a rectangular pass-band function, like the one shown in Figure 6.15.

Figure 6.15. A passband shown spectral harmonics to admit

Figure 6.16. The Sawwave and its psd

Figure 6.17. The Hi-pass filtered Sawwave and its psd

(Begin Note) For the psd depicted in this chapter, the higher frequencies are toward the center of the graph. (end Note) For the determination of the performance of the various windows, we follow [Embree] and plot the windowed waveform along side the log (in dB) of the psd. We use the dB log and zoom into the first 200 samples of the psd to make smaller details clear. We are computing the graph using:

```

public void graphPSD() {
    double psd[] = fftInstance.computePSD();
    double shortPsd[] = new double[200];
    for (int i =0 ; i < shortPsd.length; i++) {
        shortPsd[i] = 10*Math.log(Math.sqrt(psd[i]));
    }
    Graph.graph(shortPsd, "spectral mag", "a dB");
}

```

Figure 6.18. A Rectangular window and spectral dB log.

Figure 6.18 shows a sinewave with a rectangular window. The sine wave is 400 Hz with 8000 Hz sampling. This is often expressed as a relative frequency of $400/8000 = 0.05$.

Figure 6.19 shown the hanning window with the spectral log magnitude.

Figure 6.19. The hanning windowed data with the FFT result.

The triangular windowed input of the Bartlett window is shown in Figure 6.20, along with the psd.

Figure 6.20. The Bartlett windowed data and the psd

Figure 6.21. The Lyon window and psd

Figures 6.20 and 6.21 show that the Lyon window has somewhat better side lobe performance than the Bartlett window (lower spectral leakage). Also that the main lobe in the Lyon window is narrower than the Bartlett window (lower noise bandwidth). Figure 6.22 shows the spectra for the Lyon and hanning windows on the left and right, for comparison.

Figure 6.22. Spectra of Lyon vs. hanning windows

The Lyon and hanning windows appear to be very close spectrally. The Lyon window appears to have a few dB better side lobe performance over the hanning window. The main lobe is slightly wider, however, and so has a slightly larger equivalent noise bandwidth. For more information on windowing see [Mitra].

(A-heading) Frequency shifting using the FFT

To shift the pitch of a time-domain signal, we take the FFT, perform the high-pass filtering, shown in the previous section, shift the spectrum lower, and then perform the IFFT. Recall that the FFT produces a real and a complex output. Also recall that the `fftInstance` is a class variable in the `AudioFrame` class. Thus, we design our pitch shifter as just one of many possible spectral modifications that may be performed by the user before the IFFT is taken. The plan is to work on bins $0..N/2$ first, then to copy the bins about the $N/2$ point in the spectrum, assuming that the left and right-hand sides are symmetric (as is always the case for real signals). The code for the pitch shift follows:

```
public void pitchShift() {
    fftInstance = new FFT();
    double[] r_d = getTruncatedDoubleData();
    int N = r_d.length;
    double[] i_d = new double[N];
    int N_on_4 = N/4;

    fftInstance.forwardFFT(r_d, i_d);
    // shift data down
    for (int i = 0; i < N_on_4; i++) {
        r_d[i] = r_d[i + N_on_4];
        i_d[i] = i_d[i + N_on_4];
    }
    for (int i = N_on_4; i < N/2; i++) {
        r_d[i] = 0;
        i_d[i] = 0;
    }

    // reflect about center, assuming a real signal
    int i, j;
```

```

for (i=0, j=N-1; i < N/2; i++, j--) {
    r_d[j] = r_d[i];
    i_d[j] = i_d[i];
}

fftInstance.reverseFFT(r_d, i_d);

ulc = new UlawCodec(r_d);

ulc.play();

}

```

The result for synthetic tones, rich in harmonics, is to filter out some of the lower frequencies and to lower the upper harmonic content.

Figure 6.23. The Squarewave and its psd

Figure 6.24. The pitch-shifted square wave and its psd.

(A-heading) Resampling and the FFT

Resampling a 1-D waveform is a common way to perform time-compressed speech. One very easy way to perform the resampling is to perform Fairbanks sampling and throw away every other sample [Fairbanks].

The resampling method performs a 2:1 subsampling in the time domain. The code for the resample method follows:

```

public void resample() {
    double[] r_d = getTruncatedDoubleData();
    int N = r_d.length;
    double [] resampled = new double[N/2];
    for (int i=0; i < N/2; i++)
        resampled[i] = r_d[2*i];

    ulc = new UlawCodec(resampled);

    ulc.play();

}

```

Figure 6.25 shows a sawwave and its psd. Compare Figure 6.25 with 6.26 that shows the sawwave and psd after the 2:1 subsampling algorithm has been applied.

Figure 6.25. The sawwave and psd before the subsampling

Figure 6.26. The sawwave and psd after subsampling

From figures 6.26 and 6.25 we can clearly see that the 2:1 subsampling has doubled the pitch of the harmonics, and halved the number of available samples.

(A-Heading) Centering the FFT

There are a great many books that show how the lowest frequency is toward the center of the psd when taking an FFT. This is due to the process of centering the FFT. Centering the FFT is accomplished by replacing the sample data with a value that is changing from positive to negative when the sample value goes from zero to one. This may be described by the formula:

$$v_k = v_k(-1)^k \quad (6.30)$$

For the real sample on input. After the IFFT (or IDFT) the formula must be applied again [Myler].

Figure 6.27 A Pulse with A Centered psd.

The centered psd for a pulse is shown in Figure 6.27. One can modify the forwardFFT in the AudioFrame to perform this method by using:

```
public void forwardFFT(double in_r[], double in_i[]) {
    int id;

    int localN;
    double wtemp, Wjk_r, Wjk_i, Wj_r, Wj_i;
    double theta, tempr, tempi;
    int ti, tj;

    int numBits = log2(in_r.length);
    if (forward) {
        centering(in_r);
    }
}
```

Where centering is a method that computes equation (6.28) on the real part of the sample data. Centering may be implemented as follows (in the FFT class):

```
private void centering(double r[] ) {
    int s = 1;
    for (int i = 0; i < r.length; i++) {
        s = -s;
        r[i] *= s;
    }
}
```

The reverseFFT implements centering after the IFFT is finishing up:

```
public void reverseFFT( double in_r[], double in_i[]) {
    forward = false;
    forwardFFT(in_r, in_i);
    forward = true;
    centering(in_r);
}
```

Figure 6.28. An Uncentered psd

Without centering the psd is shown with the lowest frequencies on the edges. This is the convention that we have adopted for the 1-D psd display (except when explicitly marked

otherwise). We are amazed that so few books speak about the uncentered magnitude Fourier spectrum and how to correct it. Left uncentered, the FFT and DFT produces results that can match the outputs of other FFT and DFT implementations (like [Moore]). As a result, we felt it best to leave the spectrum uncentered, at least for the 1-D FFT.

(A-heading) Summary

The close of this chapter is a sad time! Just as things were getting fun, we move on to the image processing section of the book (which will, hopefully, be even more fun!!). The idea that a pitch shifter be combined with a resampler to compress speech is not new. In fact, it may be used to help perform skimming on recorded speech (a topic of current research) [Arons].

The introduction to the DFT, IDFT, FFT and IFFT is not new either. Also, it is probably the case that the FFT is not the fastest. For the fastest fourier transform in the west, see <http://theory.lcs.mit.edu/~fftw>. It may well be the fastest, but it may also rank as one of the most complex of implementations. It is still $O(N \log N)$ but it has a very low constant time. This link also has pointers to public domain software for performing FFTs (including a mixed radix FFT).

The derivation of the Lyon window, using Maple, is new, as far as we know. The Lyon window (two quintics with flat ends) is not new, though its application as a window for signals probably is. In the past the quintic was viewed as a curvature controlled trajectory for maneuvering a car [Lyon 90]. We make no claim as to the suitability of the Lyon window for signal processing since the window has not been thoroughly investigated and this poses a topic of future research.

(CN) 7. An Introduction to Image Processing

From her whole frame--an atmosphere which quite
Arrayed her in its beams, tremulous and soft and bright.
The Revolt of Islam. A Poem in Twelve Cantos.

Canto Eleventh

Peter Packed a Pickled Pixel

Its color was that of hash
for every program that displayed
This pixel it would crash

- D. Lyon

Image processing is a kind of digital signal processing that occurs on a 2-D array of sampled data. For example, it has been shown that image processing techniques may be used to compress DEM (Digital Elevation Map) data [Franklin]. DEM data consists of a 2-D array of ranges, sometimes called a range image.

Often, the input image is acquired by a sensor able to detect energy. Typically the energy to be detected is light energy. For the purpose of this chapter, we shall make the assumption that we will sample an image using a uniform grid over which we perform the spatial sampling of the energy.

When we speak about the resolution of the image, we talk about the size of the array used to store the *significant* image data. For example, an array with 640 columns and 480 rows may contain double precision floating point numbers, but if only 8 bits of information is significant, then we say that the image resolution is 640x480 by 8 bits per pixel.

Image processing has applications in the areas of art, science, industry, government, and space [Holzmann] [Pratt]. Branches of image processing (like digital image warping)

have found applications in remote sensing, computer vision, special effects, and computer graphics [Wolberg].

(B-heading) Video Cameras and Scanners

Energy used to form an image typically starts as an analog signal (just like the 1-D DSP case). The primary difference is that the 2-D transducer is able to detect both the power in the incident energy and its relative direction.

Several techniques are commonly used to determine the relative direction of energy. A mechanical scanning technique (like RADAR) can physically move a sensor in order to obtain a range image. Some flat-bed scanners have a mechanical arm, called a platten, that contains a linear solid-state array. The platten is moved over an illuminated image. Another example of a mechanical scanner is called a drum scanner. The drum scanner (typically a high-end scanner used in pre-press applications) physically spins a drum with mounted art-work.

Electronic scanning is performed by electron-beam deflection. The deflection is accomplished via an electric or magnetic field. The electron beam scans the photosensitive surface which in turn alters the beam current. The current change is proportional to the incident light. Tube cameras are an example of systems that perform electronic scanning. The semiconductor industry has produced solid state alternatives to tube cameras that dominated both in cost and in performance specifications in all but niche markets. As a result, tube cameras have fallen out of favor (both professional and consumer users typically favor solid state cameras).

The primary solid state camera in use today is the CCD (Charged Coupled Device) camera. The CCD camera was invented by Willard S. Boyle and George E. Smith at Bell Labs in the early 1970s. For semiconductor-based sensors, like the CCD cameras, a scanning array will have individual sensing elements, each of which corresponds to a pixel. The size of the elements ranges around 7-14 μm . Recall that 1 μm (also called a micron) is 1×10^{-6} meters. The wavelength of a He-Ne laser (a red light) is 638 nm (nm = nanometers = 1×10^{-9} meters).

Figure 7.1. A Stepped Gray wedge and the Output of a linear scanning array.

Figure 7.1 shows an image with an increasing gray level and a single white line that has been resampled from its center.

Flatbed scanners typically have high-resolution linear CCD elements with a mechanically deflected platten that gives them higher resolution than hand-held cameras. For this reason, people will often take a picture with a film camera, then scan the image to obtain a high resolution scan. Film ranges in quality from common to laboratory grade.

Common grade film has a grain size that is able to yield 100 lines per mm (10,000 nm), (i.e., 35 mm Kodak 5369). Laboratory grade film yields about 1000 lines per mm (1,000 nm) [Kodak]. The high-end 35 mm digital slide scanners are able to yield 2048x3072 pixels with 36 bits per pixel (see <http://www.davidmyers.com.au/rfs2035.htm> for an example). This is about 6.2 million pixels per image. Electronic still cameras have CCDs that have 850x984 pixel resolution (for example <http://www.kodak.com/daiHome/pdf/dc120.pdf>). Video cameras (image sequence digitization cameras) typically operate with CCDs that have resolutions of 640x480 24 bit pixels (for example, <http://www.kodak.com/daiHome/pdf/dvc300.pdf>).

While film is 100 to 1000 times higher in resolution than the CCD cameras, per unit of imaging area, another factor is total size. While the common size for film is 35 mm, there are larger format negatives available for special applications (i.e., 70 mm movie film, 127 mm portrait cameras, large format x-ray film, panorama cameras, etc.). These large formats can make an enormous jump in the number of pixels available. The other factor in the equation is cost. The price of digital still image cameras appears to be dropping quickly. Also, it is not clear that people need the high resolution that film has to offer. We have found that students really have no idea how much data can be produced by a high-resolution scan of a large image. For example, a 3x5 inch color photo will, when scanned at 24 bits per pixel and at 1200 pixels per inch, produce $3 \times 5 \times 1200 \times 1200 \times 3 = 64$ MB of data (MB = MegaByte, Mb = Megabit). We have found students turning up the virtual memory to 450 MB when running Photoshop, so that they can scan 8×10 photos at 1200 pixels per inch ($8 \times 10 \times 1200 \times 1200 \times 3 = 345$ MB). Some students have even said that they need such images for their home pages (No, Really)!

(A-heading) The Observer Interface

While it may seem out of place to put the observer interface here, it is a central class for understanding how the Image class works. Also, we have put off covering the Observer interface until now. The Observer interface resides in the java.util package. It is used to require implementation of methods that are needed to maintain consistency in an object-oriented environment. The relationship between the class that implements the Observer interface and class that extends the Observable class is the relationship between a view and a model (sometimes called the model-view relationship). An example of the model-view relationship is that between the gas tank of a car and a fuel gauge. If the amount of fuel in the tank changes, the fuel gauge readout changes. In Java, the simulation of a fuel tank notifies the fuel gauge readout so that the screen is consistent with the underlying simulation of the fuel tank.

There is a directed flow of information between an instance of a class that extends Observable and the instance of the class that implements Observer. The Observable instance keeps an instance of a vector that lists all the Observer instances that have registered an interest in the state of the Observable instance. Whenever the Observable instance changes, it broadcasts an update message to each of the Observer instances that have registered. This relationship is shown in Figure 7.2.

Figure 7.2. The Observable uses the update method to transmit data to the Observer. A class that implements the Observer interface supports an update method that takes an Object-typed argument.

(B-heading) Interface Summary

```
package java.util;

public interface Observer {
    void update(Observable o, Object arg);
}
```

For usage see the next section.

(A-heading) The Observable Class

The Observable class resides in the java.util package. An instance of the Observable class makes use of an instance of a list of the observer instances that have registered

themselves. The list is called an `ObserverList` and resides invisibly within the `java.util` package.

(B-heading) Class Summary

```
package java.util;
public class Observable {
    public synchronized void addObserver(Observer o)
    public synchronized void deleteObserver(Observer o)
    public void notifyObservers()
    public synchronized void notifyObservers(Object arg)
    public synchronized void deleteObservers()
    public synchronized boolean hasChanged()
    public synchronized int countObservers()
}
```

(B-heading) The NamedObservable

The `NamedObservable` class provides a mechanism to associate a `String` instance with the `Observable` instance. The code for the `NamedObserver` follows:

```
package observers;
import java.util.*;
// The NameObservable is just like an Observable
// only it has a name property associated with
// every Observable instance.
public abstract class NamedObservable extends Observable {

    private String name;

    public synchronized void setName(String nm) {
        name = nm;
    }

    public synchronized String getName() {
        return name;
    }
}
```

(BEGIN NOTE) The `get` and `set` methods in the `NamedObservable` class are synchronized, to prevent contention problems from occurring during multi-threaded operation. It is common to see class variables, such as the `name` string as being accessed only through synchronized method (as discussed in Chapter 2). This is the reason for declaring the `name` string private. (END NOTE)

(B-heading) The ObservableDouble Observer-Observable Example

The DiffCAD program depends on the model-view paradigm and so has a package called `observers`. One of the classes in the `observers` package is called `ObservableDouble`.

When the `setValue` method is invoked on an instance of the `ObservableDouble` class, the `setChanged` method is invoked and the `notifyObservers` method causes an update method to be broadcast to all the interested `Observer` instances.

```
package observers;
import java.util.*;
```

```

public class ObservableDouble extends NamedObservable {

    // The value of interest
    private double value;

    public ObservableDouble(double newValue, String nm) {
        value = newValue;
        setName(nm);
    }

    public synchronized void setValue(double newValue) {
        if (newValue != value) {
            value = newValue;
            super.setChanged();
            super.notifyObservers();
        }
    }

    public synchronized double getValue() {
        return value;
    }
}

```

(B-Heading) DoubleDialog and the IntDialog

As part of the observers package there are DoubleDialog and IntDialog classes. When an instance of the IntDialog is made, a display is generated like the one shown in Figure 7.3.

Figure 7.3. IntDialog with an Embedded Observable on display

The value of the int is initially used for the display in the text field. When the int is altered, by the user, the setValue method causes the observers to be updated. This dynamically alters the text field in the dialog box. The IntDialog class follows (the DoubleDialog is almost the same except that the DoubleDialog contains an ObservableDouble rather than an ObservableInt):

```

package observers;
import java.awt.*;
import java.applet.*;
import java.io.*;

import gui.*;

public class IntDialog extends ClosableFrame {

    Label label1;
    IntTextField textfield;

    Button okButton;
    Button cancelButton;

    ObservableInt i;
}

```

```

private static int offset = 25;

public IntDialog(
    String dialog_title,
    String label,
    ObservableInt sample_int) {
    super(dialog_title);
    init( dialog_title,  label);
    set(sample_int);
    setForeground(Color.white);
    setBackground(Color.white);
}

void set(ObservableInt sample_int) {
    i = sample_int;

    textfield = new IntTextField(i);
    add("Right",textfield);
    textfield.reshape(89, 25, 64, 16);
    cancelButton = new Button("Cancel");
    add(cancelButton);
    cancelButton.reshape(13, 175, 88, 28);

    okButton = new Button("OK");
    add(okButton);
    okButton.reshape(130, 175, 63, 29);

    pack();
    show();
}

public void init(String dialog_title, String label) {
    setLayout(new FlowLayout());
    resize(250, 250);
    reshape(offset, offset,
            250, 250);
    offset = offset + 15;
    setResizable(false);

    //Initialize components
    labell = new Label(label);
    add("Left", labell);
    labell.reshape(5, 19, 86, 23);
}

void ok() {
    System.out.println("OK!");
    String s = textfield.getText();
}

```

```

Integer i_Integer = new Integer(i.getValue());
// default values
try {
    i_Integer = Integer.valueOf(s);
    i.setValue(i_Integer.intValue());
}
catch (NumberFormatException e) {
    System.out.println(
defaults");
    // pick a reasonable default
}
return;
}

private void cancel() {
    System.out.println("Cancel!");
}

public boolean handleEvent(Event event) {
    if (Evt.match(event, 'o', okButton))
        ok();

    else if (Evt.match(event, 'k', cancelButton))
        cancel();
    return super.handleEvent(event);
}
}
}

```

(BEGIN NOTE) We realize that the user may like to make the cancel method perform a different task other than print cancel. This would be a good spot to send an application specific message or restore a value. (END NOTE)

An example of the use of the IntDialog may be found in the AudioFrame class. The example follows:

```

IntDialog spDialog = new IntDialog(
    "start Position Of Samples To Graph dialog",
    "Enter start position:",
    startPositionOfSamplesToGraph
);

```

Where

```

private ObservableInt startPositionOfSamplesToGraph;

```

is declared in the AudioFrame as a class variable.

(B-heading) Dialogs in the ImageFrame

The ImageFrame class resides in the lyon.ipl package. It is used to hold instances that contain image data and display them. It is also used to dispatch events and manage dialogs that prompt the user for parameters. The dialogs all contain instances of Observable

classes. What follows is a list of the Observable class variables in the ImageFrame and the Dialog instance creation that depends on them:

We see that in lines 1 and 2 there are two ObservableDouble instances, each provided with a label (because they are named) and a value.

```
private ObservableDouble multKonst = new
ObservableDouble(1.0, "mult:");
private ObservableDouble addKonst = new
ObservableDouble(0.0, "add:");
private ObservableInt thresholdKonst = new
ObservableInt(128, "Threshold:");
private ObservableInt scaleKonst = new
ObservableInt(1, "Scale:");
private DoubleDialog multDialog = new
DoubleDialog("Multiply Dialog",
            "* k", multKonst);
private DoubleDialog addDialog = new DoubleDialog("Add
Dialog",
            "+ offset", addKonst);
private IntDialog threshDialog = new
IntDialog("Threshold Dialog",
            "Threshold constant", thresholdKonst);
private IntDialog scaleKonstDialog = new
IntDialog("Scale Dialog",
            "Integer Scale constant", scaleKonst);
```

When an instance of the ImageFrame class is made, the IntDialog and DoubleDialog instance appear on the screen. At that point, the user may over-ride the defaults in the dialog boxes and click "OK". When this occurs, the image processing operation in invoked will use the updated values. For example:

```
public void linearComb() {
    pp.linearComb(multKonst.getValue(), addKonst.getValue());
    ;
    updateDisplay(pp);
}
```

(A-heading) The Image Class

In Java, an image is stored in an instance of the Image class. The image class resides in the java.awt package and is designed with the intention that images are produced by I/O bound systems (i.e., networked based systems). The Image class has provisions to take ImageObserver instances in many of its methods. The ImageObserver instance is registered for notification after an image becomes available.

(B-heading) Class Summary

```
package java.awt;
import java.awt.image.ImageProducer;
import java.awt.image.ImageObserver;

public abstract class Image {
    public abstract int getWidth(ImageObserver observer);
```



```

    public abstract int getHeight(ImageObserver observer);
    public abstract ImageProducer getSource();
    public abstract Graphics getGraphics();
    public abstract Object getProperty(String name,
        ImageObserver observer);
    public static final Object UndefinedProperty = new
        Object();
    public abstract void flush();
}

```

(B-heading) Class Usage

The Image class uses the observer-observable model. The Image class is an abstract class and so may not be instantiated. Recall from Chapter 3 that every Component implements an ImageObserver. This means that we can pass an instance of the Component to the image.getWidth invocation to obtain a call-back. The call-back will occur in the form of an imageUpdate method invocation on the interested ImageObserver instance.

Suppose the following variables are predefined:

```

Image image;
ImageObserver imageObserver;
ImageProducer imageProducer;
int height, width;
Graphics graphics;
String name;

```

To get the width of the image use:

```
width = image.getWidth(imageObserver);
```

(returns -1 if image is not ready and notifies the observer when the image becomes ready).

To get the height of the image use:

```
height = image.getHeight(imageObserver);
```

(returns -1 if image is not ready and notifies the observer when the image becomes ready).

To get the instance that produces the image pixels:

```
imageProducer = image.getSource();
```

To get a graphics instance for an off-screen image:

```
graphics = image.getGraphics();
```

To get an image property, by name, use:

```
object = image.getProperty(name, imageObserver);
```

If the image is not available, the getProperty returns null and notifies the imageObserver later. Image.UndefinedProperty is a static final Object instance that is returned when the property is undefined.

To destroy all the resources used by an image that will not be likely to be used for a while use:

```
image.flush();
```

(A-heading) The ImageObserver

The ImageObserver is an interface that resides in the java.awt.image package. Every Component implements the ImageObserver and Components are typically used to make instances of Images. An instance of a class that implements the ImageObserver interface

registers as having an interest in the contents of an ImageProducer instance. When the contents is altered, the ImageObserver instance is notified by the ImageProducer instance, using the imageUpdate method invocation.

(B-heading) Summary

```
package java.awt.image;
import java.awt.Image;
public interface ImageObserver {
    public boolean imageUpdate(Image img, int infoflags, int
        x, int y, int width, int height);
    public static final int WIDTH = 1;
    public static final int HEIGHT = 2;
    public static final int PROPERTIES = 4;
    public static final int SOMEBITS = 8;
    public static final int FRAMEBITS = 16;
    public static final int ALLBITS = 32;
    public static final int ERROR = 64;
    public static final int ABORT = 128;
}
```

(B-heading) Image Instancing in the ImageFrame

In the ImageFrame class, we have a getImage method that prompts the user for an input file, then proceeds to open the image file, waiting until the image file is read and the resources needed to store the Image instance are allocated. This is a design pattern that we have decided to defeat to simplify the programmers' job when reading an image. We find the code to be harder to understand, write and teach and prefer a simpler API for reading an image. To defeat the call-back routine, we make a temporary ImageProducer that is invoked with the method *getImage*. Methods such as getImage greatly simplify user code, but make the assumption that the image resides on a low latency storage device (i.e., the hard disk). The *fileName* and the *image* variables are both class variables in the ImageFrame class. The getImage method follows:

```
public void getImage() {
    // Open up the image, by file name and wait for it!
    fileName = Futil.getReadFileName();
    System.out.println("Get image:"+fileName);
    try {
        image = getToolkit().getImage(fileName);
        waitForImage(this,image);
    }
    catch (Exception e) {
        System.out.println("Get Image could not open
"+e);
    }
    // move the image, change its dimensions
    reshape(100, 100,
        image.getWidth(this), image.getHeight(this));

    //set image title
    setTitle(fileName);
}
```

```

        show();
    }

```

The *waitForImage* method permits avoidance of the call-back method and will have the thread stop before the *getImage* method returns. The *paint* method for the *ImageFrame* invokes the *drawImage* method on the *Graphics* instance, *g*.

```

    public void paint(Graphics g) {
        g.drawImage(image, 0, 0, this);
    }

```

(A-heading) The PixelPlane Class

The *PixelPlane* class resides in the *lyon.ipl* package and is used to give the user a high-level interface to pixel-based operations. The intention of the *PixelPlane* class is to permit the processing of an array of data, without having to follow *ImageProducer-ImageConsumer* design pattern. This makes the software much easier to teach, write and understand. We have found that Java programmers prefer the simpler array model for manipulating images. An instance of a *PixelPlane* class stores its pixels in a 1-D array of *int*. A *PixelPlane* instance does not do image processing. It is just a handy way to access pixels and allocate *Image* instance memory.

The basic idea is that we do not need to keep an instance of an *Image* around. All of the image processing will occur on and between instances of the *PixelPlane* class. Should we need an *Image* instance (for display) we can always make one from an instance of the *PixelPlane* class. The approach of making *Image* instances on-demand represents a space-time trade-off that is typical in computer science. An *Image* instance takes up so much space, that dynamic allocation can free up enough space during processing to make otherwise infeasible operations feasible (from a memory usage point-of-view).

Since Java does not have (as of this writing) any notion of row-major or column-major order, we cannot use doubly-nested for-loops to access 2-D arrays. To do so might cause thrashing to the disk (this is particularly slow and is typical when memory is accessed in a non-sequential fashion). To ease the burden of thrashing, a single-dimensioned array is stored internally in the *PixelPlane* instance. The 2-D array is implemented by multiplying the row index by the column index. Methods are also provided that permit linear indexing into the array, for slightly higher speed of some image processing operations.

(B-heading) Class Summary

```

package lyon.ipl;
import futils.*;
import futils.utils.*;
import java.awt.*;
import java.awt.image.*;
import java.applet.Applet;
public class PixelPlane {
    public int pels[]
    public PixelPlane(int w, int h)
    public PixelPlane(double x, double y)
    public PixelPlane copy()
    public Image makeImage()
    public int getLength()
    public int getHeight()
    public int getWidth()

```

```

    public boolean inrange(int x, int y)
    public int getRed(int i)
    public int getGreen(int i)
    public int getBlue(int i)
    public int getRed(int x, int y)
    public int getGreen(int x, int y)
    public int getBlue(int x, int y)
    public int getAlpha(int x, int y)
    public int MakePixel(int r, int g, int b, int a)
    public void setPixel(int x, int y, int r, int g, int b,
        int a)
    public void setPixel(int x, int y, int pel)
    public int getPixel(int x, int y)
    public int getPixel(int i)
    public void printSize()
}

```

(B-heading) Class Usage

Suppose The following variables are predefined:

```

PixelPlane pp, ppCopy;
int width, height;
double w, h;
int pixelArray[];
Image image;
boolean aBoolean;
int x,y;
int i, c, a, r, g, b, pel;

```

Then to make an instance of a PixelPlane that is width by height:

```
pp = new PixelPlane(width, height);
```

The PixelPlane constructor is overloaded to be used with double-type dimensions:

```
pp = new PixelPlane(w, h);
```

To get at the internal pixel array (this is not suggested, but permitted to maintain flexibility):

```
pixelArray = pp.pels;
```

To make a copy of the PixelPlane (this uses the System.arraycopy method, so it should be pretty quick):

```
ppCopy = pp.copy();
```

To make an Image instance from a file (makes a standard file open dialog box) use:

```
image = PixelPlane.openImage();
```

openImage is a static method, so you don't need to make a PixelPlane instance. To get the total number of pixels in the PixelPlane instance, use:

```
int numberOfPixels = pp.getLength();
```

To get the width and height, in pixels, from the PixelPlane instance, use:

```
height = pp.getHeight();
```

```
width = pp.getWidth();
```

To see if two x,y coordinates are in range, use:

```
aBoolean = inrange(x,y);
```

To get a color stored into an int from a location in the PixelPlane instance (using the linear array):

```
c = pp.getRed(i);
```

The color value, c, always varies from 0...255.

```
c = pp.getGreen(i);
```

```
c = pp.getBlue(i);
```

To get the same color values, using the (x,y) coordinates:

```
c = pp.getRed(x, y);
```

Each access into the pixel plane costs one multiply. After Sun releases an API that stores multidimensional arrays, in known order, the multiply can be eliminated, speeding pixel access:

```
c = pp.getGreen(x, y);
```

```
c = pp.getBlue(x, y);
```

```
c = pp.getAlpha(x, y);
```

A pixel is stored in a packed in, 4 bytes per pixel, ARGB format. To pack a pixel:

```
pel = pp.MakePixel(r, g, b, a);
```

To set a pixel at location (x, y):

```
pp.setPixel(x, y, r, g, b, a);
```

To set a pixel equal to the packed pel at location (x, y):

```
pp.setPixel(x, y, pel);
```

To get a packed pixel at location (x, y):

```
pel = pp.getPixel(x, y);
```

To get a packed pixel from the linear array at location i:

```
pel = pp.getPixel(i);
```

To print the PixelPlane instance's size to the console:

```
pp.printSize();
```

(A-heading) The ProcessPlane Class

The ProcessPlane class resides in the lyon.ipl package. It consists of an extension to the PixelPlane class. By creating instances of the ProcessPlane class, we add methods to the PixelPlane instance that provide for elementary image processing services. The services themselves are not profound, but how they are implemented in Java is of interest. A class summary appears below.

(B-heading) Class Summary

```
package lyon.ipl;
import java.awt.*;
public class ProcessPlane extends PixelPlane {
    public ProcessPlane (double x, double y)
    public ProcessPlane (int x, int y)
    public void cornergray() {
    public ProcessPlane scale(int scale) {
    public void threshold(int konst)
    public void diagGray()
    public ProcessPlane edge()
    public void linearComb(double konstD, double akD)
    public void Subimage(ProcessPlane pp)
    public void makeGray()
```

```

    public void randResample()
    public void shadow()
    public void negate()
}

```

(B-heading) Class Usage

Suppose the following variables are predefined:

```

ProcessPlane pp, pp2;
int height, width;
double heightD, widthD, a, b;
int scale, konst;

```

To make an instance of a ProcessPlane that allocates internal pixel storage for a heightxwidth by 32 color, RGBA (Red, Green, Blue and Alpha) image:

```
pp = new ProcessPlane(height, width);
```

To make an instance of a ProcessPlane that uses doubles for size and casts the doubles to int before allocating storage:

```
pp = new ProcessPlane(heightD, widthD);
```

To make an image of the previous images size, but fill it with a cornergray test pattern, as shown in Figure 7.4:

```
pp.cornergray();
```

Figure 7.4. The cornergray method output

To scale an image by a positive integer using pixel replication:

```
pp.scale(konst);
```

To threshold an image using a integer value that ranges from 0 to 255, as shown in Figure 7.5:

```
pp.threshold(konst);
```

Figure 7.5. Effect of threshold on a image

To make a diagonal test pattern using the existing images dimensions, as shown in Figure 7.6:

```
pp.diagGray();
```

Figure 7.6. The effect of the diagGray method

To make a single pixel-width wide edge from the left-most edge on the image, at the center of a thick bright stripe, as shown in Figure 7.7, use:

```
pp.edge();
```

Figure 7.7. Effect of the edge method

The edge method uses a domain-specific edge detector that is useful for range-finding via diffraction. See the following section for more information about the implementation.

(BEGIN NOTE) This edge detection method will produce unsatisfactory results when applied to images outside of the intended domain. (END NOTE)

To perform the operation $p_i = ap_i + b \quad \forall_i \in [i = 0 \dots \text{numberOfPixels}]$ where

$p_i =$ ith image pixel:

```
pp.linearComb(a, b);
```

To subtract pp2 from pp, leaving the result in pp (i.e., $pp = pp - pp2$):

```
pp.Subimage(pp2);
```

To make pp a gray image by copying the red plane to the green and blue planes:

```
pp.makeGray();
```

To perform a stocastic resampling, of the image, an effect illustrated in Figure 7.7:

```
pp.randResample();
```

Figure 7.8. The effect of randResample

To perform a shadow mask on the image (as shown in Figure 7.9) use:

```
pp.shadow();
```

Figure 7.9. Effect of the shadow method

To negate the image (as shown in Figure 7.10) use:

```
pp.negate();
```

Figure 7.10. Effect of the negate method

This section showed a collection of image processing services performed by instances of the ProcessPlane class. While all the images are shown as gray-scale images, this is a limitation of the printing process. All the operations are 24 bit color operations.

(B-heading) Class Implementation, The negate method

None of the image processing operations shown in this section are remarkable. Perhaps the more interesting question is that of how the operations are implemented. We start with the simplest of the image processing operations, the negate method:

```
public void negate() {
    int r,g,b,a;
    for (int y=0; y < getHeight(); y++)
        for (int x =0 ; x < getWidth(); x++) {
            r = getRed (x,y);
            g = getGreen(x,y);
            b = getBlue (x,y);
            a = getAlpha(x,y);
            setPixel(x, y, 255 - r,255 - g,255 - b, a);
        }
}
```

The negate method is added to the ProcessPlane class to enable any instance of a ProcessPlane to be able to negate itself. (BEGIN NOTE) Since we extend the PixelPlane class, we inherit all the methods from that class. Also, we do not negate the alpha plane, since that would make the image black. (END NOTE)

Within the ImageFrame class, we have a MenuItem instance called negate_mi:

```
MenuItem negate_mi = addItem("[n]negate");
```

Where the addItem method adds the MenuItem instance to the main menu bar:

```
public MenuItem addItem(String itemName) {
    MenuItem mi = new MenuItem(itemName);
    m.add(mi);
    return(mi);
}
```

Finally, we handle the event using the Evt's match method to select either the keyboard event or the menu selection event:

```
public boolean handleEvent(Event e) {
    if (Evt.match(e,negate_mi)) {
```

```
        negate();
        return true;
    }
}
```

(B-heading) Class Implementation, The Shadow method

The Shadow method is a filter that moves a 2 by 2 pixel window, one pixel at a time, across an image. It replaces the pixel at the upper left corner of the window with the value of the pixel minus one-half the value of the pixel at the lower-right corner.

```
public void shadow() {
    int r,g,b,a;
    for (int y=0; y < getHeight()-2; y++)
        for (int x =0 ; x < getWidth()-2; x++) {
            r = getRed (x,y);
            g = getGreen(x,y);
            b = getBlue (x,y);
            a = getAlpha(x,y);
            r = r + (127 - getRed (x+2,y+2));
            g = g + (127 - getGreen(x+2,y+2));
            b = b + (127 - getBlue (x+2,y+2));
            setPixel(x,y,r,g,b,a);
        }
}
```

(B-heading) Class Implementation, The edge method

The edge method is probably one of the more complex image processing methods in the ProcessPlane class. The edge method runs through the following steps:

1. Compute the average intensity, a , for pixel row, y .
2. Find the pixel, x_r , that exceeds a .
3. Find the very next pixel, x_l , that does not exceed a .
4. Compute the average of x_r and x_l and call it x_a .
5. Store x_a, y as an edge point.
6. Increment y .
7. If $y > \text{imageHeight}$ then stop, else goto 1.

This type of ad-hoc algorithm will always find the left-most bright edge for a well defined stripe. At the same time, it finds the centroid of the stripe. As the stripe gets closer to the left of the image, the program runs faster (since it has to search fewer pixels to find the stripe). Thus, good lighting and camera positioning are needed to make this algorithm run fast. The Java implementation of the edge method follows:

```
public ProcessPlane edge() {
    ProcessPlane pp = new ProcessPlane(getWidth(),
    getHeight());

    int r,a;
    int average; // average intensity
    for (int y = 0; y < getHeight(); y++)
        new_line: {
            average = 0;
            for (int x =0 ; x < getWidth(); x++)
                average = average + getRed (x,y);
        }
}
```



```

        average = average / getWidth();
    for (int x =0 ; x < getWidth(); x++) {
        r = getRed (x,y);
        a = getAlpha(x,y);
        if (r > average) {
            int start=x;
            for (int i=x;i < getWidth(); i++) {
                r = getRed (i,y);
                if (r < average) {
                    pp.setPixel(
                        start + (i - start) / 2,
                        y,255,255,255,a);
                    break new_line;
                }
            }
        }
    }
    return pp;
}

```

(CN) 8 (CT) Digital Images and Image Formats

"This is the Wild West of the
Information Age" -

Bart Kosko

(A-heading) The DataBahn

As the Internet connects the world together, and as more users cram to get onto this information thoroughfare, we find that it can take a long time to get data from point A to point B. Never before as now do we need to squeeze information content down in size before it is exchanged or transferred. Luckily the enormous size of digital images have already motivated the creation of a number of space saving image and file formats. Consider this: an uncompressed digital bitmap image of 640x480 pixels with 256 colors takes up 307 KB, or 1/3 MB. It can be frustrating to watch an image file of this size load into your browser from a web site.

In this chapter, you will find a discussion of the what and why of image formats - there are a lot of them ! You will learn some of the latest and greatest formats for the Internet. Finally, you will see details of the formats that are supported in Diffcad.

(A-heading) A Bird's Eye View of Image Formats

There are three broad categories of image formats: vector formats, bitmap formats and other formats.

(B-heading) Vector and Bitmap formats

In describing an image, you can resort to several levels of detail. In the lowest level of abstraction, you may describe each and every element (pixel) of the image. An image that is described in this manner is referred to as bitmapped, since the end result is a map of bits (or pixels). A bitmap image and its associated data are shown in Figures 8.1a and 8.1b.

Figure 8.1a A bitmap format image**Figure 8.1b Bitmap format data**

This is ultimately, how images are represented for display on monitors and viewed. A computer monitor or television has an addressable array of physical pixels or dots. The dot has position, and color information. Color will be talked about further very shortly. Bitmap images are difficult to scale from their original resolution (without DSP!). They can be bulky and hence cumbersome to transport. You can mitigate the size problem with compression (covered soon), but only at the expense of increased time to decode and render.

(B-Heading) Vector formats

Now we consider a way to represent images with a higher level of abstraction. Suppose you store endpoints of line segments to compose a representation of an image. This may be useful to render a CAD drawing for example. You store coordinates for the starting point, a direction and a length and maybe some color information. The rest of the screen that does not have line segments will be a background color. This is a simple vector image file format, and is a good compact format for line drawings. Vector formats are quick to read and are compact, for the types of images they are intended to represent. Vector formats typically store not only line primitives, but also some 2D shapes, such as circles and squares and curved lines, which are higher levels of abstraction. These shapes could be used to compose jet planes or integrated circuit layouts, for example. Figures 8.2a and 8.2b show an example of a simple vector image and its associated data.

Figure 8.2a Vector format image**Figure 8.2b Vector format data**

You could continue on using higher and higher levels of abstraction going to 3D objects as primitives, and Avatars (3D computer puppets) in virtual worlds.

One advantage of a vector format image, is that it is relatively easy to scale the image without loss of detail. Many clipart collections are stored as vector format files so that they can be scaled easily. A disadvantage of vector formats is that it is hard to store very detailed image information such as photographs, where you may need to vary color information on a pixel by pixel basis.

(C-heading) Conversion between vector and bitmap formats

Converting from a vector format image to a bitmap format is easy and straightforward; in fact, this conversion will be very common, since most display output devices are bitmapped. For a very detailed vector format image, it is important to choose a high

enough resolution for the destination bitmap, otherwise, some artifacts will appear in the image, such as jagged lines ("the jaggies") instead of straight lines.

Converting from a bitmap format image to a vector image is difficult. In Chapter 9, you will meet this formidable challenge with DSP routines for edge and outline detection.

Another issue is the possible loss of color information when you go from a rich bitmap representation to a (possibly) poor vector representation.

(B-heading) Other types of formats

[Murray et al.] describe several other types of formats for digital images. These are described briefly as follows:

- Scene - A scene format file has a condensed representation of an image. It is sometimes hard to tell the difference between this format and a vector format.
- Metafile - A metafile can store both vector format elements and bitmap format elements. Examples of this type of format file are the PICT format and the CGM format. Because of their versatility, these files are often used to cross the bridge between different hardware or software platforms.
- Animation - Animation formats come in many flavors. The simplest type just stores adjacent frames of an animation sequence in one file for playing. Another type stores not only images but along with them color maps for the images. Changing the color map can give the illusion of motion. Finally, a more sophisticated animation format will store frame difference information along with key frames. This technique is used to store motion video also and exploits the fact that in any given movie or animation, from frame to frame there is on average not a lot of changed information. There is usually a large chunk of the background or features that are static. If you store the data that changes, instead of all the data, you save a lot of space.
- Multimedia - Multimedia formats allow you to store all kinds of different data types and formats together; you could have video information, text information and sound information coexisting peacefully.
- 3D - 3D formats not only support descriptions of lines, shapes and 3D geometries, but also textures, reflections and anything else a rendering program would need to reconstruct a 3D image or world. Objects in a 3D file are sometimes called *scene elements*. Many existing vector file formats have been extended to support 3D. Such formats, such as Autodesk's DXF format, are referred to as *extended vector formats*. VRML is a little more than a 3D format, since it includes support for HTML style linking to other URLs on the World Wide Web.
- Font(bitmap, stroke, outline) - Fonts are special graphic files. They come in their own subsets of types based on bitmap formats or on vector formats (stroke, outline). One additional constraint usually imposed on font files is that they must be very quick to index into. So there is usually a database index associated with the font data placed in a header or footer of the file.
- PDL - PDL, or Page Description Language formats are usually textual programmatic descriptions of how to render graphics and text. An example of this type of format is the ubiquitous Postscript format. This format is more akin to source code rather than just graphics data, and hence requires a sophisticated program in order to be able to create output.

(B-heading) Color Depth, Palettes and Transparency

Before getting to a monitor, an image, regardless of the format it is stored in by an application, is usually represented by a bitmap in a special memory known as a frame buffer. (The exception would be for a random scan display monitor - a vector image display monitor.) The frame buffer stores the image that is to be displayed as it is being scanned by the graphics processor that feeds a CRT monitor for viewing. En route to the monitor, the graphics processor may be commanded by the host processor (CPU) to manipulate the stream of pixels before they are displayed. One example of processing is to convert *color depth* information by using a *color look-up table* (CLUT), also known as a *color palette*. Color depth refers to the number of bits that are used to represent one pixel, or *bits per pixel* (bpp). For a color depth of 1, you represent a black and white image; a 1 turns on a pixel (white) and a 0 turns off a pixel (black). A color depth of 8 bpp, means that you have 256 ($=2^8$) possible values for a color.

A CLUT is a table of color values with an index. Use of this table can allow for some image size reduction. You essentially form an indirect addressing scheme. Say you have 256 possible values for a color (indexes). You may store 24 bit per pixel color values in each entry of the CLUT since your hardware supports it. You achieve some data compression since your image data may reference 8 bit CLUT indexes, instead of 24 bit color data. Your image data must also include the CLUT too however. The total data for a 640x480 image is $(640 \times 480 \times 8 + 256 \times 24) / 8 = 308\text{K}$ bytes. If the CLUT was not used, the total data would be $(640 \times 480 \times 24) / 8 = 922\text{K}$ bytes. In this example the use of a CLUT results in a savings of 2/3 in file size.

A CLUT doesn't always make sense to use. If you use a large number of colors in an image, then it may be more space-efficient to store the full pixel value directly.

Generally, for images with more than 256 colors, it is better to store *literal* or *absolute* format, because the overhead of a very large CLUT is not worth the space. In fact the size of the CLUT may approach the size of the image itself.

(C-heading) Transparency

In the television world, you often see live video being overlaid onto a static image (like a weather map). In the bitmap world, this is like overlaying two bitmaps onto each other and specifying portions of one bitmap to be transparent, in certain areas, to allow the background image to show through. Similarly, you could use transparency characteristics to do a fade from one video source to another. In this case, there would be degrees of transparency (not just on or off). Transparency is often described in bitmaps on a pixel by pixel basis. Here, transparency information is appended to pixel value information. The TGA format for example uses 5 bits each for R, G, B (red, green, blue) and 1 extra bit for transparency, for a total of 16 bits. When the transparency bit is on, the display hardware must ignore that particular pixel, so that any background image may show through. A 32-bit variant of the TGA format specifies 8 bits for transparency, called the *alpha channel*. Here each of R, G and B use 8 bits and alpha uses an additional 8 bits to specify the degree of transparency (0=completely transparent to 255=completely opaque).

(A-heading) Graphics Formats Menu

This section will give you a directory of some of the more popular image formats and some of their traits. Below, Table 8.1 shows a list of many different formats along with

their type (adapted from [Murray et al.]) Diffcad uses a subset of these formats, namely: GIF, JPEG, VEC, PICT and PPM.

Later in the section, you will read about general characteristics of graphics formats, such as file organization, compression and progressive display.

Table 8.1 Image formats and type (adapted from [Murray et al.]

<u>Format</u>	<u>Type</u>
Autocad DXF	Vector
Autodesk 3D Studio	Scene description
BMP (Windows)	Bitmap
CGM	Metafile
FLI	Animation
GEM Raster	Bitmap
GEM VDI	Metafile
GIF	Bitmap
Harvard Graphics	Metafile
IFF	Bitmap
Intel DVI	Multimedia
JPEG File Interchange Format	Bitmap
Kodak PhotoCD	Bitmap
MPEG	Multimedia
PCX (Windows)	Bitmap
PICT (Mac)	Metafile
Pixar RIB	Scene Description
PNG	Bitmap
POV	Vector
PPM	Bitmap
QuickTime	Multimedia
Rayshade	Scene Description
SPIFF	Bitmap
Sun Raster	Bitmap
TIFF	Bitmap
TTDDD	Vector and Animation
Utah RLE	Bitmap
VEC	Vector
WMF (Windows)	Metafile

Most graphic formats support some form of data compression. In the next section you will see a discussion of compression methods.

(B-heading) Compression Methods : Making Bits of Bits

There are compression methods that are used on digital information (not just image data), and other methods specifically suited for image data and other special classes of data. First we look at four methods of general data compression: RLE, LZW, Huffman encoding and Arithmetic encoding. Later in this section we look at a compression method that is well suited for image data: DCT or transform based compression.

Compression can be *lossy*, or can be perfect. Lossy compressors discard information, albeit information that is considered to be the least relevant in a particular application. You will read more on this shortly.

Compression can be *symmetric*; that is the process of compression is very similar in complexity, time and methodology to decompression. On the other hand, *asymmetric* compression is a situation where a more complicated process is needed for one direction over the other; an example is the original Intel DVI video format, where a parallel supercomputer is used to compress a video sequence, while a tiny amount of microcode in a video DSP chip is used to decompress the sequence. This is highly asymmetric. For Bitmap format files, normally only the bitmap data is compressed. Any other information in the file (header, footer) is left uncompressed for easy reading. For Vector format files, there is usually no compression. This is because Vector formats are inherently compact being a higher level of abstraction than a bitmap. Also rendering a vector format file takes a lot of time to begin with and adding decompression would further slow down applications that use vector format files.

Start NOTE

In the discussions below, encoding is usually discussed. The decoding process is just the set of reverse operations to that of encoding.

End NOTE

(C-heading) Run Length Encoding (RLE)

Run length encoding is a general compression method that takes sequences or runs of a particular character, and encodes it more compactly as a number and the character.

For example:

AAAAAAAAAABC

could be coded as: 10A1B1C

The number 10 is the run count and the following letter A is the run value. If each ASCII character takes up 1 byte of storage, then the original uncompressed string takes up 12 bytes, while the compressed string takes up 7 bytes. Notice that even a run length of 1 requires a minimum of 2 characters.

For binary character encoding, there are several choices. You can encode on a bit basis (looking for runs of bits), on a byte basis or on a pixel basis, where a pixel may take up multiple bytes. The overhead of storing a run length for each run value may in some cases cause a file to be larger than the original, which is termed negative compression. One method to minimize the effect of the run length code for small runs is to encode a bit at the beginning of each block that enables run length interpretation for that block. In other words, if the enable bit is set to 1, the block is interpreted as run length encoded. If it is set to 0, then the following data is interpreted as unencoded or *literal* data.

With 2D bitmap data, you have freedom to encode data along rows, which are also referred to as *scan lines*, or along columns, or along some other sub-block partitioning of the data.. You could choose some of the options discussed to achieve the best compression.

(C-heading) Lempel, Ziv, and Welch Compression (LZW)

A very widely used algorithm for data compression was invented by Lempel, Ziv and Welch and is known as LZW. Actually there are several algorithms: LZ77, LZ78 and LZW are all patented; use of these may be subject to licensing fees and a lot of legal headaches. It is possible to adapt LZ77 so that you do not infringe on its patent (see PKZIP below). Unfortunately, several widespread programs and formats made use of these patented algorithms, like the CompuServe GIF file format. As a backlash against having to pay for what used to be in the public domain and hence free, several alternatives to the patented LZW algorithms were developed and offered to the public. The popular archiving compressor, PKZIP replaced the original LZW compressor with a compressor based on an adapted non-infringing variation of the LZ77 algorithm. GIF is still oppressed with infringement problems. The world is still full of GIF images however. There are many freeware utilities to convert GIF files to other formats, such as PPM. The PNG graphics format was created specifically as an alternative to GIF and again is also based on a non-infringing variation of the LZ77 algorithm.

How does LZW work? The LZ family of compressors are *dictionary-based encoding algorithms*. As data is read by a compressor, a table or *data-dictionary*, is built that has entries for patterns that occur in the input data stream. If new data that is read, is not in the dictionary, then a new entry is made in the table for it. When data that has a dictionary entry is read in, then the entry, which has a smaller size than the original data, is copied to the output (compressed) data stream. A key feature of LZW is that the dictionary does not need to be stored for the decoder; the decoder will be able to reconstruct the dictionary because of the way that the data is organized. This can save a lot of overhead and space. LZW is a lossless compression scheme.

(C-heading) Huffman Encoding

Like LZW, Huffman encoding is based on code words. Here, shorter codes are chosen to represent the most commonly occurring sequences in a data stream, while longer codes are used for less frequent sequences. The letter A, if used very frequently in some input text, may be coded with 2 bits instead of the usual ASCII 8 bits, while the letter Q, which occurs very infrequently may be coded with 12 bits, as an example. The dictionary used for encoding is required for the decoder to do its work. There is no on-the-fly construction of a dictionary as you saw in the LZW compressor. The data stream that is produced from Huffman Encoding has a subtle requirement: Each code word should not be the prefix of any other code word. This will allow a decoder to uniquely determine each entry of the table based on a sequential read of the data stream. An improvement on Huffman Encoding is Arithmetic Encoding, which is discussed next. Both Huffman Encoding and Arithmetic Encoding are lossless compression schemes.

(C-heading) Arithmetic Encoding

Arithmetic Encoding, or *entropy coding*, improves on Huffman Encoding in a couple of ways: (1) you can have fractional codes, that is you can have a 4.18 bit long code (this is defined in a statistical way) and (2) more complex statistics are used that look at context information to derive a code for an input pattern - a U may be assigned a long code because it does not occur too often, while a U following a Q may be assigned a short

code, since a U is very likely to follow a Q. A brand of Arithmetic Encoding, called a Q-coder is patented by IBM and AT&T and is subject to licensing considerations. An extension of the JPEG compression standard uses the Q-coder.

(C-heading) DCT based or Transform based Compression

The Discrete Cosine Transform (DCT) converts image data to the frequency domain, much like the DFT, the Discrete Fourier Transform which is discussed at length in the next chapter. The DCT is a special case of the DFT [see Netravali et al.]. The transform yields a set of values that correspond to magnitudes of frequency components. The human eye cannot distinguish very high frequency color changes, and this information may be discarded without a great loss in detail of an image. Also, transform values which are zero or close to zero may be effectively compressed with a lossless compression scheme such as Huffman encoding as may be done in JPEG. The overall JPEG compression method is lossy.

(B-heading) Progressive Display and the Internet

For users of the Internet and the WWW, it is very useful to allow for progressive display of graphics images. This means that when a user is navigating the Web and goes to a new destination, if graphics data is loaded for display, it is shown while it is loading, so the user can immediately recognize the image, instead of waiting for the entire image file to load before seeing anything. A few formats allow for this: GIF and its patent-free successor, PNG, and JPEG. In GIF, there is an option to store graphics data with every eighth line of data, then every fourth line, then every second line and finally every line, for a total of four passes over the data. You can see a preview of an image with only one-eighth of the complete data in this scheme. This storage option is called the interlaced option for GIF. The non-interlaced storage option just stores rows sequentially and does not allow for progressive display.

Figure 8.3 Interlaced and Non-interlaced storage in GIF

The PNG format goes one up on GIF. PNG has an interlaced format, where every eighth pixel of every eighth line is first transmitted. This allows an image to be viewed with only 1/64 of the full image data.

JPEG data streams have an option for progressive display. Rather than being based on scan lines, the image is sent in progressively more detailed layers. That is, approximations of the original image are sent in sequence, so that the viewer sees the whole image right away, and the quality of the image improves with time. Each scan of progressive JPEG takes a full JPEG decompression cycle to display, which can be CPU intensive however. Another extension of JPEG provides for hierarchical storage of the same image at multiple resolutions, where a complete image is available at different resolutions to match the resolution of the display or print hardware.

(A-heading) Details of several formats

In this section you will read about details of some file formats, including those that are used by DiffCAD. Note that the Sun Java AWT class library provides built-in support for reading and writing JPEG and GIF image files. DiffCAD provides wrapper classes for some of this functionality. See the classes: WriteGIF, ReadGIF and VSIImage.

(B-heading) GIF

GIF is a file format that uses the LZW compressor, as mentioned previously. There are two versions, GIF87a, the original and GIF89a. GIF89a may be incompatible with software that reads only GIF87a images, so most modern readers are expected to be able to read both formats. The formats are similar but GIF89a has further extensions. The file layout is shown below for both formats in Figure 8.4, and this highlights the differences:

Figure 8.4 GIF87a and GIF89a file layout

There are several pieces to the file format which are discussed in detail below:

- Header - the header is 6 bytes in size. The first 3 bytes are “GIF” to identify the format as GIF. The next three bytes are the version “87a” or “89a”.
- Logical Screen Descriptor - This is a fixed size group of bytes that contain information about the minimum screen resolution (height and width), and color information to reproduce the image. If the screen is smaller than the screen parameters, then some scaling will need to be performed by the application to display the image.
- Global Color Table - This is an optional section that contains a CLUT of up to 256 entries.
- Local Image Descriptor - This section has characteristics of the image data that follows including, where on the display the image should start and the image resolution and color information.
- Local Color Table - GIF is expandable to be able to include more than one image, though this is rarely used. There is therefore the provision to include a color table (termed “local”) for each of the images. This is an optional table for specifying a CLUT for the image data that follows. This table, if present, supercedes, the Global Color Table.
- Image Data - Image data when compressed by LZW usually comes out as a stream of data that must be read from beginning to end. GIF splits the data into a series of sub-blocks. Each sub-block starts with a count byte, which can range in value from 1 to 255. The count byte value specifies the number of data bytes that will follow. At the end of the sub-block a byte of value zero is used to terminate the sub-block.

(B-heading) JPEG/JFIF

The JPEG standard leaves some ambiguities that make it an incomplete file format standard. C-Cube Microsystems created a file format called JPEG File Interchange Format (JFIF) that fills in the gaps. It is completely based on the baseline JPEG standard. JPEG is generally best applied to high-resolution full color (24bpp) images. This is because the transform-based coding will have more latitude for compression with more color information. Keep in mind that sharp edges, such as those created by overlaid text, can become blurry. When compressing with JPEG, an application usually presents a quality setting that you may change to trade off compression to quality. For high frequency detail in your source image, you may want a high quality setting. This will result in lower compression however. The tradeoff between quality and compression is a thorny and persistent issue for JPEG.

A JPEG encoder uses the following steps:

1. Create header information
2. Read in the source image data in RGB
3. Transform data to YUV color space - Y is black and white intensity information, and the U and V channels have color information. This is done with a linear transformation (see Chapter 9 for color space conversion).
4. Subsample the U and V channels - that is throw away some color information because it should be imperceptible to the viewer; use fewer samples of U and V for every sample of Y.
5. Perform the DCT on the Y, U and V data.
6. Quantize the resulting coefficients into different bins (this performs some compression by reducing the number of different possible values; more aggressive quantization is used for the color components).
7. Huffman encode the quantized data and produce an output data stream.

Both a raw JPEG file and a JFIF file start with the the bytes 255 and 232 to signify the start of image marker. For a JFIF file, you will see the bytes 255 and 240 followed by the characters “JFIF”, and information about the image. Data that follows the first block is standard JPEG data as defined by the specification. For detailed information, obtain the specification from the American National Standards Institute [see ANSI].

(B-heading) PPM

PPM is a bitmap format that is used as an intermediate format when converting from one system or file format to another. There are a set of portable freeware utilities written by Jeff Poskanzer that convert to and from PPM to many other graphic file formats. For example, *ppmtogif* converts from the PPM format to GIF.

The file organization is extremely simple for PPM. You start with an ASCII header, and the bitmap data follows as either ASCII data or binary data. No compression is used. Data elements are separated by white space (space, tab, carriage return or linefeeds).

The PPM header looks like the following:

MagicValue P3= ASCII data, P6= binary data
 ImageWidth Width of image in pixels (ASCII decimal value)
 ImageHeight Height of image in pixels (ASCII decimal value)
 MaxGrey Maximum color value (ASCII decimal value)

The MaxGrey value specifies the maximum value for a color component. Each pixel is specified by three values for R, G and B components.

Here is an example file:

```
# example of a 3 x 3 bitmap
P3
3 3
255
0 0 0      0 0 0      0 0 255
0 0 128    0 7 0      0 1 89
9 0 0      0 9 9      0 0 0
```

Comments may be included in a file starting with the # character. The bitmap is for 3 pixels tall by 3 pixels across. The third pixel of the second row has RGB values of (0, 1, 89).

Because PPM is a simple format, the entire source code is shown below in listing 8.1 -how DiffCad implements a PPM reader.

Listing 8.1 Reading the PPM format: The ReadPPM class

```
/**
 * ReadPPM is a class that reads an image from
 * a PPM format file.
 *
 * Victor Silva (victor@cse.bridgeport.edu).
 *
 */

import java.io.*;
import java.awt.image.*;

public class ReadPPM
{
    public ReadPPM(InputStream in)
    {
    }

    private int type;
    private static final int PBM_ASCII = 1;
    private static final int PGM_ASCII = 2;
```

```
private static final int PPM_ASCII = 3;
private static final int PBM_RAW = 4;
private static final int PGM_RAW = 5;
private static final int PPM_RAW = 6;

private int width = -1, height = -1;
private int maxval;

/// Subclasses implement this to read in enough of the image stream
// to figure out the width and height.
void readHeader(InputStream in) throws IOException
{
    char c1, c2;

    c1 = (char) readByte( in );
    c2 = (char) readByte( in );

    if (c1 != 'P')
    {
        throw new IOException( "not a PBM/PGM/PPM file" );
    }
    switch(c2)
    {
    case '1':
        type = PBM_ASCII;
        break;

    case '2':
        type = PGM_ASCII;
        break;

    case '3':
        type = PPM_ASCII;
        break;

    case '4':
        type = PBM_RAW;
        break;

    case '5':
        type = PGM_RAW;
        break;

    case '6':
        type = PPM_RAW;
        break;

    default:
```

```

        throw new IOException( "not a standard PBM/PGM/PPM file" );
    }
    width = readInt( in );
    height = readInt( in );
    if ( type != PBM_ASCII && type != PBM_RAW )
    {
        maxval = readInt( in );
    }
}

int getWidth()
{
    return width;
}

int getHeight()
{
    return height;
}

void readRow( InputStream in, int row, int[] rgbRow ) throws IOException
{
    int col, r, g, b;
    int rgb = 0;
    char c;

    for(col=0; col<width; col++)
    {
        switch(type)
        {
            case PBM_ASCII:
                c = readChar( in );
                if ( c == '1' )
                {
                    rgb = 0xff000000;
                }
                else
                {
                    if ( c == '0' )
                    {
                        rgb = 0xffffffff;
                    }
                    else
                    {
                        throw new IOException( "illegal PBM bit" );
                    }
                }
            }
        }
        break;
    }
}

```

```
case PGM_ASCII:
    g = readInt(in);
    rgb = makeRgb(g, g, g);
    break;
case PPM_ASCII:
    r = readInt( in );
    g = readInt( in );
    b = readInt( in );
    rgb = makeRgb( r, g, b );
    break;
case PBM_RAW:
    if ( readBit( in ) )
    {
        rgb = 0xff000000;
    }
    else
    {
        rgb = 0xffffffff;
    }
    break;
case PGM_RAW:
    g = readByte( in );
    if ( maxval != 255 )
    {
        g = fixDepth( g );
    }
    rgb = makeRgb( g, g, g );
    break;
case PPM_RAW:
    r = readByte( in );
    g = readByte( in );
    b = readByte( in );
    if ( maxval != 255 )
    {
        r = fixDepth( r );
        g = fixDepth( g );
        b = fixDepth( b );
    }
    rgb = makeRgb( r, g, b );
    break;

default:
    break;
}
rgbRow[col] = rgb;
}
```

```

private static int readByte(InputStream in) throws IOException
{
    int b = in.read();

    // if end of file
    if (b == -1)
    {
        throw new EOFException();
    }
    return b;
}

private int bitshift = -1;
private int bits;

private boolean readBit( InputStream in ) throws IOException
{
    if ( bitshift == -1 )
    {
        bits = readByte( in );
        bitshift = 7;
    }
    boolean bit = ( ( ( bits >> bitshift ) & 1 ) != 0 );
    --bitshift;
    return bit;
}

/// Utility routine to read a character, ignoring comments.
private static char readChar( InputStream in ) throws IOException
{
    char c;

    c = (char) readByte( in );
    if ( c == '#' )
    {
        do
        {
            c = (char) readByte( in );
        }
        while ( c != '\n' && c != '\r' );
    }

    return c;
}

/// Utility routine to read the first non-whitespace character.
private static char readNonwhiteChar( InputStream in ) throws IOException
{

```

```

char c;

do
{
    c = readChar( in );
}
while ( c == ' ' || c == '\t' || c == '\n' || c == '\r' );

return c;
}

/// Utility routine to read an ASCII integer, ignoring comments.
private static int readInt( InputStream in ) throws IOException
{
    char c;
    int i;

    c = readNonwhiteChar( in );
    if ( c < '0' || c > '9' )
    {
        throw new IOException( "junk in file where integer should be" );
    }

    i = 0;
    do
    {
        i = i * 10 + c - '0';
        c = readChar( in );
    }
    while ( c >= '0' && c <= '9' );

    return i;
}

/// Utility routine to rescale a pixel value from a non-eight-bit maxval.
private int fixDepth( int p )
{
    return ( p * 255 + maxval / 2 ) / maxval;
}

/// Utility routine make an RGBdefault pixel from three color values.
private static int makeRgb( int r, int g, int b )
{
    return 0xff000000 | ( r << 16 ) | ( g << 8 ) | b;
}
}

```


(B-heading) VEC

The VEC format is a simple native vector file format used by DiffCad. Data in the file is integer ASCII data. There are no other markers in the file. There are two types of use in the format: POINT and VECTOR. First here is the POINT type:

Points are stored as ASCII decimal numbers and are comprised of two coordinates, x and y that are separated by spaces or tabs. Points are separated by newlines (linefeeds). Here is a file describing a 3 pixel square.

```
0 0
3 0
3 3
0 3
```

Storing separate points can be useful to describe shapes and objects. Edge detection (see Chapter 9) of an image can create this sort of output. Also, this format may be used for computer vision and vector display applications.

The VECTOR type of data is specified in a similar manner, except that two points are defined per line as follows:

```
x1 y1 x2 y2
x3 y3 x4 y4
...
```

Here (x_1, y_1) defines the tail of a vector and (x_2, y_2) defines the head:

(x_1, y_1) (x_2, y_2)

Begin NOTE

Diffcad has a routine to convert from xy points to vectors.

End NOTE**(B-heading) PICT**

The PICT format is a Macintosh metafile format. It can incorporate both bitmap and vector data. Diffcad can write PICT vector data only (the reader is referred to the savepict.java class). The PICT format is a fairly complex format and few details will be shown here. PICT can use two different forms of compression: JPEG and PackBits. PackBits is an RLE type of encoding scheme.

(A-heading) Summary

Digital image file formats are defined by several characteristics: the type of file format it is (vector, bitmap or other), the size efficiency based on the compression technology it uses, the number of colors it can handle and the resolution of images that it supports. Another factor useful for Internet based graphics is progressive display, which is the

display of a partial image as a graphics file is downloaded. Three formats have specific provisions for this : GIF, PNG and JPEG. Although the universe of graphic file formats is very large there is a great deal of similarity between formats of the same type. One bitmap format is likely to be as capable as another. Metafile formats provide the capability for vector and bitmap representations in a single format. Higher levels of abstraction are available in vector formats ultimately leading to representation of 3D objects, scenes and worlds in 3D formats. DiffCad supports GIF, JPEG, PPM, VEC, and a subset of PICT.

(CN) 9. Image Processing in Java

The worth of a book is to be measured by what you can carry away from it.

-James Bryce

Save the mandrills, collect the whole set.

- DL

This chapter covers the computation of the histogram of an image. We follow this with a derivation of the basis for the 2D fast Fourier transform (2D FFT) and a summary of a class that implements the 2D FFT. We show how to use the FFT to perform high and low pass filtering.

The high-pass filtering of the FFT is used to create edges. These edges are linked using a raster to vector converter. The raster to vector converter inputs an edge-detected image and outputs a series of line segments that may be drawn to the screen and saved as a pict file.

Finally we cover color-space conversions and elementary 2D rotation and scaling.

(A-heading) The Histogram

The histogram of an image is the probability mass function (PMF) of the pixel intensities. The probability mass function shows the statistical frequency of occurrence for an event. Thus, the computation of the PMF is obtained by counting the number of times an event (a particular intensity) occurs in the data, then dividing by the total number of pixels in the image.

For example, suppose an image consists of a 1-D array given by:

$$[255 \ 255 \ 128 \ 64] \quad (9.1)$$

The PMF is computed by counting the total number of times a particular event occurs, then dividing by the total number of elements. The array is then expressed as an event with its associated PMF number:

$$\begin{bmatrix} event & PMF \\ 64 & 1/4 \\ 128 & 1/4 \\ 255 & 1/2 \end{bmatrix} \quad (9.2)$$

(BEGIN NOTE) The PMF is a discrete probability distribution function (PDF). Like the PDF, the PMF numbers will always sum to one. (END NOTE). Naturally, the image arrays are much larger than that given in (9.1). Further, the array list shown in (9.2) is typically shown as a bar chart. An example histogram is shown in Figure 9.1.

Figure 9.1. An Example Histogram

The histogram of Figure 9.1 is shown for 255 intensities in red, green, blue (RGB) and intensity (I). The intensity is computed by averaging the RGB components and truncating to the nearest integer. The Histogram frame extends the PictFrame, and in doing so, is able to save the histogram as a pict file (for editing). An example is shown in Figure 9.2.

Figure 9.2. A section of the Histogram frames' pict output.

(BEGIN NOTE) Once the histogram is saved as pict, the fonts may be changed so that they are no longer bit-mapped. In this case, they are selected as Times Roman. (END NOTE)

A code fragment from the Histogram class in the lyon.ipl package shows how to implement the display of Figure 9.1:

```
package lyon.ipl;
import java.awt.*;
import java.awt.image.*;
import gui.*;
import lyon.dclap.*;
class Histogram extends PictFrame {

    public int red[] = new int[256];
    public int green[] = new int[256];
    public int blue[] = new int[256];
    public int intensity[] = new int[256];
```

The constructor for the Histogram class takes a PixelPlane instance as an argument,

```
public Histogram(PixelPlane p) {
```

... Histogram uses the RGB values to act as indices into three arrays of 256 integers.

```
for (int i=0; i<tp; i++) {
    red[p.getRed(i)]+=1;
    green[p.getGreen(i)]+=1;
    blue[p.getBlue(i)]+=1;
}
```

The Histogram constructor then computes the intensity by taking the truncated average of the three colors.

```
for (int i=0; i<intensity.length; i++) {
    intensity[i] = (red[i]+green[i]+blue[i])/3;
}
```

The rest of the code is devoted to normalization and display. (BEGIN CDROM) The full source code is available on the book's CDROM. (END CDROM)

(A-Heading) The 2D DFT

Recall the 1D DFT from chapter 6:

$$V_k = \frac{1}{N} \sum_{j=0}^{N-1} e^{-2\pi ijk/N} v_j \quad (6.4).$$

and that the inverse DFT is given by:

$$v_j = \sum_{k=0}^{N-1} e^{2\pi ijk/N} V_k \quad (6.10).$$

Suppose that we have a 2D array of uniformly sampled and quantized data:

$$f(x, y)$$

where

$$x \in [0..W-1], y \in [0..H-1]$$

The terms W and H are the width and height of the image, in pixels.

The 2D discrete fourier transform (2D DFT) is given by

$$F(u, v) = \frac{1}{WH} \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} f(x, y) e^{-2\pi i(ux/W + vy/H)} \quad (9.1)$$

The inverse 2D DFT (2D IDFT) is given by:

$$f(x, y) = \sum_{v=0}^{H-1} \sum_{u=0}^{W-1} F(u, v) e^{2\pi i(ux/W + vy/H)} \quad (9.2)$$

(BEGIN NOTE) The $(1/WH)$ term is not present in the IDFT. Nor is the negative sign in the exponent. (END NOTE) We introduce a notation (following [Gonzalez et al.]), that uses a symbol called the double arrow, to shorten the expressions in (9.1) and (9.2) to:

$$f(x, y) \Leftrightarrow F(u, v) \quad (9.2a)$$

In order to turn the 2D DFT into a 2D FFT, we use the *separability* property to break the 2D DFT into two fast 1D FFTs. The exponential functions separability is due to the laws of exponents, namely, $e^a e^b = e^{(a+b)}$.

To employ the separability property, we factor (9.1) into:

$$F(u, v) = \frac{1}{H} \sum_{y=0}^{H-1} \left[\frac{1}{W} \sum_{x=0}^{W-1} f(x, y) e^{-2\pi i u x / W} \right] e^{-2\pi i v y / H} \quad (9.3)$$

Similarly, we factor the 2D IDFT from (9.2) into:

$$f(x, y) = \sum_{v=0}^{H-1} \left[\sum_{u=0}^{W-1} F(u, v) e^{2\pi i u x / W} \right] e^{2\pi i v y / H} \quad (9.4)$$

The separability property means that the 2D DFT may be computed by finding the 1D DFT on the rows of the image, then finding the 1D DFT on the columns. This means that we can use our 1D DFT (and FFT) code from Chapter 6 to help perform our 2D FFT. Our implementation of (9.3) transforms each row, placing the outcome in a complex array whose dimensions match that of the original image. Then we transform each column of the complex array. Thus, the 1-D DFT is performed on each row, then again on each column.

Recall the centering of Chapter 6 that used

$$v_k = v_k (-1)^k \quad (6.30)$$

to cause a shift in the psd. This comes about as a result of the time-shift theorem that states that in the frequency domain, spatial translation causes an added linear phase with slope proportional to the shift so that

$$f(x - x_0, y - y_0) \Leftrightarrow F(u, v) e^{-2\pi i(x_0 u / W + y_0 v / H)} \quad (9.5)$$

The dual of (9.5) is

$$F(u - u_0, v - v_0) \Leftrightarrow f(x, y) e^{2\pi i(u_0 x / W + v_0 y / H)} \quad (9.6)$$

Where the left-hand side of (9.5) is the Fourier transform of the right-hand side (using double arrow notation). Thus, a positional shift in the input plane causes a phase shift in the output plane. To put it another way, a positional shift in the time domain causes a phase shift in the frequency domain. This makes sense if we think of a sine wave being shifted in time. Relative to the unshifted sine wave, the shifted sine wave has a different phase. We represent a phase shift by multiplying by the complex exponential. To center the frequency on the 2D DFT, we shift the frequency bins by $W/2$ and $H/2$. Thus, (9.6) becomes:

$$F\left(u - \frac{W}{2}, v - \frac{H}{2}\right) \Leftrightarrow f(x, y) e^{2\pi i(Wx/(2W) + Hy/(2H))} \quad (9.7)$$

which simplifies to

$$F\left(u - \frac{W}{2}, v - \frac{H}{2}\right) \Leftrightarrow f(x, y) e^{\pi i(x+y)} \quad (9.8)$$

by Euler's relation, $e^{i\theta} = \cos \theta + i \sin \theta$, and since x, y are integers, we get:

$$e^{\pi i(x+y)} = \cos(\pi(x+y)) + i \sin(\pi(x+y)) = (-1)^{x+y} \quad (9.9)$$

Substituting (9.9) into (9.8) results in:

$$F\left(u - \frac{W}{2}, v - \frac{H}{2}\right) \Leftrightarrow f(x, y) (-1)^{x+y} \quad (9.10)$$

All 2D FFT's (described in this book) are centered using (9.10). A proof of the time-shift theorem for the 1D continuous case follows.

$$F[v(t - t_d)] = \int_{-\infty}^{\infty} v(t - t_d) e^{-i2\pi t} dt$$

let $\gamma = t - t_d$ and $dt = \gamma$ so that

$$F[v(t - t_d)] = \int_{-\infty}^{\infty} v(\gamma) e^{-i2\pi(\gamma + t_d)} d\gamma$$

so that

$$F[v(t - t_d)] = e^{-i2\pi t_d} \int_{-\infty}^{\infty} v(\gamma) e^{-i2\pi\gamma} d\gamma$$

Q.E.D.

We can take a similar approach in the 2D discrete time domain to prove (9.5):

$$f(x - x_0, y - y_0) \Leftrightarrow F(u, v) e^{-2\pi i(x_0 u/W + y_0 v/H)} \quad (9.5)$$

Invoking the definition of the continuous Fourier transform (9.5) yields:

$$F(f(x - x_0, y - y_0)) = \iint f(x - x_0, y - y_0) e^{-2\pi i(ux/W + vy/H)} dx dy \quad (9.7)$$

Let

$$X = x - x_0 \text{ and } Y = y - y_0 \quad (9.8)$$

so that

$$dX = dx \text{ and } dY = dy$$

By substitution we obtain

$$F(f(x - x_0, y - y_0)) = \iint f(X, Y) e^{-2\pi i(u(x+x_0)/W + v(y+y_0)/H)} dXdY$$

$$F(f(x - x_0, y - y_0)) = e^{-2\pi i(ux_0/W + vy_0/H)} \iint f(X, Y) e^{-2\pi i(ux/W + vy/H)} dXdY$$

from which

$$f(x - x_0, y - y_0) \Leftrightarrow F(u, v) e^{-2\pi i(x_0u/W + y_0v/H)} \quad (9.5)$$

follows.

Q.E.D.

Further usage and optimization details are discussed in the following section.

(A Heading) The FFTPlane Class

The FFTPlane class resides in the lyon.ipl package. It provides an object-oriented 2D color FFT and IFFT service. An instance of the FFTPlane class may be created from an instance of the PixelPlane class. The FFTPlane class treats all images as color and is not smart about conserving memory when working with achromatic images. The FFTPlane class makes a copy of the pixels in the PixelPlane instance. The copy is stored internally using complex red, green and blue arrays of float type.

(BEGIN NOTE) It is important to set an FFTPlane instance to null in order to reclaim the memory used when an FFTPlane instance is done. (END NOTE)

The FFTPlane class has the ability to multiply each of its internally complex numbers in the frequency domain by real numbers stored in a PixelPlane instance. This enables the implementation of 2D filters.

(B heading) Class Summary

```
package lyon.ipl;
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import gui.*;
import VS.*;
public class FFTPlane {
    public FFTPlane(ProcessPlane ppIn)
    public void mult( ProcessPlane ppIn)
    public void fft()
    public void ifft()
}
```

(B heading) Class Usage

Suppose the following variables are predefined:

```
FFTPlane fftp;
ProcessPlane pp; // pp is a 2**n by 2**n image
ProcessPlane filter;
```

Then to make an instance of the FFTPlane, use:

```
fftp = new FFTPlane(pp);
```

(BEGIN NOTE) The ProcessPlane instance is altered by the methods in the FFTPlane instances. (END NOTE)

To perform an in-place FFT on the instance of the fftp:

```
fftp.fft();
```

The results of the `fft` method are left in internal data-structures. The `ProcessPlane` instance is altered to show the log of the `psd`.

To perform a multiplication, pixel by pixel, from the real pixels in *filter*, replacing the complex pixels in the `fftp` instance:

```
fftp.mult(filter);
```

(BEGIN NOTE) The filter instance should be the same dimensions as the `fftp`. Also, the pixels should vary from 0 to 255. They will be divided by 255 (so as not to increase the magnitude of the `fftp` result) before they are multiplied. The multiplication works with color filters (i.e., any `ProcessPlane` that has colors in it). (END NOTE)

To invoke the an IFFT on an instance of the `FFTPPlane` and place the result in the original `ProcessPlane` instance, `pp`:

```
fftp.ifft();
```

(BEGIN NOTE) This destroys the original `ProcessPlane` instance, `pp`. (END NOTE)

In the following section, we show how to retro-fit the `ProcessPlane` class so that any instance of the `ProcessPlane` can support the `fft()`, `mult(pp)` and `ifft()` invocations.

(B heading) The `ProcessPlane` implementation

We have retrofitted the `ProcessPlane` class with an `fft` method. We hold that this is the preferred way to perform a 2D FFT, leaving the `FFTPPlane` defined for those who would like the ability to extend the `FFTPPlane`'s abilities. The `ProcessPlane` is retrofitted as follows:

```
private FFTPlane fftp;
public void fft() {
    System.out.println("Running the FFT...");
    fftp = new FFTPlane(this);
    fftp.fft();
}
public void multFFT(ProcessPlane pp_) {
    if (fftp == null) {
        fft();
    }
    else {
        fftp.mult(pp_);
    }
}
public void ifft() {
    System.out.println("Running the iFFT...");
    if (fftp == null) {
        System.out.println("You must take the FFT
first!");
    }
    else
        fftp.ifft();
}
```

The `ImageFrame` class has been updated to permit multiplication by images that are stored in the `childFrame`. The update for the `ImageFrame` class follows:

```
public void fft() {
```

```

        pp.fft();
        updateDisplay(pp);
    }
    public void multFFT() {
        pp.multFFT(childFrame.pp);
        updateDisplay(pp);
    }

    public void ifft() {
        pp.ifft();
        updateDisplay(pp);
    }
}

```

(B heading) DiffCAD and the Example 2D FFT's

Some examples of Fourier Transform pairs are shown in the following Figures. On the left is $f(x,y)$. On the right is $F(u,v)$. The results are obtained using the DiffCAD program. To improve the visibility of the psd, we auto-scale and invert the psd. Figure 9.3 shows one and two squares painted with a bit-mapped paint program called Debabelizer [Debabelizer].

Figure 9.3 A Square and Its psd.

Figure 9.4 shows a black ring (which acts as a notch filter) with its psd.

Figure 9.4. A black ring and its psd

These are used as input. To have a little more control than exists in most paint programs, we have retrofitted the PixelPlane class with a *dot* method. It permits the generation of a filled circle (like that shown in Figure 9.5) with a specified radius.

Figure 9.5 A black dot with its psd.

The code for the dot method follows:

```

    public void dot(int xc, int yc, int r) {

        int x1 = 0;
        int y1 = 0;
        int x2 = getWidth();
        int y2 = getHeight();
        float r2 = r * r;
        for (int x = x1; x < x2; x++)
            for (int y = y1; y < y2; y++) {
                if (((x-xc)*(x-xc) + (y-yc)*(y-yc)) < r2)
                    setPixel(x,y,0,0,0,255);
                else
                    setPixel(x,y,255,255,255,255);
            }
    }
}

```

To synthesize diffraction gratings (a research goal of the DiffCAD program), we add a lines method to the ProcessPlane whose prototype is given by:

```

    public void lines(Rectangle r, int increment)

```

Figure 9.6 shows an odd shaped circle, with its psd.

Figure 9.6. An odd-shaped circle with its FFT

Figures 9.7 to 9.10 show psds multiplied by bi-level images. After the multiplication is performed, an IFFT is taken. This enables the construction of images that can be used as filters in the frequency domain.

Figure 9.7. Atriangle, with its FFT

Figure 9.8. mandrill, Its FFT

Figure 9.9. , A filter that removes some of the low frequencies from the mandrill and the IFFT

Figure 9.10. A filter that removes some of the high frequencies from the mandrill and the IFFT

Another neat trick we can perform with the FFT, is Fraunhofer diffraction. Fraunhofer diffraction is a special case of Fresnel diffraction and occurs when either the distance from the grating is large or when the rules in the grating are close together. Fraunhofer diffraction is used when performing X-ray diffraction for studying underlying crystal-like structures (like DNA). The DiffCAD program uses Fraunhofer diffraction to perform diffraction-based rangefinding [DeWitt and Lyon]. The following formula (whose derivation appears in [Walker]) shows the relationship between the DFT and Fraunhofer diffraction:

$$I(u, v, D) \approx \left| \frac{1}{\lambda D} \hat{A} \left(\frac{u}{\lambda D}, \frac{v}{\lambda D} \right) \right|^2 \quad (9.11)$$

Where λ is the wavelength of light, $I(u, v, D)$ is the intensity from Fraunhofer diffraction, and D is the distance between the target and the grating. \hat{A} is the 2-D Fourier transform of the aperture (i.e., the grating):

$$\hat{A}(u, v) = \iint_{-\infty}^{\infty} A(x, y) e^{-2\pi i(ux+vy)} dy dx \quad (9.12)$$

Figure 9.11, with its associated FFT, is offered as an example of a far field diffraction image computed by taking an FFT.

Figure 9.11. A grating with associated FFT

An increment paramter is input using the scale dialog box, and indicates the number of pixels to be skipped between lines. An example of the ProcessPlane being used is given in the ImageFrame class:

```
public void lines() {
    pp.lines(bounds(), scaleKonst.getValue());
    updateDisplay(pp);
}
```

A variable inter-rule width grating (called a chirp grating) can give a radically different diffraction pattern from a fixed with grating. An example is shown in Figure 9.12.

Figure 9.12. A chirp grating with FFT

(BEGIN NOTE) For the diffraction images shown, $\lambda D = 1$. If $\lambda = 638$ nm (about the wavelength of a He-Ne laser) then $D = 1/\lambda \approx 1.5$ million meters.(END NOTE) Based on the assumption that the light is far, the FFT can assist us directly in the computation of (9.12). At modest distance and with a small aperture, relative to D , we can use the Fresnel diffraction formula:

$$I(u, v, D) \approx \left| \frac{1}{\lambda D} \iint_{-\infty}^{\infty} A(x, y) e^{\frac{\pi i}{\lambda D} [(u-x)^2 + (v-y)^2]} dy dx \right|^2 \quad (9.13).$$

Developing code to solve (9.13) is beyond the scope of this book. A numeric solution of (9.13) may be found in [Baker].

(A Heading) Raster to Vector Conversion

In this section we show how to take an edge detected image and turn it into a list of line segments. We take an ad-hoc approach to the conversion that runs in $O(N^2)$ time. For small numbers of N the quadratic growth does not appear to be a problem. We do not address refinements to the algorithm.

Converting a list of point coordinates into vectors (i.e., line segments) has applications in the graphic arts, CAD and computer vision. In the graphic arts we use the vectors to form an editable outline of an image. The outline may be reproduced without the jaggies inherent in bit-mapped images. In CAD we use the vectors to help construct a geometric model. In computer vision, we use the vectors to obtain geometric features that can be used in recognition. Another application for raster to vector conversion is in the output of line data using a vector-based device. For example, a pen plotter that uses a mechanical arm to deflect a pen must have the raster data converted to vector for efficient operation. Vector display systems (like laser-based mirror deflection systems) form another class of devices where the raster to vector conversion process is needed. In fact, in addition to converting from raster to vector, we must also order the vectors end-to-end. Ordering the vectors end-to-end will speed plotting on a mechanical pen plotter by minimizing the pen-up time. A pen plotter will typically lift a pen off of the drawing surface before attempting to draw. A vector display device (like an oscilloscope or laser) will typically use blanking signal to turn off the beam. If the vector list is not retraced frequently enough, the human visual system will experience the sensation of flicker. Ordering the vectors will maximize the number of vectors to be displayed without flicker.

The process of traversing the vector list in minimum time is a similar problem to that faced by a mail carrier starting out from a post office and delivering letters to each block with minimum walking. This is called the *Chinese postman problem*. See [Roberts] for a discussion of the Chinese postman problem.

Finally, one application that would appear to be of critical need in this day and age of network communications is in data compression. If we could transmit a geometry faster than a bit-mapped rendering of the geometry, then we could render the geometry at any resolution using a Java program running on the client. For example, suppose that we wanted to distribute a popular test pattern (like color bars). We could render the test pattern at various resolutions and then download them over the net, but this would be very wasteful of bandwidth and local data storage. Color bars consist of 11 rectangles of various sizes and colors. A 640x480 image of color bars could take over 300 k bytes of memory. The program needed to store and display 11 rectangles takes less than 100 bytes of memory (and is much more flexible!).

Before the raster to vector conversion process can start, we assume that we have good edges. Edge detection is a tuff problem and may be accomplished in one of several ways. We have already seen how to perform edge detection using a high-pass filter created with an FFT. In Chapter 7 we saw how a domain-specific edge detector can find the centroid

of a thick edge. Sometimes we need to combine methods, like the auto-scale and negate methods with a threshold. In this section, we assume that a good edge-detected image is used as input. The goal is to take the edge detected image and create a vector description of the image. Figure 9.13 shows the output of the edge detector, and the result of a raster to vector conversion. The file was saved as line segments to a pict file.

Figure 9.13 Raster to Vector converter

The basic idea behind raster to vector conversion is that a list of integer coordinates of the form:

```
x1 y1
x2 y2
```

Should be used as input. The output consists of a series of vectors of the form:

```
x1 y1 x2 y2
x3 y3 x4 y4
```

The idea is that the points used as input will exhibit intra-frame coherence. We exploit this coherence by connecting the dots (the white-pixels). The success of the algorithm depends on many points satisfying a *criterion of adjacency*. The criterion of adjacency is used to determine when two points are next to one another.

Typically, we say that two points are next to one another if they lie within a circle whose radius is one pixel. If two points are next to one another, then they can be used to form a line segment that is exactly two pixels long. The line segment consists of a head, a tail and a slope. The compression ratio depends on the quality of the image edge detection, the amount in intra-frame coherence and the criterion of adjacency. Using a randomly selected GIF image, found on the net, we were able to take 1314 points (edge detected pixels) and create 51 editable lines (a 25:1 compression ratio). When converting an image with lines in it (like a diffraction grating) the compression ratio goes up to 132:1. Further, our slope tolerance was very tight, so there was no introduced distortion. Our experience with different images indicates that it is difficult to generalize about the expected compression ratios.

(B Heading) A raster to vector algorithm

In this section we present an algorithm that runs in $O(N^2)$, where N is the number of input points. We start with a set of points in a point list, `pl`. Our objective is to create a set of vectors, `v`. The following code is excerpted from the `Xy2vec` class in the `lyon.ipl` package:

```
public static void main(String args[]) {
    Vector v = new Vector();
    PointList pl = new PointList();
    Xy2vec x = new Xy2vec();
```

The following line will read the points from an input file and place the data into the point list, `pl`:

```
x.readPoints(pl);
```

The point list is treated like a stack, where the `popPoint` method returns a point from the top of the stack:

```
Points p = pl.popPoint();
```

`Points` is a point list data type, a class in the `lyon.ipl` package that we cover in a later section. In the following while loop, we check to make sure that the point list is not

empty. Our objective is to place the point in one of a list of vectors. If we cannot, we make a new vector:

```
while (p != null) {
    boolean point_not_stashed = true;
    // Empty vector ?
    if (v.size() == 0) {
```

The Vec class is another class in the `lyon.ipl` package. An instance of a Vec consists of a head, a tail and a slope. When a Vec instance is first created the tail is null. To keep track of this, we provide an flag called, `tail_is_empty`. We set this to true whenever we make a new Vec instance. To complicate matters somewhat, the list of Vec instances is stored in an instance of the Vector class, `v`.

```
        Vec W = new Vec(p);
        W.tail_is_empty = true;
        v.addElement(W);
        point_not_stashed = false;
    } else {
```

For each Vec instance in the list of vectors, `v`, we test to see if the point is adjacent to the Vec instance. If it is, we test to see if the slope is within a tolerance. If the point is adjacent to the vectors end-points and the slope is within tolerance, we grow the vector by one point. We then declare that the point is placed in the list of vectors and move on to the next point. If we cannot place the point, we create a new vector with the orphaned point at the head.

```
        for (int i=0;
            (i<(int)v.size())&&point_not_stashed;i++) {
            Vec newV = (Vec)v.elementAt(i);
            if (newV.head.isAdjacent(p)&&
                newV.isSlopeAcceptable(p)) {
                newV.addHead(p);
                point_not_stashed = false;
            }
            // Put new point to tail
            if (point_not_stashed) {
                if ((newV.tail_is_empty &&
                    newV.head.isAdjacent(p)) ||
                    (!newV.tail_is_empty &&
                    newV.tail.isAdjacent(p)&&
                    newV.isSlopeAcceptable(p))) {
                    newV.addTail(p);
                    if (newV.tail_is_empty) {
                        newV.tail_is_empty = false;
                        newV.getSlope();
                    }
                    point_not_stashed = false;
                }
            }
        } // for
        // Put new point to head
        if (point_not_stashed) {
```

```

        Vec nv = new Vec(p);
        nv.tail_is_empty = true;
        v.addElement(nv);
        point_not_stashed = false;
    }
}

```

After we are done with the point, we proceed to the next point;

```

    p = pl.popPoint(); // Get new point
} // while
x.printOutData( v);

```

We say that the algorithm is $O(N^2)$ because in the worst-case, we will be able to create vectors that are one pixel long. This is a pathologic case create by an advisory and occurs when there is no pixel adjacency. This hardly ever happens. In fact it might be better to consider an average case of $O(V^2)$ where V is the expected number of vectors.

Computing the average case is probably only practical when the image domain is known in advance (i.e., compression ratio for text images).

(B heading) The Slope class

There are two parameters that are central in controlling the behavior of the raster to vector algorithm. The first, known as the slope tolerance, is stored in the Slope class (seen below). The second, known as the radius squared of a circle about the vector endpoints is described in the next section. The following code resides in the lyon.ipl package and is contained in the Slope.java file:

```

package lyon.ipl;
class Slope {
    public double dx, dy;

```

The slope of a vector is represented by the change in y divided by the change in x. As we attempt to extend an existing vector, we will add to a vector's endpoint only if the slope changes by less than some ϵ amount. The value for ϵ will depend on the desired compression ratio and the tolerance for loss in the compression. Lower values for ϵ will result in a higher compression ratio with greater loss.

```

    public static double eps = 0.001;

```

```

    Slope() {};

```

```

    Slope(double dx_, double dy_) {
        dx = dx_;
        dy = dy_;
    }

```

The *isEqual* method checks for ϵ difference in the slopes between the existing slope and a proposed slope.

```

    public boolean isEqual(Slope s) {
        if (s.dx == 0 && dx == 0)
            return true;
        if ((s.dx != 0) && (dx != 0))
            return (Math.abs((s.dy/s.dx) - (dy/dx)) < eps);
        else

```

```

        return false;
    }
} // class Slope

```

(B heading) The Points class

The Points class resides in the lyon.ipl package and is used to store a list of points that are used as input to the Xy2vec algorithm. A central parameter that controls the loss in the output as well as the compression ratio is the radius of a circle about a vector endpoint that determines adjacency.

To determine if two points are adjacent, we use a boolean method called *isAdjacent* which we embed into the Points class:

```

package lyon.ipl;
class Points {
    double x,y;

```

The square of the radius of the distance between two points that are judged adjacent is called *dr*

```

    double dr = 1.0;
    Points(double px, double py) {
        x = px;
        y = py;
    }

```

To decide if two points are adjacent we check the distance between them, in pixels. This distance is compared with *dr*. We can adjust *dr* to be a number greater than one, but it will introduce some geometric distortion:

```

    public boolean isAdjacent(Points p) {
        double dx = x-p.x;
        double dy = y - p.y;
        double r = dx * dx + dy * dy;

        return r <= dr;
    } // isPointsAdjacent

} // class Points

```

(A Heading) Color Models

Light energy is a form of electromagnetic radiation. As Maxwell said, It consists of waves propagated through an electromagnetic field according to electromagnetic laws [Banerjee]. A light source may radiate energy with several spectral components. To determine the spectra of a light source, an instrument called a spectrometer is used. A typical light source (like a tungsten filament in an incandescent lamp) will be heated to a temperature that causes a phenomena known as *radiancy*. Radiancy is defined as the rate per unit surface area at which energy is radiated into the forward hemisphere [Resnick]. Radiancy is typically given in watts per centimeter squared and is found by integrating the *spectral radiancy*. The spectral radiancy is found by using a spectrometer to measure the amount of energy which falls over an area at a particular wavelength.

A *cavity radiator* is an idealized heated solid that emits a radiancy that is given by:

$$R_c = \sigma T^4 \quad (9.14)$$

Where T is in degrees Kelvin and $\sigma = 5.67 \times 10^{-8} \text{ W / m}^2 \text{ K}^4$ is the Stefan-Boltzmann constant. A typical cavity radiator is made of a block of metal (i.e, tantalum, tungsten or molybdenum). The cavity is created by drilling a small hole. Radiancy is typically measured when temperatures are typically in the 1500-7000 K range. It is important to realize that the radiancy at the surface of different materials is a material property but that the radiancy of all cavity radiators is governed by (9.14) and is independent of the material.

Another attribute of the cavity radiator (which is also known as a *black-body radiator*, *full radiator* or *Planckian radiator*) is the spectral radiancy. Spectral radiancy for the Planckian radiator is given by

$$M_e = \frac{c_1}{(e^{c_2/\lambda T} - 1)\lambda^5} \quad (9.15)$$

where $c_1 = 3.74183 \times 10^{-16} \text{ Wm}^2$, and $c_2 = 1.4388 \times 10^{-2} \text{ mK}$ and M_e is in W / m^3 [Hunt].

(BEGIN NOTE) Cavity radiators all have the same spectral radiancy.(END NOTE)

To summarize, light is energy that has a spectral radiancy. We can compute the psd for light just like we do for sound. To reconstruct the light we must therefore duplicate the spectral radiancy. Such an effort may be exceedingly impractical if the objective is to build a computer display for the purpose of human vision. For the rest of this section we discuss how to reconstruct the spectral radiancy of light for the purpose of human perception. This assumption enables the construction of practical computer display systems.

The human perception of the spectral radiancy of light is called *color vision*. For the purpose of discussion, we shall use the term *color* to mean the human perception of color. Our goal is to describe a color model that can assist in reconstructing colors for humans to see.

When the objective is to reconstruct light for the purpose of the human *perception* of color then color becomes a *psychophysical* quantity. The known relationships between the psychological color experience and physical light stimulation has given rise to several conflicting theories about how the human nervous system works [Teewan].

Color sensation in the human eye is produced when light of various wavelengths fall upon the eye. Most of the color models that are in common use today are based upon the *tristimulus* theory of color perception. The theory is based on the physiological approach that attempts to explain the eyes' behaviour in terms of its components. The eye is a photosensitive sensor that contains optical elements and photoreceptor cells. There are two types of photoreceptor cells, rods and cones. These enable a spectral response for human vision. The spectral response of human vision not only varies from person to person, but also as an individual ages. Age alters the spectral response of the human eye because age causes a progressive yellowing of the lens of the eye. Thus, there is no general agreement on the spectral sensitivities of the human visual system [Cowan et al.]. Rods are responsible for low-light imaging and do not play a part in normal color vision. The rods have varying amounts of a photosensitive pigment, called *rhodopsin*. Rhodopsin absorbs light most strongly in the blue-green part of the spectrum.

Cones are divided into three types, R, G, and B. It has not yet been possible to isolate the pigments in cones [Hunt]. The cones have been found to have a logarithmic response.

Also, the spectral response for each of the cones differs in central-frequency, peak-amplitude and bandwidth [Faugeras].

The tristimulus theory of color perception says that we only need three color primaries to create the gamut of visible color. Also, an examination of the spectral response curves should enable us to approximate the colors for which the cones have a peak spectral response. These happen at about 444 nm (b), 526 nm (g) and 645 nm (b) [Cohen et al.]. (Begin NOTE) The red and green cones have a similar spectral sensitivity.(END NOTE) Typical range in the human visual system is from 380 to 770 nm. So, the RGB monitors (and systems for the storage of image data) has its roots in the human visual system. Further, the perceptual system is logarithmic in response (for both vision and hearing!). This accounts for the popularity of the logarithmic companding techniques discussed in Chapter 5.

The tristimulus theory leads to a color model that consists of a 3D color space. A color model provides a framework for color specification. There are two kinds of color synthesis, additive synthesis and subtractive synthesis.

The additive synthesis color model adds light created via radiance. The RGB color model is a common example of an additive synthesis color model. With the RGB color model, red, green and blue light are added in equal amounts to create various shades of gray. When red, green and blue light are combined in equal amounts, at maximum intensity (on, for example, a computer monitor) the color is called white.

The subtractive synthesis color model removes light energy via absorption. The cyan, yellow and magenta pigments are applied in equal amounts to make varying shades of black. Figure 9.14 shows the relationship between the RGB system and the CMY system. When colors are combined in equal amounts, the color-space points lies on the main diagonal between the white and black points on the color cube.

Figure 9.14. Additive vs. Subtractive Color Synthesis

(B Heading) The HLS System

One of the color spaces held in common use is the hue, luminance and saturation model (HLS). This model uses a cylindrical coordinate system. In a cylindrical coordinate system, you specify the magnitude of a vector as a function of the height and angle. The height is used to represent luminance. The angle is used to represent hue. The magnitude of the vector represents the saturation. The basic idea behind the conversion is that red is assigned a hue angle of 0 degrees, blue 240 degrees and green 120 degrees. Luminance and saturation vary from zero to one, while the hue varies from zero to 360 degrees.

The ImageFrame class has been modified to perform the RGB to HLS and HLS to RGB conversions. These facilities are built into the ProcessPlane (pp) instance. Their invocation follows:

```
public void rgb2hls() {
    pp.rgb2hls();
    updateDisplay(pp);
}
public void hls2rgb() {
    pp.hls2rgb();
    updateDisplay(pp);
}
```


These are methods in the ImageFrame. The ProcessPlane instance is stored in pp. Menu items have been added to the ImageFrame to make the invocation of the conversions automatic. Figure 9.15 shows the color menu from the ImageFrame.

Figure 9.15 The Color Menu in the ImageFrame

One complicating aspect of color-space computation is the use of floating-point numbers in the transformation matrices. Recall that in the ProcessPlane class, we store an image as a packed array of 32 bit ints. Such an array is unsuitable for the storage of floating-point computations. The reason why is that round-off error will build rapidly when only 8 bits are used to represent each color. To mitigate the effect of round-off error, and to facilitate other floating-point computations, we have elected to store the images as 3 arrays of float. This presents the programmer with the burden of setting instances of the color conversion classes to null in order to reclaim memory. Further, it requires that the garbage collector be explicitly invoked if out-of-memory errors occur. We have found that this is hardly ever needed as we have associated these instances with windows that are under the users' control. When a user disposes of an ImageFrame instance, all image data is reclaimed. Thus, memory errors will occur if there are too many windows open. The user must, as a result, close some windows before proceeding.

(B Heading) The IYQ System

The IYQ system was invented for color television transmission. The symbol I stands for In-phase, Y for lumenance and Q for Quadrature. The IYQ color space is used in several analog color television transmission systems including PAL (Phase Alternating Lines), NTSC (National Television Systems Committee) and SECAM (Sequentiel Couleur avec Memoire). NTSC is used in the United States, Canada, Mexico and Japan. SECAM is used in France, the former USSR and eastern Europe. PAL is used in western Europe. The IYQ system is based on the idea that the human visual system requires crisp outlines, but that it can tolerate lower color bandwidths. This system reduces color bandwidth by low-pass filtering the I and Q color components. Further, because the eye is less sensitive to magenta than to orange, the I color component (orange-cyan) is given more bandwidth than the Q color component (green-magenta). Thus there are compelling human visual reasons for using the IYQ system (even in non-analog video compression systems) [Inglis].

To transform the RGB color into the IYQ color, a 3x3 matrix multiplication is required.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.522 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (9.16)$$

To convert back from the IYQ space we use:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.522 & 0.311 \end{bmatrix}^{-1} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \quad (9.17)$$

For reasons of computational precision, we do not evaluate the matrix inverse expressed in (9.17). Some books will give the matrix and its inverse in print, with 3 significant figures of resolution [Rogers]. We can get a much more precise answer if we allow Java to perform the inverse using the precision of a float or a double. This is particularly important since we will be using a floating-point format to store our pixels.

Some images (like X-rays, MRI, and Hubble telescope pictures) contain more than 8 bits of color information per color plane. A floating-point format is ideal for storing this type of image. While we cannot help the fact that display buffers will only show 8 bits per color, we certainly do not have to introduce that limitation into our computations or our data files.

(A Heading) The FloatImage class

The FloatImage class resides in the lyon.ipl package. The purpose of this class is to turn a ProcessPlane instance into 3 arrays of floats. The floats are needed in order to perform high-precision image processing. A reference to the ProcessPlane instance that is used to create the FloatImage is kept, in internal private storage.

(B heading) Class Summary

```
package lyon.ipl;
import java.net.URL;
import java.applet.Applet;
import java.awt.*;
import java.awt.image.*;
import futils.utils.*;

public class FloatImage {
    public float r[];
    public float g[];
    public float b[];
    public FloatImage(int l)
    public FloatImage(ProcessPlane pp_)
    public ProcessPlane makeProcessPlane()
    public int getLength()
    public int getHeight()
    public int getWidth()
    public float getRed(int i)
    public float getGreen(int i)
    public float getBlue(int i)
    public float getAlpha(int i)
    public float getRed(int x, int y)
    public float getGreen(int x, int y)
    public float getBlue(int x, int y)
    public float getAlpha(int x, int y)
    public void setPixel(int x, int y, float r_, float g_,
        float b_)
    public void setPixel(int i, float r_, float g_, float
        b_)
    public void printSize()
    public float max(int i)
    public float min(int i)
}
```

(B heading) Class Usage

Suppose the following variables are predefined

```
FloatImage fi;
ProcessPlane pp;
double red[];
double green[];
double blue[];
int l;
```

To get and set the color components from a FloatImage instance:

```
red = fi.r;
green = fi.g;
blue = fi.b;
```

(BEGIN NOTE) after great deliberation, we have decided to make the rgb data structures public. The alternative is to fill code with `getArray` accessor methods. At this time, we are still not sure if this was the right design choice. Typically, accessor methods are used to keep threads synchronized. We have abandoned this approach. The result is code that is unsuitable for multi-threading, but is smaller, faster and easier to understand. (END NOTE)

To make an instance of a FloatImage with *l* pixels:

```
fi = new FloatImage(l);
```

To make an instance of a FloatImage by making a copy of a ProcessPlane instance:

```
fi = new FloatImage(pp);
```

(BEGIN NOTE) Operations performed on the FloatImage instance do not alter the ProcessPlane instances data. All elements in the ProcessPlane instance were copied and type-converted during the copy process.(END NOTE)

To turn a an instance of a FloatImage into an instance of a ProcessPlane:

```
pp = fi.makeProcessPlane();
```

To get the number of pixels in the FloatImage instance:

```
l = fi.getLength();
```

To get the height and width of the FloatImage instance:

```
int h = fi.getHeight();
int w = fi.getWidth();
```

To get the red, green, blue and alpha components in the FloatImage instance, treating all the arrays as 1D arrays:

```
int i;
float r = fi.getRed(i);
float g = fi.getGreen(i);
float b = fi.getBlue(i);
float a = fi.getAlpha(i);
```

(BEGIN NOTE) The alpha channel is never kept in the FloatImage instance. Instead, the alpha channel is converted from the ProcessPlane on demand. The reason is that no FloatImage methods alter the alpha channel, so no floating-point version of the alpha channel is needed. (END NOTE)

To get the red, green, blue and alpha components in the FloatImage instance, treating all the arrays as 2D arrays:

```
int x, y;
float r = fi.getRed(x, y);
float g = fi.getGreen(x, y);
float b = fi.getBlue(x, y);
```

```
float a = fi.getAlpha(x, y);
```

To set a floating-point pixel, using 2D coordinates:

```
float r, g, b;
int x, y;
fi.setPixel(x, y, r, g, b);
```

To set a floating-point pixel, using the 1D internal order:

```
fi.setPixel(i, r, g, b);
```

To print the size of the array to the System.out:

```
fi.printSize();
```

To find the minimum and maximum amplitude color components located at position i in the 1D internal order:

```
float intensity = fi.max(i);
float intensity = fi.min(i);
```

(A Heading) The ColorConverter class

The ColorConverter class is an abstract class that resides in the VS package. The ColorConverter class is extended with implementations required for methods that convert from and to the RGB color space. Internally, the ColorConverter class provides a storage area for the FloatImage and ProcessPlane instances.

(b heading) Class Summary

```
package VS;
import java.awt.*;
import java.io.*;
import lyon.ipl.*;
public abstract class ColorConverter {
    public ProcessPlane pp;
    public FloatImage fi;
    public ColorConverter(ProcessPlane pp_)
    public abstract int[] fromRGB();
    public abstract int[] toRGB();
    public FloatImage getFloatImage()
}
```

(b heading) Class Usage

A class that extends the ColorConverter class must implement the methods *fromRGB()* and *toRGB()*. A class that extends the ColorConverter class is used to convert a ProcessPlane instance (an RGB image) into an another color-space encoded FloatImage instance. The IYQ class is an example of a class that extends the ColorConverter class:

```
package lyon.ipl;
import VS.*;
public class IYQ extends ColorConverter {
```

Recall that the RGB to IYQ conversion requires a matrix multiplication given by (9.16):

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.522 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (9.16)$$

An implementation of (9.16) follows:

```
double A[][] = {
```

```

    { 0.299, 0.587, 0.114},
    { 0.596, -0.274, -0.322},
    { 0.211, 0.522, 0.311}
};

```

To create instances of a matrix that knows how to invert itself and multiply itself by other matrices we use the `Mat3` class. We shall discuss the `Mat3` class in the following section.

```
Mat3 rgb2iyqMat = new Mat3(A);
```

Performing the matrix inversion in Java is done at instantiation. The results are much more precise than those found in most publications:

```
Mat3 iyq2rgbMat = rgb2iyqMat.invert();
```

The constructor calls the super class, `ColorConverter`, which copies the `ProcessPlane` instance into a new `FloatImage` instance. This constructor is required of any class that is to extend the `ColorConverter`.

```

public IYQ ( ProcessPlane pp_ ) {
    super(pp_);
}

```

The following two methods are typical of any that implement a color-space conversion using a 3x3 matrix multiplication. This happens to be quite common with color-space conversions, but there are several that do not perform the matrix multiplication of (9.16) (i.e., polar coordinate color systems like HVS and HLS).

```

public int[] fromRGB() {
    double pel[];
    float r, g, b;
    for (int i=0; i < fi.getLength(); i++) {
        pel =
    rgb2iyqMat.multiply(fi.r[i],fi.g[i],fi.b[i]);
        fi.r[i] = (float) pel[0];
        fi.g[i] = (float) pel[1];
        fi.b[i] = (float) pel[2];
        pp.setPixel(i, (int) pel[0], (int) pel[1],
(int)pel[2], 255);
    }
    return(pp.pels);
}

public int[] toRGB() {
    double pel[];
    for (int i=0; i < fi.r.length; i++) {
        pel = iyq2rgbMat.multiply(fi.r[i],fi.g[i],
fi.b[i]);
        fi.r[i] = (float) pel[0];
        fi.g[i] = (float) pel[1];
        fi.b[i] = (float) pel[2];
        pp.setPixel(i, (int) pel[0], (int) pel[1],
(int)pel[2], 255);
    }
    return(pp.pels);
}

```

}
(A heading) The Mat3 Class – Maple, the Java Gin Joint

The Mat3 class is a public class that resides in the lyon.ipl package. The purpose of the Mat3 class is to provide an optimized way to multiply non-sparse 3x3 matrices. It also acts as a test-bed for experiments in using Maple (a symbolic manipulator) to generate optimized Java code.

We have found that the code generated by Maple is both human readable and optimized for speed of execution. When used correctly, the code is also found to be error-free. Maple is able to ease the programmer's burden but only in some cases. In general the code that Maple generates requires human-manipulation before it becomes usable as Java.

(B heading) Class Summary

```
package lyon.ipl;
public class Mat3 {
    public Mat3 (double a[][])
    public double [] [] getArray()
    public Mat3 invert()
    public Mat3 multiply(Mat3 bmat3)
    public double[] multiply(double v1, double v2, double v3
    )
    public void print()
}
```

(B heading) Class Usage

There is an example of class usage in the IYQ class. The Mat3 class is a public class that resides in the lyon.ipl package. It is an optimized class designed for multiplying 3x3 matrices by each other or by 3x1's. It is designed for color-space conversion, but is generally applicable to other tasks. The Mat3 class also supports a fast 3x3 matrix inversion. Suppose that the following variables are pre-defined:

```
double A[][] = {
    { 0.299, 0.587, 0.114},
    { 0.596, -0.274, -0.322},
    { 0.211, 0.522, 0.311}
};
Mat3 m3;
Mat3 m3inverse;
```

To make a new instance of Mat3 use:

```
m3 = new Mat3(A);
```

To find the inversion of an instance of Mat3:

```
m3inverse = m3.invert();
```

To get the double precision 3x3 array stored in a Mat3 instance:

```
A = m3.getArray();
```

To multiply two Mat3 instances:

```
Mat3 identity = m3.multiply(m3inverse);
```

(BEGIN NOTE) Floating-point error prevents the Mat3 instance of the identity matrix from being exact, but it is close. (END NOTE)

To multiply the 3x3 in Mat3 by a 3x1 to obtain a 3x1 array of double (also optimized), use:

```
double r, g, b;
```

```
double v3[] = m3.multiply(r, g, b);
To print a Mat3 instance to the System.out PrintStream:
m3.print();
```

(B heading) Maple: The Java Gin Joint

Maple is a language for symbolic manipulation [Char et al.]. Maple V was used to generate optimized Java code for implementing Matrix manipulation in the Mat3 class.

This is an interesting twist in Java that we have not seen in any other Java books.

The cotton gin is one of the symbols of the beginning of the industrial revolution. Before the industrial revolution, all work was performed by hand, and assembly-line methods were unknown. Perhaps, software is in its preindustrial-revolution days too. We have yet to formulate automatic methods for the generation of software, except in special cases.

The area of matrix manipulation is one of those cases (and there are few others).

In Maple, the following code will generate the Java source needed to generate a time-optimal 3x3 matrix inversion:

```
1. with(linalg):
2. readlib(C):
3. a =array(0..2,0..2,[]):
4. b =array(0..2,0..2,[]):
5. b =inverse(matrix(a)):
6. C(b,optimized);
```

The lines are numbered for reference only. Line 1 reads in the linear algebra package into Maple. Line 2 reads in the C-language generator. Lines 3 and 4 create arrays whose index starts at zero. Line 5 converts the *a* array into a *matrix* type, forms the inverse symbolically and sets the answer to *b*. Line 6 outputs the following code:

```
public Mat3 invert() {
    double b[] [] = new double [3][3];
    double t4 = a[0][0]*a[1][1];
    double t6 = a[0][0]*a[1][2];
    double t8 = a[0][1]*a[1][0];
    double t10 = a[0][2]*a[1][0];
    double t12 = a[0][1]*a[2][0];
    double t14 = a[0][2]*a[2][0];
    double t17 =
        1/(-t4*a[2][2]+t6*a[2][1]+t8*a[2][2]-t10*a[2][1]-
t12*a[1][2]+t14*a
[1][1]);
    b[0][0] = -(a[1][1]*a[2][2]-a[1][2]*a[2][1])*t17;
    b[0][1] = -(-a[0][1]*a[2][2]+a[0][2]*a[2][1])*t17;
    b[0][2] = (-a[0][1]*a[1][2]+a[0][2]*a[1][1])*t17;
    b[1][0] = (a[1][0]*a[2][2]-a[1][2]*a[2][0])*t17;
    b[1][1] = (-a[0][0]*a[2][2]+t14)*t17;
    b[1][2] = -(-t6+t10)*t17;
    b[2][0] = (-a[1][0]*a[2][1]+a[1][1]*a[2][0])*t17;
    b[2][1] = -(-a[0][0]*a[2][1]+t12)*t17;
    b[2][2] = (-t4+t8)*t17;

    return new Mat3(b);
}
```

}
 We find this pretty handy. Maple can automatically unwind for-loops, generate auxiliary variables and save a lot of typing. The declaration of double for the temporary variables (ttn) that Maple generates must be done by hand.

The technique of using Maple extends well into other matrix operations. In Maple, for example, we can unwind the for loops in a multiplication using:

`C(multiply(matrix(b),matrix(a)),optimized):`
 The optimized output is reformatted, annotatted and appears below:

```
public Mat3 multiply(Mat3 bmat3) {
double WW [][] = new double[3][3];
double b [][] = bmat3.getArray();
    WW[0][0] =
a[0][0]*b[0][0]+a[0][1]*b[1][0]+a[0][2]*b[2][0];
    WW[0][1] =
a[0][0]*b[0][1]+a[0][1]*b[1][1]+a[0][2]*b[2][1];
    WW[0][2] =
a[0][0]*b[0][2]+a[0][1]*b[1][2]+a[0][2]*b[2][2];
    WW[1][0] =
a[1][0]*b[0][0]+a[1][1]*b[1][0]+a[1][2]*b[2][0];
    WW[1][1] =
a[1][0]*b[0][1]+a[1][1]*b[1][1]+a[1][2]*b[2][1];
    WW[1][2] =
a[1][0]*b[0][2]+a[1][1]*b[1][2]+a[1][2]*b[2][2];
    WW[2][0] =
a[2][0]*b[0][0]+a[2][1]*b[1][0]+a[2][2]*b[2][0];
    WW[2][1] =
a[2][0]*b[0][1]+a[2][1]*b[1][1]+a[2][2]*b[2][1];
    WW[2][2] =
a[2][0]*b[0][2]+a[2][1]*b[1][2]+a[2][2]*b[2][2];
    return (new Mat3(WW));
}
```

The process of generating Java is not totally automatic. Some human intervention is required. The method entry points and returns must be added. Most of the hard work is done by Maple. The following Maple code will generate Java code, like the above, for any N, where N is an integer greater than one:

```
readlib(linalg):
readlib(C):
multn := proc(N)
    local a, b, c, v;
    a := array(0..N,0..N);
    b := array(0..N,0..N);
    c := array(0..N,0..N);
    v := vector([seq(vec[i],i=0..N)]);
    print(v);
    c := multiply(matrix(a),matrix(b));
    print(`Mult N by N times N by N`); print(N+1);
    C(c, optimized);
    print(`Mult N by N times N by 1`); print(N+1);
end proc;
```



```

c := multiply(matrix(a),v);
C(c, optimized);
c := inverse(matrix(a));
print('inverse'); print(N);
C(c, optimized);
end;

```

The amount of time Maple takes to perform the *multn* procedure increase quadratically with increasing N . Therefore, this technique will not be practical for all sizes of N . Also, we have not optimized for specially sparse matrices, like rotation matrices that occur in computer graphics and robotics.

(A heading) Image Geometry

In this section we cover the derivation and implementation of elementary 2D image transformation using matrix multiplication. We will cover the operations of translation, rotation, scale and shear, using 3x3 matrix multiplications. To perform these operations, we shall make extensive use of the *Mat3* class, derived in the previous section.

(B heading) 2D translation

To translate a point, p , whose coordinates are $(p.x, p.y)$ plane by an amount, t , whose offset is given by, $(t.x, t.y)$, use:

$$p' = (p.x + t.x, p.y + t.y) \quad (9.17)$$

In Java notation

```
pPrime = p.add(t);
```

An implementation of this may be found in the *point* class, which resides in the *lyon* package:

```

public class point extends Computation {
    double x = 0;
    double y = 0;
    ...

    public point add(point t) {
        return new point(x + t.x, y + t.y);
    }
}

```

This is a particularly slow implementation, because a new point is allocated every time an addition is performed. We use such code for illustration, only.

(B heading) 2D scaling

To scale a point, p , whose coordinates are $(p.x, p.y)$ plane by an amount, t , whose scale is given by, $(s.x, s.y)$, use:

$$p' = (p.x * s.x, p.y * s.y) \quad (9.18)$$

To scale about a point, t , first translate to the origin, perform the scaling, then translate back. This may be represented by:

$$p' = ((p.x - t.x) * s.x + t.x, (p.y - t.y) * s.y + t.y) \quad (9.19)$$

In Java, we can express (9.19) and (9.18) by overloading a method in the *point* class:

```

public point scale(point s) {
    return new point(x * s.x, y * s.y);
}

```

```

}
In the following implementation of (9.19) we scale about point t.
public point scale(point s, point t) {
    return new point((x - t.x) * s.x + t.x, (y - t.y) *
s.y+t.y);
}

```

Since (9.18) scales about the origin, the point will move relative to the origin. Further, if scaling were uniform, the amount of scaling in each dimension would be equal.

(B heading) 2D rotation

An example of rotation about the center of an image is shown in Figure 9.16.

Figure 9.16 Lena rotated about the center of the frame

To rotate about the center of an image, first translate the center to the origin, rotate, then translate back.

To rotate a point, p , whose coordinates are $(p.x, p.y)$ plane by an amount, θ , about the origin use:

$$p' = (p.x \cos \theta - p.y \sin \theta, p.x \sin \theta + p.y \cos \theta) \quad (9.20)$$

To rotate about a point, t , first translate to the origin, perform the rotation, then translate back. This may be represented by the column vector:

$$p' = \begin{bmatrix} (p.x - t.x)(p.x \cos \theta - p.y \sin \theta) + t.x \\ (p.x - t.y)(p.x \sin \theta + p.y \cos \theta) + t.y \end{bmatrix} \quad (9.21)$$

Positive angles are measured **counterclockwise**. As an exercise, you should be able to reformulate the rotational transformations for negative angles. Use the identities:

$$\cos(-\theta) = \cos \theta$$

$$\sin(-\theta) = -\sin \theta$$

Proof of $p' = (p.x \cos \theta - p.y \sin \theta, p.x \sin \theta + p.y \cos \theta)$:

Suppose that a complex number of magnitude r represents p so that $p = re^{i\phi}$. Then rotation, with respect to the origin, by an amount of θ , can be had by multiplication via another complex number, $p_\theta = e^{i\theta}$ so that

$$p' = pp_\theta = re^{i\phi} e^{i\theta} = re^{i(\theta+\phi)} \quad (9.22)$$

by Euler's relation, $e^{i\theta} = \cos \theta + i \sin \theta$ we obtain:

$$e^{i(\theta+\phi)} = r[\cos(\theta + \phi) + i \sin(\theta + \phi)] \quad (9.23)$$

We then invoke the double angle formulas for sine and cosine to obtain:

$$e^{i(\theta+\phi)} = r \begin{bmatrix} \cos \phi \cos \theta - \sin \phi \sin \theta + \\ i(\cos \phi \sin \theta + \sin \phi \cos \theta) \end{bmatrix} \quad (9.24)$$

Recall that $p = re^{i\phi}$ is, by Eulers relation:

$$p = re^{i\phi} = r \cos \phi + ir \sin \phi = p.x + ip.y \quad (9.25).$$

Substituting (9.25) into (9.24) yields

$$p' = re^{i(\theta+\phi)} = \begin{bmatrix} p.x \cos \theta - p.y \sin \theta + \\ i(p.x \sin \theta + p.y \cos \theta) \end{bmatrix} \quad (9.26)$$

From (9.26), it follows directly that rotation about the origin is given by:

$$p' = (p.x \cos \theta - p.y \sin \theta, p.x \sin \theta + p.y \cos \theta)$$

Q.E.D.

(BEGIN NOTE) An display of the amplitude vs the phase of a waveform can be rotated by the introduction of a delay.(END NOTE)

In the Shape class we implement the point rotation about any point using a translation to the center, followed by a rotation, followed by a translation back. The Shape class resides in the lyon package. This code modifies the original point so that time does not have to be spent allocating and disposing of new point instances.

```
public void pointRotation(point p, point pc, double theta)
{
    // rotate point p about pc an amount of theta radians
    // return the modified point
    double c_theta = Math.cos(theta);
    double s_theta = Math.sin(theta);
    double tx = pc.x + (p.x - pc.x) * c_theta - (p.y -
pc.y) * s_theta;
    double ty = pc.y + (p.y - pc.y) * c_theta + (p.x -
pc.x) * s_theta;
    p.x = tx;
    p.y = ty;
}
```

The point rotation may be applied to images or to graphic objects. Figure 9.17 shows a pin-hole camera used for simulating a diffraction rangefinder in the DiffCAD program. As the camera is repositioned, it automatically pans to point toward the center of a diffraction grating.

Figure 9.17. A pin-hole camera rotated about its center of focus.

homogeneous coordinate transforms in 2D

So far we have see that translation, scaling and rotation are:

$$p' = (p.x + t.x, p.y + t.y) \quad (9.17)$$

$$p' = ((p.x - t.x) * s.x + t.x, (p.x - t.y) * s.y + t.y) \quad (9.19)$$

and

$$p' = \begin{bmatrix} (p.x - t.x)(p.x \cos \theta - p.y \sin \theta) + t.x \\ (p.x - t.y)(p.x \sin \theta + p.y \cos \theta) + t.y \end{bmatrix} \quad (9.21)$$

In order to concatenate several transforms into a single computational entity, we introduce homogeneous coordinates. This will speed the computation of combinations of serveral transformations by creating a single matrix against which points will be multiplied. For this discussion, we follow [Foley et al.].

With homogeneous coordinates we use *tuples* in 2D; (x,y,w). Also, one of the coordinates must be non-zero (typically w is non-zero).

Iff

$$P = \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$$P' = \alpha P \tag{9.27}$$

then P' and P represent the same point. That is, a linear combination of P does not alter the position of the original point. The proportionality factor, α , is eliminated by computing the Cartesian coordinates of the homogeneous point:

$$\text{Cartesian coordinates} = (x/w, y/w, 1) \tag{9.28}$$

The XYW homogeneous coordinate space, with the $w=1$ plane and point $P(x,y,w)$ projects onto the $w=1$ plane.

Homogeneous coordinate transformations in 2 space require a 3x3 matrix multiplication. (BEGIN NOTE) If all you want to do is translate, you are doing 3 multiplies and 2 adds for nothing!(END NOTE)

The matrix form for the translation is:

$$\begin{bmatrix} p'.x \\ p'.y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t.x \\ 0 & 1 & t.y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p.x \\ p.y \\ 1 \end{bmatrix} \tag{9.29}$$

Which is just like

$$p' = (p.x + t.x, p.y + t.y) \tag{9.17}$$

with the exception of the w variable. (BEGIN NOTE) Multiplication by zero in (9.29) takes the computer as long to do as the multiplication by a non-zero (unless we intercept this special case) (END NOTE)

Some graphics text books premultiply rather than postmultiply by the column vectors. To convert between the forms, use transposition:

$$\begin{bmatrix} p'.x \\ p'.y \\ 1 \end{bmatrix}^T = \begin{bmatrix} p.x \\ p.y \\ 1 \end{bmatrix}^T \begin{bmatrix} 1 & 0 & t.x \\ 0 & 1 & t.y \\ 0 & 0 & 1 \end{bmatrix}^T \tag{9.30}$$

Expanding (9.30) results in:

$$[p'.x \quad p'.y \quad 1] = [p.x \quad p.y \quad 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t.x & t.y & 1 \end{bmatrix} \tag{9.31}$$

The Java implementation of (9.29) may be found in the translation method in the Mat3 class of the lyon.ipl package:

```
public Mat3 translation(point t) {
    a = new double[3][3];
    a[0][0] = 1;
    a[1][1] = 1;
    a[2][2] = 1;
    a[0][2] = t.x;
```

```

        a[1][2] = t.y;
        return new Mat3(a);
    }

```

Suppose we write:

```

public static void main(String args[]) {
    Mat3 trans = Mat3.translation(new point(2,3));
    trans.print();
}

```

Then the output is:

```

1 0 2
0 1 3
0 0 1

```

Now we can perform the homogeneous coordinate transformations using matrix multiplications. We use this ability to concatenate many transformations into a single matrix representation. For example:

```

public static void main(String args[]) {
    Mat3 trans1 = Mat3.translation(new point(2,3));
    Mat3 trans2 = Mat3.translation(new point(1,2));
    Mat3 trans3 = trans1.multiply(trans2);
    trans1.print();
    System.out.println(" * " );
    trans2.print();
    System.out.println(" = " );
    trans3.print();
}

```

```

1 0 2
0 1 3
0 0 1
*
1 0 1
0 1 2
0 0 1
=
1 0 3
0 1 5
0 0 1

```

This is a concatenation of two transformations, trans1 and trans2, into a single transformation matrix, trans3. The matrix product is also called compounding, catenation, concatenation or composition. Once the transform is formulated, it may be applied to all points in the scene. Several policy issues must be resolved in order to implement matrix compounding:

We have allocated the memory for a transformation matrix using a static method in the Mat3 class. The memory allocation is dynamic and possibly wasteful. This might be OK, however, since the number of transformation matrices may be small, relative to the number of points that require transformation. When we are done with the transformation matrix, we should set it to null so that the garbage collector can reclaim the storage. It is probably good policy to keep a copy of the master object and apply the composite transform to all the points each time. Incrementally transforming the points on an integer

coordinate system (like an image plane) will distort the image. For example, suppose that I wanted rotate an image by 50 degrees in 10 degree increments. The progression of the image is shown in Figure 9.18.

Figure 9.18. Incremental rotation in an integral coordinate system.

Figure 9.18 shows that when the result of a rotation is used to supply data for the transform input, the result gets distorted and that error accumulates through the transforms. To reduce the distortion, we start with the original data and recompute the transformation matrix. The same 50° rotation can be had, without accumulating distortion, as shown in Figure 9.19.

Figure 9.19. Non-incremental rotation in an integral coordinate system

An interesting aspect of the distortion is the aesthetic quality of an image after the incremental rotation. Figure 9.20 show what happens to the poor mandrill after a 360° rotation is accomplished in 18 steps of 20° each.

Figure 9.20. Mandrill after 18 incremental rotations of 20° each.

The matrix form for the scaling is:

$$\begin{bmatrix} p'.x \\ p'.y \\ 1 \end{bmatrix} = \begin{bmatrix} s.x & 0 & 0 \\ 0 & s.y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p.x \\ p.y \\ 1 \end{bmatrix} \quad (9.32)$$

The Java implementation of (9.32) is:

```
public static Mat3 scaling(point s) {
    double m[][] = new double[3][3];
    m[0][0] = s.x;
    m[1][1] = s.y;
    m[2][2] = 1;
    return new Mat3(m);
}
```

Successive scalings are multiplicative, just like successive translations so that:

```
public static void main(String args[]) {
    Mat3 trans1 = Mat3.scaling(new point(2,3));
    Mat3 trans2 = Mat3.scaling(new point(1,2));
    Mat3 trans3 = trans1.multiply(trans2);
    trans1.print();
    System.out.println(" * ");
    trans2.print();
    System.out.println(" = ");
    trans3.print();
}
```

Results in:

```
2 0 0
0 3 0
0 0 1
*
1 0 0
0 2 0
0 0 1
```

```
=
2 0 0
0 6 0
0 0 1
```

Figure 9.20 shows the mandrill with a 3:1 zoom out followed by a 3:1 zoom in. This effect was first done optically by systems like the Blockpix processor [Manning]. Our empirical test show that the color gamut produced by Blockpix processors has a wider range than that produced by color monitors.

Figure 9.20. Original mandrill, Zoom out 3:1, then Zoom in

The matrix form for the rotation is:

$$\begin{bmatrix} p'.x \\ p'.y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p.x \\ p.y \\ 1 \end{bmatrix} \quad (9.33)$$

The Java implementation of (9.33) follows:

```
public static Mat3 rotation(double theta) {
    double m[][] = new double[3][3];
    double cas = Math.cos(theta);
    double sas = Math.sin(theta);
    m[0][0] = cas;
    m[1][1] = cas;
    m[0][1] = -sas;
    m[1][0] = sas;
    return new Mat3(m);
}
```

(BEGIN NOTE)

The Math.sin and Math.cos functions in Java use radians for input.

(END NOTE)

As an example:

```
public static void main(String args[]) {
    Mat3 trans3 = Mat3.rotation(90 * piOn180);
    trans3.print();
}
```

produces:

```
0 -1 0
1 0 0
0 0 0
```

The rotation and translation transforms are rigid-body transformations since they do not distort the object. Scale is not a rigid body transformation since scaling in the x and y directions may be different.

Affine transformations consist of sequences of rotation, translation, scale and shear operations. Affine transformations preserve the parallelism of lines, but not lengths or angles.

Shear can go in the x or y-direction. For example:

The shear transform in the x-direction is given by:

$$\begin{bmatrix} p'.x \\ p'.y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh.x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p.x \\ p.y \\ 1 \end{bmatrix} \quad (9.34).$$

The shear transform in the y-direction is given by:

$$\begin{bmatrix} p'.x \\ p'.y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ sh.y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p.x \\ p.y \\ 1 \end{bmatrix} \quad (9.35).$$

An implementation for the construction of a shear matrix follows:

```
public static Mat3 shear(point sh) {
    double m[][] = new double[3][3];
    m[0][0] = 1;
    m[1][1] = 1;
    m[2][2] = 1;
    m[0][1] = sh.x;
    m[1][0] = sh.y;
    return new Mat3(m);
}
```

The following shows an example of the shear method being invoked:

```
public static void main(String args[]) {
    Mat3 transl = Mat3.shear(new point(1,2));
    transl.print();
}
```

The output follows:

```
1 1 0
2 1 0
0 0 1
```

(B heading) Applications of affine transforms

In this section we show some examples of the use of affine transforms in the DiffCAD program for manipulating image data. In all the cases, the ProcessPlane class has been modified to permit the use of a 3x3 matrix transform to process the output coordinates of the image. The output coordinates are multiplied by the matrix transform, centered on the image coordinates. The result of the multiplication is used to resample the input image. No filtering is performed, so we can expect aliasing effects. The heart of the transformation is a method called xform:

```
public void xform(Mat3 transform) {
    int w = getWidth();
    int h = getHeight();
    int xc = w/2;
    int yc = h/2;
```

We see that the xform method takes a precomputed transformation matrix in the form of an instance of the Mat3 class. The height and width are obtained from the input image. The output image is stored in an instance of the ProcessPlane, pp:

```
ProcessPlane pp = new ProcessPlane(w,h);
```


The result of the 3x3 matrix multiplication by a 3x1 is a 3x1. The result is stored in an array called p:

```
double p[] = new double [3];
int pixel;
int xp, yp;
```

We follow the idea of [Expeset] and start to scan the output image using coordinates that bias the center of the image toward the origin. This permits rotation about the center of the image, rather than the origin (the upper left corner of the image in the Java AWT).

```
for (int x = -xc; x < xc; x++)
  for (int y=-yc; y < yc; y++) {
    p=transform.multiply(x,y,1);
    xp = (int) p[0]+xc;
    yp = (int) p[1]+yc;
    if ((xp < w) && (yp < h) && (xp >= 0) && (yp >= 0)) {
      pixel = getPixel(xp, yp);
      pp.setPixel(x+xc,y+yc,pixel);
    }
  }
pels = pp.pels;
```

The xform method permits implementation of any of the transforms by creating a 3x3 matrix and calling xform. For example, the ProcessPlane method, *turn*:

```
public void turn(double degrees) {
  double pion180 = Math.PI / 180.0;
  double theta = degrees * pion180;
  xform(Mat3.rotation(theta));
}
```

Another example is the ProcessPlane method, *zoom*:

```
public void zoom(double percentage) {
  xform(Mat3.scaling(new point(percentage,percentage)));
}
```

Finally we implement the shear in x and y using:

```
public void shearx(double shx) {
  xform(Mat3.shear(new point(shx,0)));
}
public void sheary(double shy) {
  xform(Mat3.shear(new point(0,shy)));
}
```

The results of a shear in x and a shear in y are shown in Figure 9.21.

Figure 9.21. Shear in x and shear in y.

Instead of making a new image, we can use the input image as the canvas for our output. We can then reprocess the image, using the result. This is known as a feedback loop.

Many cool effects are based on feedback. Figure 9.22 shows zoom being used with feedback. The only modification that we need to make is to add a new xform method to the ProcessPlane, called xformfeedback. The xformfeedback method is just like the xform method except for one important difference:

```
for (int x = -xc; x < xc; x++)
```

```

    for (int y=-yc; y < yc; y++) {
        p=transform.multiply(x,y,1);
        xp = (int) p[0]+xc;
        yp = (int) p[1]+yc;
        if ((xp < w) && (yp < h) && (xp >= 0) && (yp >= 0)) {
            pixel = getPixel(xp, yp);
            setPixel(x+xc,y+yc,pixel);
        }
    }
}

```

A this this point in the code, xform would set the pixel on a ProcessPlane instance...instead xformfeedback uses its own.

Figure 9.22. Many cool effects are based on feedback

Originally analog feedback was obtained by pointing two mirrors at one another. People who frequent barbershops are typically placed in chair that have a mirror in front and a mirror behind them. The images are scaled in size and place inside one another. This feedback appears to go on forever. Electronically, the feedback may be performed by pointing a video camera at a monitor that outputs what the video camera scans. This type of processing permits video synthesizers to be placed into the loop. Now a days companies like Quantel and Ampex produce video equipment that is able to produce effects like this in real-time [Schwartz et al.]. Figure 9.23 shows Lena after rotational feedback is performed (the process of evolution into the form shown is actually more interesting than the final form itself). To implement the feedback effect, we simply invoke the feedback version of the xform method:

```

public void turnfb(double degrees) {
    double pion180 = Math.PI / 180.0;
    double theta = degrees * pion180;
    xformfeedback(Mat3.rotation(theta));
}

```

We can combine effect sequentially, without any modification to DiffCAD. Figure 9.24 shows zoom feedback applied to the result of rotational feedback.

Figure 9.24 Rotational and zoom feedback (sequentially applied)

With a small modification, we can create a composite transform matrix that can yield composite transform feedback. The following method (called shearyfb) shows how multiplication of several transform matrices can create a concatenated transform matrix:

```

public void shearyfb(double s) {
    Mat3 t = Mat3.shear(new point(.5,.5));
    point tp = new point(1.4, 0.9);
    t.multiply(Mat3.scaling(tp));
    t.multiply(Mat3.rotation(20*Math.PI/180));
    xformfeedback(t);
}

```

The result of this effect is shown in Figure 9.25.

Figure 9.25 Composite transform feedback

As a final thought, we propose a theory that allows for the clear distinction of visual computer art. We claim that art is a language of communication and that, like any language, art forms have grammar. For example, camera grammar might be; dolly, pan,

tilt and rotate. Thus we claim that computer art is different from other art forms in that it contributes new effects for image manipulation (i.e., the affine feedback transform).

(A Heading) Summary

In this chapter we showed how to take a histogram, perform an FFT, IFFT and to use these transforms to filter an image. One application that we explored was the creation of diffraction in the far-field. This was made possible by some simplifying assumptions that made far-field diffraction computationally identical with the FFT.

We also introduced an algorithm for raster to vector conversion. This is an essential first step in extracting linear features from an image. The vectors still require ordering (the Chinese Postman problem). Space and time do not permit us to explore a solution, though we do have one [Lyon 85].

In this chapter we have explored affine transforms in their typical use, take an input image, process it, place it in an output frame. We showed how to make a simple variation on the affine transform by using feedback.

This simple variation on the use of the affine transforms makes for a wonderfully rich aesthetic exploration (i.e., eye candy). The rotational feedback about the center has a polar-coordinate symmetry appears to lead to decentralized eye-movement. Such patterns appear to have an almost mandala-like effect. We theorize that decentralized eye movement assists the mind in achieving the meditative state. Time did not permit us to develop a nice interface to our image manipulation system. This might be a topic of future exploration. One aspect of the 2D image effects, that we have not been able to convey in the figures, is the aesthetic aspects of the image evolution. During small incremental steps, intermediate results have been saved to create fascinating experimental animations that permit an exploration into the grammar of affine feedback.

As another whirlwind chapter comes to a close, it is doubly sad, as it marks the end of the book and there is so much left to do! To explore image warping further, the reader is guided to [Wolberg]. To further research into 2D image effects, [Holtzman] provides a unique perspective.

Image processing is an huge topic and there are so many good books to choose from. We have been looking to [Pratt] and [Myler] for inspiration and guidance.

$V(t)$ 297, 302
 μ -law 314
 μ -law CODEC 313
“Write Once, Run Anywhere”™ 8
.i.Java, C, C++ -> HTML 278
2D DFT 486
2D FFT 483, 495
3x3 matrices 524
3x3 matrix 515
3x3 matrix transform 545
A-Law 314
action 188, 196
Ada 17
addCheckBoxes 214
Adding Checkboxes to Frames 213
Adding Labels to Frames 224
addItem 225
addNoise 388
addNotify 169, 188, 196, 225
Affine transformations 543
affine transforms 545
ALGOL 13
Ampex 548
applet 29
Arabic 119
arglist 57
Arithmetic Encoding 454
ASCII 462
ASCIIByteArray 124
Asymetrix SuperCede 35
AU File 309
Audio 104, 295
Audio Files 303
AudioData 304, 307, 308
AudioData Class 307
AudioDataFromTable 330
AudioDataStream 304, 308, 309
AudioDataStream Class 308
AudioFrame 325, 337
AudioPlayer 304
AudioPlayer Class 312
AudioStream 304, 305, 306
AudioStreamSequence 304, 311
available 265
Avatars 442
average power 300
AWT 148

Backus Naur Form 50
Baker 499
Banerjee 509
Bart Kosko 439
Bartlett window 393
benchmark 26
Benchmarking 352
BenchMarking the DFT 354
Bengali 119
Binary PCM 315
bitExpression 56
bitmap formats 440
bitr 66
Bjarne 2
black 149
black-body radiator 510
blue 149
Boolean 50
Borg 232
bounds 186
Boyle 408
bpp 445
brighter 150
Button 184
byteArray 165
ByteArrayInputStream 308
bytesWidth 165
C++ 278
Cameras 407
Canada 514
Canvas 184
Cartesian coordinates 536
castingExpression 56
Cat.fileToStream 273
Cat.javasToFile 273
Cavity 510
cavity radiator 509
CCD 408
Centering the FFT 402
Cesáro 390
Chan and Lee 148
Chan and Lee] 28
chaos 298
Char 526
Character 50
character 58
Character.isDefined 119
Character.isDigit 119

- Character.isJavaLetter 119
- Character.isJavaLetterOrDigit 119
- Character.isLetter 119
- Character.isLetterOrDigit 119
- Character.isLowerCase 119
- Character.isSpace 119
- Character.isTitleCase 119
- Character.isUpperCase 119
- charArray 165
- charsWidth 165, 166
- charWidth 165, 166
- Checkbox 184
- Checkbox Class 211
- checkImage 187
- Chinese postman problem. 500
- chirp grating 498
- Choice 184, 225
- Choice Class 224
- class ReadPPM 464
- Class Usage 156
- classDeclaration 53
- className 57
- clearRect 155, 158, 159
- clipRect 155, 157
- Clocks in 51
- CLUT 446
- CMY 149, 512
- COBOL 17
- CodeWarrior 36
- Color Class 148
- Color Depth 445
- color look-up table 445
- Color Models 509
- color vision. 510
- ColorConverter class 520
- companding 315
- compilationUnit 52
- Component 148, 190, 211
- component = locate(x, y) 195
- Component Class 184
- Component Hierarchy 184
- component.addNotify() 196
- component.checkImage 195
- component.createImage 194
- component.deliverEvent(anEvent) 195
- component.disable() 191
- component.enable() 190
- component.enable(cond) 191

- component.getFontMetrics(font) 193
- component.getGraphics() 193
- component.hide() 191
- component.imageUpdate 194
- component.inside 195
- component.isVisible 190
- component.layout() 193
- component.move(x, y) 192
- component.nextFocus() 196
- component.paint(graphics) 193
- component.paintAll(graphics) 193
- component.postEvent(anEvent) 195
- component.prepareImage 194
- component.print(graphics) 194
- component.printAll(graphics) 194
- component.removeNotify() 196
- component.repaint() 193
- component.repaint(milliseconds) 193
- component.repaint(milliseconds, x, y, width, height) 194
- component.repaint(x, y, width, height) 194
- component.requestFocus() 196
- component.reshape(x, y, width, height) 192
- component.resize(aDimension) 192
- component.resize(width, height) 192
- component.setBackground(aColor) 192
- component.setFont(font) 192
- component.setForeground(aColor) 191
- component.show() 191
- component.show(cond) 191
- component.toString() 196
- component.update(graphics) 193
- component.validate() 193
- ComponentPeer 185
- compression 502
- Compression Methods 450
- Concurrent Euclid 18
- cones 511
- constructorDeclaration 53
- Container 148, 184
- Container Class 197
- context switch 132
- ContinuousAudioDataStream 304
- convertToHtml 280
- convolution 301
- Cooley-Tukey 361
- copyArea 155, 158
- counterclockwise 533
- countItems 225

- Cowan 511
- crash-proof 10
- createImage 187
- creatingExpression 57
- criterion of adjacency. 501
- CROSSHAIR_CURSOR 203
- cyan 149
- cylindrical coordinate system 513
- Danielson-Lancoz 363
- Danielson-Lanczos Lemma 361
- darker 150
- darkGray 149
- Data Types
 - Abstract Classes and Methods 102
 - addElement 128, 129
 - AppletFrame 90
 - AppletUtil 90
 - array types 82
 - Arrays 127
 - boolean 82, 117
 - Camera_grating_line 92
 - Casting 98
 - Character 118
 - char[] 124
 - class types 82
 - ClassCastException 98
 - classDeclaration 88
 - Classes 87
 - className 88
 - concat 125
 - Constants 86
 - constructor 92
 - Data Types 82
 - deep_array 128
 - Double 121, 123
 - endsWith 126
 - fieldDeclaration 89
 - Float 121, 123
 - getBytes 126
 - identifier 88
 - indexOf 126
 - integer 82, 122
 - interface types 82
 - interfaceName 88
 - lastIndexOf 127
 - long 123
 - modifier 88
 - new Vector() 128

- Null 97
- numeric 82
- numeric wrapper classes 121
- Object 124
- Overloaded Methods 92
- primitive types 82
- reference types 82
- Shape 92
- startsWith 126
- Static Methods 94
- String 121, 124
- String.equals 126
- String.indexOf 126
- String.lastIndexOf 127
- String.valueOf 124
- Strings 123
- Subclassing 98
- substring 125
- toLowerCase 125
- toString 125
- toUpperCase 125
- valueOf 122
- Vectors 128
- Wrapper classes 116
- DataInputStream 271, 274
- DataInputStream Class 269
- DataOutputStream 274, 275
 - DataOutputStream(is) 277
 - flush() throws IOException 275
 - size() 276
 - write(byte b[], int off, int len) 275
 - write(int b) 275
 - writeBoolean(boolean v) throws IOException 275
 - writeByte(int v) throws IOException 275
 - writeBytes(String s) throws IOException 276
 - writeChar(int v) throws IOException 276
 - writeChars(String s) throws IOException 276
 - writeDouble(double v) throws IOException 276
 - writeFloat(float v) throws IOException 276
 - writeInt(int v) throws IOException 276
 - writeLong(long v) throws IOException 276
 - writeShort(int v) throws IOException 275
 - writeUTF(String str) throws IOException 276
- DataOutputStream Class 274
- DataViz 279
- Date 137, 167
- DCT 455
- Debabelizer 495

- decimalDigits 58
- Decimation in time 363
- default 109
- DEFAULT_CURSOR 203
- Delphi 17
- DEM 406
- Devanagari 119
- Developer Environments 2
- DeWitt and Lyon 497
- DFT 344, 497
- dialog
 - addNotify() 234
 - getTitle 234
 - getTitle() 235
 - isModal() 234
 - isResizable() 234
 - setResizable(boolean resizable) 234
 - setTitle(String title) 234
 - setTitle(title) 235
- Dialog Class 233
- Dialogs in the ImageFrame 419
- DiffCAD 80, 105, 279
- diffraction 497
- Digital Elevation Map 406
- digital image warping 406
- Digital Oscilloscope. 202
- Digital Signal Processing 295
- digital-to-analog converter 296
- DigitalThread 136
- Dimension 137
- Dirac delta function 302
- Discrete Cosine Transform 455
- Discrete Fourier Transform 345
- dispose 156, 204
- distanceInPixels 167
- docComment 53
- doStatement 55, 72
- dot 496
- Double 50
- DoubleDataProducer 334, 335
- DoubleDataProducer Interface 332
- DoubleDialog 414
- DoubleGraph Class 342
- draw a grid 162
- Draw a String with a Background 167
- Draw a Vertical String 168
- draw3DRect 155, 159
- drawArc 155, 160

- drawBytes 156, 161
- drawChars 156, 161
- drawImage 156, 161
- drawLine 155, 158, 163
- drawOval 155, 159
- drawPolygon 155, 160
- drawRect 159
- drawRoundRect 155, 159
- drawString 156, 161
- drawVerticalString 168
- drum scanner 407
- DualTraceOscopeFrame 344
- dum_constants 113
- DXF 448
- Electronic scanning 407
- Embree 397
- enable 169, 186
- enumerate 145
- Euler 346, 533
- Euler's 299
- Event
 - ACTION_EVENT 173
 - ALT_MASK 171
 - AudioFrame 177
 - controlDown 173
 - CTRL_MASK 171
 - DOWN 171
 - END 171
 - Event 173
 - Event Handling 176
 - Evt Class 179
 - Evt.match 180
 - F1 171
 - F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12 175
 - F10 171
 - F11 172
 - F12 172
 - F2 171
 - F3 171
 - F4 171
 - F5 171
 - F6 171
 - F7 171
 - F8 171
 - F9 171
 - getKeyboardShortCut 181
 - GOT_FOCUS 173
 - HOME 171

Keyboard 176
keyDown 177
KEY_ACTION 172
KEY_ACTION_RELEASE 172
KEY_EVENT, SHIFT_MASK, CTRL_MASK, META_MASK, ALT_MASK,
HOME, END, PGUP, PGDN, KEY_PRESS, KEY_RELEASE, KEY_ACTION,
KEY_ACTION_RELEASE 175
KEY_PRESS 172
KEY_RELEASE 172
LEFT 171
LIST_DESELECT 172
LIST_EVENT, LIST_SELECT, LIST_DESELECT 176
LIST_SELECT 172
LOAD_FILE 173
LOST_FOCUS 173
match 179
matchKey 179
metaDown 173
META_MASK 171
MISC_EVENT, ACTION_EVENT, LOAD_FILE, SAVE_FILE, GOT_FOCUS,
LOST_FOCUS 176
modifiers 174
Mouse 182
mouseDown 182
mouseDrag 183
mouseEnter 183
mouseExit 183
mouseMove 183
mouseUp 183
MOUSE_DOWN 172
MOUSE_DRAG 172
MOUSE_ENTER 172
MOUSE_EVENT, MOUSE_DOWN, MOUSE_UP, MOUSE_MOVE,
MOUSE_ENTER, MOUSE_EXIT, MOUSE_DRAG 176
MOUSE_EXIT 172
MOUSE_MOVE 172
MOUSE_UP 172
PGDN 171
PGUP 171
pick device 182
RIGHT 171
SAVE_FILE 173
SCROLL_ABSOLUTE 172
SCROLL_EVENT, SCROLL_LINE_UP, SCROLL_LINE_DOWN,
SCROLL_PAGE_UP, SCROLL_PAGE_DOWN, SCROLL_ABSOLUTE 176
SCROLL_LINE_DOWN 172
SCROLL_LINE_UP 172
SCROLL_PAGE_DOWN 172

- SCROLL_PAGE_UP 172
- shiftDown 173
- SHIFT_MASK 171
- Target 177
- Tognazzini's 182
- translate 173
- UP 171
- UP, DOWN, LEFT, RIGHT 175
- WINDOW_DEICONIFY 172
- WINDOW_DESTROY 172
- WINDOW_EVENT, WINDOW_DESTROY, WINDOW_EXPOSE,
WINDOW_ICONIFY, WINDOW_DEICONIFY, WINDOW_MOVED,
WINDOW_EVENT 175
- WINDOW_EXPOSE 172
- WINDOW_ICONIFY 172
- WINDOW_MOVED 172
- Event Class 170
- Exceptions 130
- Expeset 546
- exponentPart 58
- expression 55
- ExtendedArabic 119
- eye 511
- E_RESIZE_CURSOR 203
- Fairbanks 401
- far-field diffraction 550
- Fast Fourier Transform 66
- Fast Hartley Transform 66
- Faugeras 511
- Feudal Times 232
- FFT 66, 361, 483, 494
- FFT.testFFT 374
- FFTPlane 492
 - fft() 492
 - ifft() 492, 493
 - mult(ProcessPlane ppIn) 492
 - mult(filter) 493
 - new FFTPlane(pp) 492
- FFTPlane Class 491
- FHT 66
- fieldDeclaration 53, 112
- FileDialog 189
- FileInputStream 261, 262, 265
- FileInputStream Class 261
- Files
 - closeOutputStream 255
 - deleteFile 249
 - deleteWildFiles 249

- dialog.dispose() 238
- dialog.getDirectory() 237
- dialog.getFile() 237
- DirFilter 246
- File Class 239
- FileDescriptor 252
- FileDialog 235, 237
 - addNotify() 235
 - FileDialog(Frame parent, String title) 235
 - FileDialog(Frame parent, String title, int mode) 235
 - getDirectory() 235, 236
 - getFile() 235
 - getFilenameFilter() 236, 237
 - getMode() 235, 236
 - LOAD 235
 - SAVE 235
 - setDirectory(String dir) 235
 - setFile(path) 237
 - setFile(String file) 235
 - setFilenameFilter(FilenameFilter filter) 236
 - setFilenameFilter(filter) 237
- FileFilter Class 247
- FilenameFilter 239
 - canRead() 240
 - canWrite() 240
 - delete() 240
 - equals(Object obj) 240
 - exists() 240
 - File(absPath) 241
 - File(dirFile,fileName) 241
 - File(dirName,fileName) 241
 - File(File dir, String name) 240
 - File(String path) 239
 - File(String path, String name) 239
 - getAbsolutePath() 240
 - getName() 240
 - getParent() 240
 - getPath() 240
 - hashCode() 240
 - isAbsolute() 240
 - isDirectory() 240
 - isFile() 240
 - lastModified() 240
 - length() 240
 - list() 240
 - list(FilenameFilter filter) 240
 - mkdir() 240
 - makedirs() 240

- pathSeparator 239
- pathSeparatorChar 239
- renameTo(File dest) 240
- separator 239
- separatorChar 239
- toString() 240
- FilenameFilter interface 245
- FileOutputStream 253
- FileOutputStream Class 252
- fos.write(b) 253
- fos.write(bytes) 253
- Futil.closeOutputStream 254
- Futil.getDirFile 245
- Futil.getFileOutputStream 254
- Futil.getReadFile 245
- Futil.getReadFileName 237
- Futil.getWriteFile 245
- Futil.getWriteFileName 238
- futils 246
- getFileOutputStream 255
- getWildNames 249
- getWriteFileName 254
- INCLUDE 250
- lower case 250
- Ls.deleteFile 244
- Ls.deleteWildFile 249
- Ls.getDirName 243
- Ls.getWildNames 248
- Ls.lowerFileNames 250, 251
- Ls.wildToConsole 249
- Ls.WordPrintMerge 249
- OutputStream 255
- PICT 250
- recursivly traverses the file system 250
- WildFilter Class 247
- WordPrintMerge 250
- write 252
- fill3DRect 155, 159
- fillArc 155, 160
- fillOval 155, 160
- fillPolygon 156, 160, 161
- fillRect 155, 158
- fillRoundRect 155, 159
- FilterInputStream 306
- FilterOutputStream 275
- Final Classes and Methods 103
- Fitz-Greene Halleck 295
- Float 50

FloatImage
 b[] 516
 FloatImage(int l) 516
 FloatImage(ProcessPlane pp_) 516
 getAlpha(int i) 517
 getAlpha(int x, int y) 517
 getBlue(int i) 517
 getBlue(int x, int y) 517
 getGreen(int i) 517
 getGreen(int x, int y) 517
 getHeight() 516
 getLength() 516
 getRed(int i) 517
 getRed(int x, int y) 517
 getWidth() 517
 g[] 516
 makeProcessPlane() 516, 519
 max(int i) 517
 min(int i) 517
 printSize() 517
 r[] 516
 setPixel(int i, float r_, float g_, float b_) 517
 setPixel(int x, int y, float r_, float g_, float b_) 517
FloatImage class 516
floatLiteral 58
floatTypeSuffix 58
FontMetrics 164
Fonts 445
Formats 278
forStatement 55
forwardFFT 375, 376
Fourier analysis 298
Fourier coefficients 298
Frame Class 203
France 514
Franklin 406
Fraunhofer 497
Fraunhofer diffraction 497
Frequency shifting using the FFT 399
Fresnel diffraction 497
fromRGB 523
Futil 232
Futil.readDataFile 286
Futil.available 265
Futil.getFileInputStream 263
Futil.makeTocHtml 259
Futil.Print 288
Futil.writeFilteredHrefFile 290

- futils 232
- futils.bench 367
- futils.DirList 294
- futils.DirList class 265
- futils.Timer Class 352
- G.711 314
- Gehani 18
- Geometry 288
- George E. Smith 408
- getAscent 164, 166
- getBackground 186
- getBlue 150
- getCheckboxGroup 212
- getClipRect 157, 158, 163
- getColor 150, 154, 157
- getColorModel 186
- getComponents 199
- getDescent 164, 166
- getFileInputStream 263, 264
- getFileOutputStream 273
- getFont 154, 158, 164, 186
- getFontMetrics 136, 154, 155, 158, 165, 167, 187
- getForeground 186
- getGreen 150
- getHeight 164, 166
- getHSBColor 150
- getIconImage 204
- getItem 225
- getLabel 169, 211
- getLeading 164, 166
- getLineIncrement 216
- getMaxAdvance 165
- getMaxAscent 164, 166
- getMaxDecent 165
- getMaxDescent 166
- getMaximum 215
- getMinimum 215
- getOrientation 215
- getPageIncrement 216
- getParent 185, 190
- getPeer 185
- getRed 150
- getRGB 150
- getSelectedIndex 225
- getSelectedItem 225
- getState 212
- getSystemThreadGroup 146
- getTitle 204

- getToolkit 185
- getTruncatedDoubleData 383
- getUlawData 319
- getValue 215
- getVisible 216
- getWidths 165
- getWriteFileOutputStream 288
- GIF 278, 447, 461
- GIF87a 457
- GIF89a 458
- Gosling 17
- gotFocus 188, 196
- grammar 130
- grapher.Graph 367
- Graphics Class 154
- Graphics Formats 447
- gray 149
- Gray wedge 408
- green 149
- GridLayout 210
- GUI 148
- Gujarati 119
- Gurmukhi 119
- Hamming 390
- handleEvent 188
- HAND_CURSOR 203
- Hanning 367, 391
- harmonic analysis 298
- He-Ne laser 408
- height 137
- Hi-pass filter 396
- histogram 483, 484
- HLS System 512
- Holtzman 551
- Holzmann 406
- homogeneous coordinate transforms 535
- homogeneous point 536
- HORIZONTAL 215
- HSB 149
- HSBtoRGB 150
- hsbvals 153
- HTML 2, 33, 278, 295
- HTML Converter 278
- HTML generator 105
- HTML Generator Panel 209
- HTML Model 32
- HtmlGenerator 106, 295
- hue 513

- Huffman 450
- Huffman Encoding 454
- human perception 510
- Hunt 511
- hyper-linked 295
- identifier 59
- IDFT 344, 487
- IFFT 380, 495
- Image 421
 - flush() 421
 - getGraphics() 421
 - getHeight(ImageObserver observer) 421
 - getProperty(String name, ImageObserver observer) 421
 - getSource() 421
 - getWidth(ImageObserver observer) 421
 - UndefinedProperty = new Object() 421
- Image Class 420
- Image Formats 439
- Image Geometry 530
- Image Instancing 424
- Image Processing 405
- Image Processing in Java 483
- ImageObserver 423
 - ABORT 424
 - ALLBITS 424
 - ERROR 424
 - FRAMEBITS 424
 - HEIGHT 423
 - PROPERTIES 424
 - SOMEBITS 424
 - WIDTH 423
- ImageProducer 190
- imageUpdate 187
- Imports 107
- importStatement 53, 108
- Indic 119
- InputStream 306
- IntDialog 414, 418, 420
- Integer 50
- integerLiteral 58
- interfaceDeclaration 53, 112
- interfaceName 58, 112
- Interfaces 112
- Inverse DFT 356
- isAdjacent 504
- isEnabled 169, 185
- ISO-LATIN-1 119
- isResizable 205

- isShowing 185
- isSlopeAcceptable 504
- isValid 185
- isVisible 185
- IYQ 514
- iyq2rgbMat 522
- James Bryce 483
- Japan 514
- Java 278
- java.awt.Frame(String) 208
- java.lang 123
- java.lang.ThreadGroup 143
- Javaholic 107
- JFIF 459
- Joseph de Fourier 298
- JPEG 278, 447, 459
- JVM 34
- Kannada 119
- Kelvin 509
- keyDown 104, 188, 196
- keyUp 188, 196
- Kodak 408, 448
- Kona 6
- Label 184
- Label Class 222
- Lao 119
- Laplace distribution 317
- layout 186
- leading 166
- Lempel 452
- lightGray 149
- linalg 529
- linear scanning array 408
- Linotype 164
- List 184
- listFilteredHrefFile 287
- literalExpression 57
- log2 375
- logicalExpression 56
- Long 50
- lossy 450
- lostFocus 188, 196
- low-pass filter. 296
- ls -al */* 265
- ls -al */* >foo 265
- luminance 513
- Lyon window 394
- lyon.ipl 419, 431, 516

- LZW 452
- m-law 307
- MacLinkPlus 279
- magenta 149
- MainMenuBar 380
- makeBartlett 393
- makeHanning 367, 391
- makeTocHtml 260
- Malayalam 119
- mandrills 483
- Maple 394, 524, 526
- Maple code 529
- Mat3 522, 524, 537
 - getArray() 524, 525
 - invert() 524, 525
 - Mat3 (double a[][]) 524
 - multiply(double v1, double v2, double v3) 524
 - multiply(m3inverse) 525
 - multiply(Mat3 bmat3) 524
 - print() 524
- Mat3 Class 524
- Maxwell 509
- MAX_RADIX 120
- MAX_VALUE 120
- MBNF 50
- mechanical scanning 407
- MenuComponent 169
- MenuItem Class 168
- method_declaration 53
- Mexico 514
- milliseconds in a day 135
- MIME 32
- minimumSize 186
- MIN_RADIX 120
- MIN_VALUE 120
- Modified Backus Naur Form 50
- modifier 57, 108
- molybdenum 509
- Moore 404
- mouseChoice 227
- mouseDown 188, 195
- mouseDrag 188, 195
- mouseEnter 188, 196
- mouseExit 188, 196
- mouseMove 188, 195
- mouseUp 188, 195
- MOVE_CURSOR 203
- multFFT 495

- multHanning 392
- Murray 443
- Myler 402, 551
- NamedObservable 411
 - getName() { 412
 - setName(String nm) { 412
- National Television Systems Committee 514
- Natural Intelligence 45
- Netravali 455
- new Date 137
- New York Tribune 164
- nextFocus 188
- NE_RESIZE_CURSOR 203
- Noise filter using the FFT 386
- NTSC 514
- NumberFormatException 417
- Numeric Check of the DFT and IDFT 359
- numericExpression 56
- NW_RESIZE_CURSOR 203
- Nyquist 296
- N_RESIZE_CURSOR 203
- Observable 410, 411
 - addObserver(Observer o) 411
 - countObservers() 411
 - deleteObserver(Observer o) 411
 - deleteObservers() 411
 - hasChanged() 411
 - notifyObservers(Object arg) 411
- Observable Class 411
- ObservableDouble 413, 419
 - getValue() { 414
 - notifyObservers() 413
 - setChanged() 413
 - setValue
(double newValue) { 413
- ObservableDouble 412
- Observer Interface 409
- Observer-Observable Example 412
- Ole-Johan Dahl 11
- optimized output 528
- orange 149
- orientation 215
- Oscillator
 - getAM 323
 - getDuration 323
 - getFM 323
 - getFrequency 323
 - getSampleRate 323

- getSawWave 323
- getSineWave 323
- getSquareWave 323
- getTriangleWave 323
- setModulationFrequency 323
- setModulationIndex 323
- Oscillator Class 322
- OscopeFrame 337
- OscopeFrame Class 333
- Package small 107
- packageName 57, 105
- Packages 105
- packageStatement 52, 105
- PAL 514
- Palettes 445
- Panel 148
- Panel Class 209
- parallelism 132
- parameter 54
- parameterList 54
- paramString 169
- Parzen 390
- Pascal 49
- PCM 296
- PCM decoder 296
- PDA's 5
- peer 189
- perception 510
- Phase Alternating Lines 514
- photoreceptor 511
- Photoshop 409
- physiological 511
- picoJava 1, 6
- PICT 447, 482
- PictFrame 333
- pink 149
- PixelPlane 427, 431
 - getAlpha(int x, int y) 428
 - getBlue(int i) 427
 - getBlue(int x, int y) 428
 - getGreen(int i) 427
 - getGreen(int x, int y) 427
 - getHeight() 427
 - getLength() 427
 - getPixel(int i) 428
 - getPixel(int x, int y) 428
 - getRed(int i) 427
 - getRed(int x, int y) 427

- getWidth() 427
- inrange(int x, int y) 427
- makeImage() 427
- MakePixel(int r, int g, int b, int a) 428
- printSize() 428
- setPixel(int x, int y, int pel) 428
- setPixel(int x, int y, int r, int g, int b, int a) 428
- PixelPlane Class 426
- Planckian radiator 510
- playAsync 319
- playSync 319
- PMF 484
- PNG 456
- point 531
- Points
 - isAdjacent(Points p) { 508
- Points class 507
- polymorphism 103, 129
- poor mandrill 540
- Power Spectral Density 345
- PPM 278, 447, 461
- Pratt 406, 551
- preferredSize 186
- prepareImage 187
- pressure waves 295
- Print 289
- printAll 187
- PrintStream 255, 288
 - checkError() 256
 - close() 256
 - flush() 256
 - print(boolean b) 256
 - print(char c) 256
 - print(double d) 256
 - print(float f) 256
 - print(int i) 256
 - print(long l) 256
 - print(Object obj) 256
 - println() 256
 - println(boolean b) 256
 - println(char c) 256
 - println(char s[]) 256
 - println(double d) 256
 - println(float f) 256
 - println(int i) 256
 - println(long l) 256
 - println(Object obj) 256
 - println(String s) 256

- public void print(char s[]) 256
- public void print(String s) 256
- write 255
- write(byte b[], int off, int len) 255
- PrintStream Class 255
- printThreadGroups 146
- private 109
- private protected - 109
- probability mass function 484
- ProcessPlane 431, 493, 496, 514, 547
 - cornergray() { 431
 - diagGray() 431
 - edge method 437
 - edge() 431
 - linearComb(a, b) 434
 - linearComb(double konstD, double akD) 431
 - makeGray() 431, 434
 - negate method 435
 - negate() 432
 - randResample() 432, 434
 - scale(int scale) { 431
 - Shadow method 436
 - shadow() 432, 434
 - shearx(double shx) { 547
 - sheary(double shy) { 547
 - shearyfb(double s) { 549
 - Subimage(pp2) 434
 - Subimage(ProcessPlane pp) 431
 - threshold(int konst) 431
 - turn(double degrees) { 546
 - xformfeedback. The xformfeedback method is just like the xform method except for one important difference 548
 - zoom(double percentage) { 547
- ProcessPlane Class 431
- Progressive Display 455
- Prolog 51
- protected 109
- PSD 300
- PSD Computations 377
- psychological 510
- psychophysical 510
- public 109
- Pulse Code Modulation 296
- Quadrature 514
- Quantel 548
- Quantization 296
- RaceThread 135
- RADAR 407

- Radiancy 509
- range image 406
- raster to vector algorithm 502
- Raster to Vector Conversion 499
- RatFOR 13
- readAUFFile 310
- readBoolean 270, 272
- readByte 270, 272
- readChar 270, 272
- readDataFile 286
- readDouble 270
- readFloat 270, 272
- readFully 270, 271
- readInt 270, 272
- readlib 529
- readLine 270
- readLong 270, 272
- readShort 270, 272
- readUnsignedByte 270, 272
- readUnsignedShort 270, 272
- readUTF 270, 273
- real-time codec 374
- real_dumb 113
- rectangular window 398
- recurses on all directory names 268
- Recursive File Lister 265
- red 149
- removeNoise 388
- removeNotify 188
- requestFocus 188
- Resampling 401
- reshape 186
- Resnick 509
- reverseFFT 371
- RGB 148
- rgb2hls 513
- rgb2iyqMat 522
- RGBtoHSB 150
- RLE 450, 451
- Robert Frost 232
- Roberts 500
- rods 511
- Rogers 515
- rotation 532
- RTF 33, 278
- Run Length Encoding 451
- sampling 301
- saturation 513

- sawtooth 298
- scaling 531
- scaling labels 339
- Scanners 407
- Schwartz 548
- Scrollbar 184
- Scrollbar Class 215
- SECAM 514
- select 225
- separability 487
- setBackground 186
- setCheckboxGroup 212
- setColor 154, 157
- setFont 154, 186
- setForeground 186
- setIconImage 204
- setLabel 169, 212
- setLineIncrement 216
- setMenuBar 204
- setPageIncrement 216
- setPaintMode 154, 157
- setPrintStream 289
- setResizable 205
- setState 212
- setTitle 204
- setUlawData 319
- setValue 215
- setXORMode 154, 157
- SE_RESIZE_CURSOR 203
- shear 544
- show 186
- Simula 11
- skip 272
- skipBytes 270
- Slope
 - isEqual(Slope s) { 507
- Slope class 505
- SNR 317
- Sound 295
- Sound blaster 305
- Spectra 298
- Spectral Leakage of the DFT 389
- spectral radiancy. 509
- spectrum 298
- startAtThisDir 267
- statement 54
- statementBlock 54
- staticInitializer 54

Stefan-Boltzmann 509
stop() 136
StreamTokenizer 281, 282, 294
 commentChar(int ch) 282
 collsSignificant(boolean flag) 283
 lineno() 283
 lowerCaseMode(boolean fl) 283
 nextToken() throws IOException 283
 ordinaryChar(int ch) 282
 ordinaryChars(int low, int hi) 282
 parseNumbers() 283
 pushBack() 283
 quoteChar(int ch) 282
 resetSyntax() 282
 slashSlashComments(boolean flag) 283
 slashStarComments(boolean flag) 283
 StreamTokenizer.TT_EOF 284
 StreamTokenizer.TT_EOL 284
 toString() 283
 TT_EOF 282
 TT_EOL 282
 TT_NUMBER 282
 TT_WORD 282
 whitespaceChars(int low, int hi) 282
 wordChars(int low, int hi) 282
string 58
stringExpression 56
stringWidth 165, 166
Stroustrup 3, 11
subsampling 402
sun.audio 304
switchStatement 55
SW_RESIZE_CURSOR 203
syntax 49, 60
 additive_operators 63
 assignment_operators 65
 bitExpression 68
 bitwise_AND_operator 64
 bitwise_OR 64
 bitwise_XOR 64
 Break 80
 break_statement 80
 Comments 59
 conditional_operator 64
 continue 80
 Continue 78
 continue_statement 79
 creation_operators 63

- equality_operators 64
- expression 67
- Expressions 67
- Flow of Control 66
- For 77
- forStatement 77
- identifier 61
- Identifiers 61
- If 68
- ifStatement 68
- keyword 61
- logicalExpression 67
- logical_AND 64
- logical_OR 64
- MBNF 59
- multiplicative_operators 63
- numericExpression 67
- Operators 62
- postfix_operators 63
- relational_operators 64
- Return 81
- return_statement 81
- shift_operators 64
- stringExpression 68
- Switch 73
- switchStatement 73
- Syntax 59
- testingExpression 67
- unary operators 62
- unary_operators 63
- variableDeclaration 77
- variableDeclarator 77
- variableInitializer 77
- While and do statements 72
- systemThreadGroup 142, 145
- S_RESIZE_CURSOR 203
- Tamil 119
- TargetControlPanel 210
- Teevan 510
- television 441
- Telugu 119
- testDFT 359
- testingExpression 56
- testPSD 378
- TextComponent 184
- TEXT_CURSOR 203
- Thai 119
- The FontMetrics Class 163

- theDate 137
- theDate.toString 167
- theFontMetrics 167
- ThreadGroup 142, 143, 145
- threadGroupsArray 142, 145
- Threads 132
- Timer 353
- tokens.nextToken() 287
- tokens.TT_EOL 287
- tokens.TT_NUMBER 287
- toLowerCase 120
- Tony Hoare 49
- toString 120, 188
- toTitleCase 120
- toUpperCase 120
- traditional typesetting 164
- transformation matrices 514
- Transforms in the AudioFrame 379
- translate 154, 158
- translation 530
- Transparency 445, 446
- tristimulus 511
- tristimulus theory 511
- tryStatement 55, 130
- turbulence 298
- type 57
- typeDeclaration 53
- typeSpecifier 57
- ulawArrayOfByte 319
- UlawCodec 309, 319, 325
- UlawCodec Class 318
- Unicode 273
- USSR 514
- UTF 273
- validate 186
- validation 30
- variableDeclaration 54
- variableDeclarator 54
- variableInitializer 54
- VEC 447, 480
- Vector formats 441
- vector formats, 440
- VERTICAL 215
- Very Low Bit Rate Voice Compression 374
- Video 407
- Visibility 110
- Visibility 108
- visibility_modifier 109

VLBRVC 374
VS package 520
WAIT_CURSOR 203
Walker 390, 497
Watson 15
wave table length 329
WaveTable 330
WEBOS 1, 4
Welch 390, 452
whileStatement 55, 72
white 149
width 137
William Wordsworth 344
windowAudio 392
windowing 390
WINDOW_DESTROY 208
Wolberg 407, 551
wrapper class 50
writeAUFFile 319, 322
writing μ -law 322
W_RESIZE_CURSOR 203
xScaleFactors 219
Xy2vec 502
yellow 149
yScaleFactors 220
ZetaLisp 2
Ziv 452