Java Programming Fundamentals

WAVE

THOMSON LEARNING™

JVPG-SSMN-0101A

Java Programming Fundamentals
JVPG-SSMN-0101A

This book is sold as is, without warranty of any kind, either express or implied, respecting the contents of this book, including, but not limited to, implied warranties for the book's quality, performance, merchantability, or fitness for any particular purpose. Neither Wave Technologies International, Inc., nor its dealers or distributors shall be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this book.

Trademarks

Trademarks and registered trademarks of products mentioned in this book are held by the companies producing them. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

The Wave Technologies International, Inc. logo is a registered trademark of Wave Technologies International, Inc., St. Louis, Missouri.

Copyrights of any screen captures in this book are the property of the software's manufacturer.

Mention of any products in this book in no way constitutes an endorsement by Wave Technologies International, Inc.

10 9 8 7 6 5 4 3 2 1

Contents

Chapter 8—Constructors 119

Chapter 9—Interfaces and Abstract Classes 145

Chapter 10—Packages and Access Modifiers 163

Chapter 11—Swing Components 177

Chapter 12—Layout Managers 211

Chapter 15—Inner Classes 277

Chapter 16—Applets 291

Chapter 17—Exceptions 311

Introduction

COURSE PURPOSE

This coursebook was developed for self-study training and will assist you during class. Along with comprehensive instructional text and objectives checklists, this coursebook provides easy-to-follow hands-on lab exercises and a glossary of specific terms. It also provides Internet addresses needed to complete some exercises, although due to the constantly changing nature of the Internet, some addresses may no longer be valid.

Many coursebooks also list additional reference works for continued learning outside the classroom. When you return to your home or office, you will find this coursebook to be a valuable resource, one to which you can refer whether you want to review the steps in an exercise or apply what you have learned.

This course teaches you how to write Java applications and applets. You will learn the Java language mechanics found in other programming languages, such as variables, iterations, control statements, methods and arrays. You will also discuss object-oriented theory as it relates to Java. You will create Graphical User Interfaces (GUIs) for both applications and applets, emphasizing components, layouts, and graphics. This course will also prepare you for the Sun Certified Programmer Exam for the Java 2 Platform by providing reviews and examples relative to the exam.

Upon completion of this course, you will be experienced in writing Java applications and applets. You will also be very familiar and comfortable the Java 2 API.

COURSE GOALS

This self-study course will provide you with the information you need to complete the following:

- Describe the Java Runtime Environment (JRE).
- Use Java variables, control statements, methods and arrays.
- Describe object-oriented theory, including abstraction, encapsulation, inheritance and polymorphism.
- Describe method overloading and overriding.
- Use Java static and instance members.

- Create Java constructors.
- Identify the differences between instance and class members.
- Use Java abstract classes and interfaces.
- Use Java Strings and StringBuffers.
- Describe Java packages and accessibility.
- Use the Java Abstract Windowing Toolkit (AWT) and Swing components central to JDK 1.2.
- Use the JDK 1.2.x event delegation model.
- Define applets and the applet life cycle.
- Throw and handle exceptions.
- Create threads.
- Use streams.
- Describe Java and networking.
- Complete Sun certification examination examples.

EXERCISES

The exercises in this manual are designed to give you hands-on practice working in both stand-alone and network environments. It is suggested that you complete the exercises when referenced. However, this may not always be convenient. If you need to skip an exercise, you should plan on completing the exercise later when time and circumstances allow.

You may find that there are some exercises that you are unable to complete due to hardware or software requirements. Do not let this stop you from completing the other exercises in this manual.

NOTICE:

The exercises in this self-study product are designed to be used on a system that is designated for training purposes *only*. Practicing the exercises on a LAN or workstation that is used for other purposes may cause configuration problems, which could require a reinstallation and/or restoration from a tape backup of the original configuration. Please keep this in mind when working through the exercises.

SCENARIO-BASED LEARNING

This self-study manual uses a number of scenario-based learning exercises. In these, you are presented with a situation similar to those you are likely to encounter in day-to-day support and management. You will be provided with the information you need and asked to determine the best solution. A suggested solution is provided at the back of the self-study manual.

These exercises are being used to supplement hands-on practice and to help get you started thinking critically about practical applications. In some cases, they have been used as a replacement for hands-on practice for scenarios where it would be especially difficult to emulate a real-world situation.

It is important that you take the time to work through the scenario-based exercises. These are an important supplement to the training materials and are meant to reinforce the text information in your manual.

MULTIMEDIA OVERVIEW

The Interactive Learning CD-ROM is a robust collection of learning tools designed to enhance your understanding and prepare you for certification. You access these tools from the Start menu: select Wave Interactive Learning and then select the appropriate curriculum.

Videos

A key element of the Interactive Learning CD-ROM included with this course is digital video. Digital video lessons describe key concepts covered in the manual. Often concepts are best understood by drawing a picture. Digital video segments provide a graphical illustration, accompanied by an instructor’s narration. These lessons are ideal both as introductions to key concepts and for reinforcement.

Assessment

As reinforcement and review for certification exams, the *Challenge! Interactive* is significantly helpful. The *Challenge!* contains sample test items for each exam. The sample tests are comprised of multiple-choice, screen simulation, and scenario questions to better prepare you for exams. It is a good idea to take the *Challenge!* test on a particular exam, read the study guide and then take the *Challenge!* test again. It is useful to take the *Challenge!* tests as frequently as possible because they are such excellent reinforcement tools.

HARDWARE AND SOFTWARE REQUIREMENTS

Hardware Requirements

CIW hardware specifications	Greater than or equal to the following
Processor	Intel Pentium II (or equivalent) personal computer with processor speed greater than or equal to 300 MHz
L2 cache	256 KB
Hard disk	2-GB hard drive
RAM	At least 128 MB
CD-ROM	32X
Video adapter	At least 4 MB
Monitor	15-inch monitor
Internet Connectivity	28.8 modem and ISP account
Sound	16-bit sound card or better with speakers
Network Interface Card	10BaseT and 100BaseT

Software Requirements

- Microsoft Windows 95, 98, ME, 2000, NT 4.0 or higher.
- Netscape Navigator 4.5 or higher.
- Microsoft Internet Explorer 5.5 or higher
- Java Development Kit version 1.2.2 or higher (http://java.sun.com)

CHAPTER 1

Java Runtime Environment

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Identify the differences between stand-alone applications and applets.
- Describe the role of the Java Virtual Machine.
- Create a main() method.
- Describe the differences between *.java and *.class files.
- Use statement terminators.
- Use Java comments.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is the extension of a Java file?

2. What is the extension of a compiled Java file?

INTRODUCTION

One of the major difficulties associated with programming over the years has been the problem of creating cross-platform programs. How can one create software, which, when compiled, will run exactly the same way from one type of computer to the next? This is the problem for which Java was created. Java is a programming language which allows a programmer to create and compile a program on one machine that will run on computers with different operating systems. For example, one could create a program on an Apple computer that could then run on Windows, Linux, or Solaris. This is a definite advantage over standard programming languages which require that programs be compiled for each operating system. In this chapter we will walk through a basic installation of the Java Development Kit (JDK) and the creation, compilation and execution of a small Java program.

THE JAVA VIRTUAL MACHINE

The Java Virtual Machine (JVM) separates Java from most other programming languages. One of the primary design goals of the Java language is to enable the same code to run on any platform. The JVM is a software program that behaves like an entire computer. By using this virtual computer on different platforms (UNIX, Win32, Macintosh, and so forth), you can reuse programs without creating a version for each platform. You will always run Java programs in a JVM.

Java allows you to write stand-alone applications (Java programs that need not be run in a browser), servlets (Java applications that run on a server) or applets (which must be run in a Web browser or a program called Appletviewer). When the code is compiled, the compiler creates bytecode that is stored in a .class file. The virtual machine is responsible for interpreting this byte code into a machine-useable format.

Because of the combination of compilation and interpretation, Java is considered to be a hybrid language. Note that many compilers exist that will compile Java code into a native format. If this is done, the compiled program will not be portable; however, the program will execute without the virtual machine.

THE JAVA 2 SOFTWARE DEVELOPMENT KIT

The development platform used in this course is the Java 2 Software Development Kit (SDK), Standard Edition, v 1.2.2 or higher. The Java 2 SDK was previously known as the Java Development Kit (JDK) 1.2, and it is backward compatible with JDK 1.1. It contains several components, including:

- The Java Runtime Environment (JRE), which provides the JVM.
- The development tools and compilation libraries necessary to create Java applications and applets. This grouping includes the Java compiler, debugging programs and tools to run applets without a browser.

INSTALLATION

Below are the installation procedures for Microsoft Windows and Linux operating systems. If you do not have the current software from Sun, you can download it from http://java.sun.com/products. Look for the most recent release of the Java2 Software Development Kit (SDK) Standard Edition. At the time of this writing the latest release was version 1.3.

Windows 95/98/Me

1. Once you have the installer program downloaded to your computer, simply run it by double clicking its icon.
2. You will be prompted to accept some license agreements and to select an install directory (The default for the current release is C:/jdk1.3, if you change this you should note the path to which you change it).
3. You will also be prompted for the various packages to install (Again, it is recommended to install all of them, although if you are short of disk space, the demo and/or the sources packages would probably suffice). Once you have selected the packages you want, the installation program will do its job.
4. Once the installation program is finished, you should put the Java directory into your Path so that it can be run from any directory. To do this, click Start/Run and run sysedit. In AUTOEXEC.BAT you need to edit the PATH statement to include the directory containing the Java executables. These executables are in the \bin directory within the installation directory that you chose in step 2. Here is an example of a PATH statement where Java was installed to the default directory. The text after the last semicolon is what you would need to add.

   ```
   PATH C:\WINDOWS;C:\WINDOWS\COMMAND;C:\JDK1.3\BIN
   ```

   You can then activate your changes by opening a Command Prompt window and typing "autoexec.bat" at the prompt. If you do not do this, the PATH will not be reset until you restart Windows.
5. To check that Java has been set up correctly, open up a Command Prompt window and type `javac -` if everything is set up correctly, it should return usage options.

Windows NT/2000

The installation procedure for NT and 2000 should be done while logged in as administrator. The process itself is identical to that for Windows 95/98/Me except for the PATH setting procedure.

To set the PATH in NT/2000, go to the Control Panel and double click the System Icon. Go to the Advanced tab and select Environment. Look for "Path" in the System Variables list and edit it as was described above. Then click OK, SET, or APPLY. The new Path will take effect with every new Command Prompt window you open.

Linux

At the time or this writing, Sun provided two methods of installation on a Linux machine, one via RPM file and one via shell script. The following procedure utilizes the shell script method as it is generally not dependent on the version of Linux you are running.

1. Once you have the installation file saved to your computer, move it to the directory you want to install java in (it will make its own subdirectory), and make the program executable. To make it executable, open a terminal (if you are in X Window) and type

   ```
   chmod +x j2sdk*.bin
   ```

2. Then run the program by typing the file's name at the prompt, for example:

   ```
   johndoe$ ./j2sdk-1_3_0-linux.bin
   ```

3. You will be prompted to accept the licensing agreement. If you agree, type yes. The installation program will unpack the files and install them in the directory you chose in step 1. Once it is finished, if you type "ls" you should see a new directory named jdk1.# that corresponds to the package you installed.
4. Now you should set the Path environment variable to point to the directory containing the java executables. The method for doing this varies with the type of shell you are working under (bash,csh,sh etc...), so it is left to the reader to decide.
5. Once the PATH is set, you can test that it is correct by going to a command prompt and, from any directory, typing `javac`. If everything is set correctly, it should return usage options.

Creating a Simple, Stand-Alone Application

You will begin writing a simple program by defining a class. Give the class a name and a pair of braces to contain the body of the class. By convention, the class name should start with a capital letter as in the following example:

```
class HelloWorld
{
}
```

When you write a stand-alone application, the Java Virtual Machine must know where to begin executing the code. To begin executing code, the JVM looks for and calls a special method by the name of public static void main(String[] args). The JVM uses this method to begin executing your program.

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

This program is complete. Note that Java programs are case-sensitive; the word *Class* is not the same as *class*. You must save the program with your text editor as HelloWorld.java (giving it the same name as your class), and compile it using a Java compiler. You can use the Javac program if you have the Java 2 Software Development Kit (SDK) supplied by Sun.

```
javac HelloWorld.java
```

The Java compiler converts your source code (saved as HelloWorld.java, a text file) to the HelloWorld.class file. This *.class file is no longer a standard text file, but a file compiled into bytecode.

Bytecode is another element that makes Java different from other programming languages. Other compiled programming languages generate machine dependent binaries, which are files that contain native machine language statements. To make Java portable, the Javac compiler generates bytecode. Java bytecode is composed of an instruction set native only to the JVM. Because JVMs exist for multiple platforms, the Java bytecode is portable to any JVM. This portability makes Java incredibly powerful, while at the same time, it makes Java less robust than languages such as C++.

TECH TIP:

As noted earlier, bytecode is interpreted by the JVM. However, the loss of speed caused by interpretation can be overcome. If the code is compiled into native binaries, the program will execute like any other binary.

To execute the program, you must run it in a JVM. This is accomplished by going to the command prompt and invoking the Java interpreter java:

```
java HelloWorld
```

Note that you type java HelloWorld, not java HelloWorld.class. When you enter the proper command, the JVM will display "Hello World!" in a command line window. The Java application development cycle is shown in Figure 1-1.

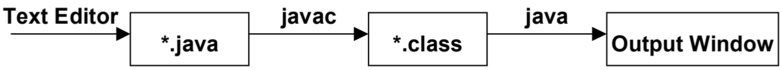


Figure 1-1: Java development cycle

CREATING A SIMPLE APPLET

Applets are Java programs that run within a Java-enabled Web browser. They do not utilize the main() method as stand alone applications do. Instead there is a hierarchy of methods that are run at startup. This will be discusses in more detail later. For now, open up a text editor and enter the following program.

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloApplet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("Hello World", 10, 10);
    }
}
```

Save this file as HelloApplet.java (you MUST give the file the same name as the primary class). Once it is saved, compile it. Open a command prompt, go to the directory you saved the file in and type

```
javac HelloApplet.java
```

This will, if successful, generate a bytecode file named HelloApplet.class.

Now you need to create a HTML file to execute the applet. Again, open a text editor and create the following file:

```
<HTML>
<BODY>
<APPLET CODE=”HelloApplet.class” Width=250, Height=50>
</APPLET>
</BODY>
</HTML>
```

This should be saved in the same directory that contains the HelloApplet.class file, and you may name it anything that has a .htm or .html extension. Save it as hello.html. Once it is saved, open a Java-enabled browser (the latest Netscape or Internet Explorer should work) and point it to the file that you just saved. It should open with the text “Hello World” in the window.

JAVA COMMENTS

When programming, it will be useful to add comments to your code. Java supports three types of comments; two will be recognizable to those familiar with C or C++, and the third type is unique to Java.

The three types of comments are:

- Single-line comment //
- Multiline comment /* … */
- Javadoc comment /** … */

The following code shows examples of all three comment types:

```
class HelloWorld
{
  //**
  This is a javadoc comment. It is a multiline commment unique to
  Java. If you use the javadoc utility (which comes with Java),
  it will automatically create HTML-based documentation for you. The
  online help that comes with the Sun SDK was created using javadoc
  comments.
  */
  public static void main(String[] args)
  {
      System.out.println("Hello World!");
  // Two forward slashes indicate a single-line comment.
  // Notice that each line must begin with the
  // two forward slashes.
  /*
  This is a multiline comment. It is possible to have multiple lines
  of code within this block.
  */
  }
}
```

The single-line and multiline comments will be used in this book.

SUN CERTIFICATION

The goal of this course is to prepare the student for the Sun Certified Java Programmer examination. The exam is administered at various independent testing centers. For more information on the exam, go to http://java.sun.com/100%. The exam consists of 59 questions and lasts 90 minutes. Some of the questions have only one correct answer and others require that multiple correct answers be selected from a list. There may also be some short text answers.

SUMMARY

Java is used primarily to create three types of programs: applications, applets, and servlets. This chapter walked you through the creation of a simple application and a simple applet. A Java program is written and saved as a text source file with a .java extension. The program is converted into bytecode with a .class extension by the Java compiler. The compiled bytecode can be executed using the Java interpreter. The following chapters will cover the basic syntax of the Java programming language.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is the command to compile a java file?

2. What is the command to execute a java file?

CHAPTER 2

Data Types, Variables and Operators

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Use primitive data types.
- Declare variables.
- Distinguish between implicit and explicit casting.
- Determine if a variable is a local, instance, or class variable.
- Use the operators for primitive types.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What are the primative data types in Java and how many are there?

   ....

   ....

2. How would a variable named test be declared as type int and assigned the value 5?

   ....

   ....

3. Can a short variable be assigned to a long variable? Can the reverse happen?

   ....

   ....

4. What is the difference between using "=" and "=="?

   ....

   ....

INTRODUCTION

This chapter focuses on the basic building blocks of the Java language, such as data types, variables, and operators. A programmer cannot learn the advanced concepts of a programming language without having a thorough understanding of the fundamental topics discussed in this chapter.

DATA TYPES

Java is considered a strongly typed language. To use a variable, the programmer must declare the type of information that this variable can store. Java has eight data types that are referred to as primitives (also known as elemental or simple types): `byte`,`short`,`int`, `long`, `float`, `double`, `char`, and `boolean`. Java also has many predefined types that are listed in the Java API, and the language allows the creation of user defined data types. These topics are covered later in the text, and only the primitives are considered here. The Java primitive data types are listed in Table 2-1.

Table 2-1: Primitive data types

Type Name	Size	Range
boolean	8 bits	true or false
byte	8 bits	-128 to 127 (-2^7 to 2^7-1)
char	16 bits	0 to 65.535
short	16 bits	-32,768 to 32,767 (-2^{15} to 2^{15}-1)
int	32 bits	-2,147,483,648 to 2,147,483,647(-2^{31} to 2^{31}-1)
float	32 bits	~-3.4 x 10^{38} to ~3.4 x 10^{38}
long	64 bits	~-9.2 x 10^{15} to ~9.2 x 10^{15}(-2^{63} to 2^{63}-1)
double	64 bits	~-1.8 x 10^{308} to ~1.8 x 10^{308}

VARIABLES

Java supports three types of variables: local variables, instance variables, and class variables. This section will discuss the placement and scope of variables within a Java class. The following code shows the differences between variable types.

```
class MyClass
{
     // This is a comment

     double salary = 45678.00;  // instance variable
     static int totalcount = 0; // class variable

     public static void main(String[] args)
     {
          char male = 'm';      // local variable
     }
}
```

Class and instance variables are defined outside of a method, while local variables are defined inside a method. The scope of class and instance variables is global within the class, however, the scope of a local variable is only within the block of code in which it is defined. At this point, the only difference to note between class and instance variables is that class variables must be preceded by the keyword `static`, while instance variables are not.

Default variable values

All variables must be inititalized before they can be used. Some variables are automatically initialized to default values by the compiler. Class variables and instance variables default to the values listed in Table 2-2.

Table 2-2: Default values for class and instance variables

Data Type	Default Value
byte, short, int, long	0
float, double	0.0
char	'\u0000' (the null character)
boolean	false
non-primitive types	null

Local variables do not have default values, and must be initialized by the programmer.

Variable declaration and initialization

To declare a variable, Java requires a type and an identifier an initial value is optional. Identifiers must begin with a letter and can contain letters, numbers, and the underscore character (_). An identifier can be any length but must not contain any embedded spaces or special characters. To declare an `int` that stores the value of 40, use the following statement:

```
int age = 40;
```

In this example, `int` is the type, `age` is the identifier, and `40` is the initial value. By convention variable names start with a lower-case letter. The previous statement could also have been written as follows:

```
int age;
age = 40;
```

The number of bits contained in each Java primitive is platform independent. This feature of the JVM is necessary to maintain consistency across platforms.

`boolean` variables have the value `false` or `true`. The values `0` and `1` are not used as in other languages. For example, the following is a valid Java statement:

```
boolean isRunning = false;
```

However, the following statement will not compile because an `int` cannot be converted to a `boolean`.

```
boolean isRunning = 0;
```

The `char` data type is not associated with the 8-bit ASCII table found in other languages, but the 16-bit Unicode character set. Sixteen bits allows 65,536 possible characters to be displayed, which is considered adequate for representing most characters in the majority of written languages. With this approach, Java intends to be an international programming language. Unicode characters are specified by their hexadecimal value between `'\u0000'` and `'\uFFFF'`. English characters can be specified either by their Unicode values, or within single quotes, such as `'A'` or `'7'`.

NOTE: It is important to point out that the low 7-bits of a Unicode character are identical to an ASCII character.

CASTING

Casting is the process of converting values from one data type to another. Java allows implicit casting to larger data types, but casting to smaller data types must be done explicitly.

The following are valid implicit castings:

```
byte b   = 50;
short s  = b;     // Valid because short is 16 bits
                  // easily storing an 8-bit byte.

float f  = 10.0F;
double d = f;     // A 64-bit double has no problem
                  // storing a 32-bit float.
```

The following would not compile because explicit casting is required:

```
int i   = 100;
short s = i;          // Will not compile!
short s = (short) i; // This explicit cast is needed.
```

The explicit casting will work, but be aware that a `short` has 16 bits and an `int` has 32 bits. The result of this cast is that the upper 16 bits of the `int` will be lost, possibly changing the `int's` value.

Other casting rules also apply. Conversion is not allowed between a `boolean` and another data type. The cast from one of the floating-point types to one of the integer types must be explicit even though both `int` and `float` contain 32 bits and both `double` and `long` contain 64 bits. When one of these casts are made, the fractional part of the `float` will be lost. The integer types can be implicitly cast to one of the floating-point types without loss of data. Any implicit cast can be explicitly stated to improve readability, as follows:

```
byte b = 50;
short s = (short) b;
```

In general, casting rules follow an elegant order, as shown in Figure 2-1.

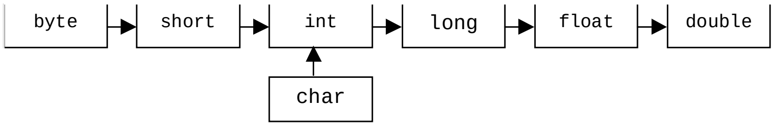


Figure 2-1: Casting rules chart

This chart demonstrates that a `byte` can be assigned to a `short` or `float`, but a `double` cannot be assigned to a `float`, `int`, for `byte` without an explicit cast.

The `char` type holds a 16-bit number representing the Unicode value for a particular character. The following code demonstrates the implications of this:

```
char c = 97;
System.out.println(c); // prints a lowercase 'a'
int i = 'a';
System.out.println(i); // prints the number 97
```

With objects, casting can only be accomplished from a class to its parent class. It is important to note that the Java complier will not allow a cast from a parent class to any of its children.

OPERATORS

Java supports the standard set of arithmetic, bitwise, relational, and logical operators that will be covered throughout this book.

Arithmetic operators

The modulus operator (%) can be used with for both integers and floating points. For example:

```
int a = 24;
double b = 24.2;
System.out.println("a%10 = " + a%10) // would result in 4
System.out.println("b%10 = " + b%10) // would result in 4.2
```

Increment and decrement operators function as they do in C and C++. For example:

```
int a = 1;
a++;  // a now equals 2. This is the same as a = a + 1.
a--;  // a now equals 1 again, same as a = a - 1.
```

The assignment operators also work as they do in C and C++. For example:

```
int a = 1;
a+=10;    // a now equals 11. The same as a = a + 10;
a-=5;     // a equals 6. Same as a = a - 5
a*=2;     // a = 12. Same as a = a * 2;
```

Relational operators

As in C and C++, comparison for equality is done with the double equal signs (==) as opposed to a single equal sign (=). The single equal sign is used only for value assignment. For example:

```
int a = 4;  // Assigns the value of 4 to a.
if(a == 4)  // This asks "does a equal 4?"
            // This would evaluate to true.
```

The following lists the complete set of relational operators.

==	Equality
!=	Not Equal
<	Less Than
<=	Less Than or Equal to
>	Greater Than
>=	Greater Than or Equal to

Logical operators

Logical expressions are used in conditional statements such as if and while loops. These expressions must evaluate to a boolean value. In other languages, numeric expressions can be substituted and evaluated; Java does not allow this.

Java supports short-circuit logical operators for the AND (&&) and OR (||) operators. These allow the program to bypass evaluation of part of the expression when the value of the entire expression is not needed. For example:

```
int a = 2;
int b = 2;
if(a == 1 && b == 2)   // b == 2 would not be evaluated
                       // because a == 1 is false.

if(a == 2 && b == 2)   // Both expressions must be tested
                       // to determine if the expression
                       // is true.

if(a == 1 || b == 2)   // Here, both expressions will be
                       // evaluated because a == 1
                       // is false.

if(a == 2 || b == 2)   // This time only a == 2 is
                       // evaluated because it is true and
                       // sufficient to determine if the
                       // expression is true.

if(a == 1 && ++b == 2) // Notice that because
                       // the first condition
                       // fails, b is never
                       // incremented.
```

TECH TIP:

Throughout this course, you will practice the material before it is implemented in a final project. For this reason, make a file called Practice.java, or create a new Practice file for each chapter. For example, the file for this chapter might be called Practice2.java.

PRECEDENCE

The order in which operators are evaluated in an expression is known as precedence. Associativity refers to the order in which the operands of a particular operator are read. The following table lists the Java operators in order from lowest to highest precedence.

Table 2-3: Java operators from lowest to highest precedence

Operator	Example	Associativity
Assignment	= *= /= %= += -=	Right to left
Conditional	?:	Right to left
Logical OR	||	Left or right
Logical AND	&&	Left to right
Boolean (or bitwise) OR	|	Left to right
Boolean (or bitwise) XOR	^	Left to right
Boolean (or bitwise) AND	&	Left to right
Equality	== !=	Left to right
Relational	< <= > >= instanceof	Left to right
Shift (bitwise)	<< >> >>>	Left to right
Multiplicative	* / %	Left to right
Unary	++ -- ! - (type)	Right to left
Reference Operations	. []	Left to right

SUN CERTIFICATION

Java data types

In Java, each data type has a length and range of values defined by the language. These values should be memorized before taking the Sun Certification exam. Java `char` values are represented by Unicode values preceded by the designator `\u`. The range of all ASCII characters (the character set used in the United States) is `'\u0000'` to `'\u00ff'`. The value `'\u0000'` is the null character not the space character `'\u0020'`. Standard English characters can be specified as their literal values within single quotation marks, such as `'A'` or `'7'`.

Default initial values

Java instance and static variables have default initial values when they are declared. Local variables do not have default initial values. The initial values are 0 or 0.0 for numeric types, `'\u0000'` for `chars`, `false` for `booleans`, and `null` for all other types. Instance and static variables will be discussed in greater detail in the next section of the course.

Operators

The bit-wise operators, which can be used to modify variables at the bit level, appear on the test. The operators are right shift (>>), left shift (<<), and right shift not keeping the sign bit (>>>). The point of emphasis here is on the difference between the >> and >>> operators, which is demonstrated in the following program.

```
class BitShifter
{
    public static void main(String[] args)
  {
      int i = 0x80000000;
      int answer1, answer2;
      answer1 = i >> 3;
      answer2 = i >>> 3;

      System.out.println("initially: " + i);
      System.out.println(">> 3: " + answer1);
      System.out.println(">>> 3: " + answer2);
  }
}
```

The output of this program is as follows:

```
C:\Java>java BitShifter
initially: -2147483648
>> 3: -268435456
>>> 3: 268435456
```

SUMMARY

Java's primitive data types and their ranges were discussed in this chapter. The fixed lengths of these data types is a feature that makes Java platform independent. Variables were declared and the differences between local, instance, and class variables were discussed. Also, implicit and explicit casting, and several operators were introduced. This chapter showed that many standard operators used in other languages are supported in Java.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. How many bits are occupied by each data type?

2. What is the short circuit OR operator in Java?

3. What is the difference between a local variable and a class variable?

4. What is the output of

```
int a = 1;
int b = 2;
a += b;
b = a%2;
a--;
System.out.println(“a: “ + a);
System.out.println(“b; “ + b);
```

5. What is the difference between a static variable and an instance variable?

CHAPTER 3

Control Statements

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Explain the block structure of Java.
- Use expressions inside select and iteration statements.
- Use Java select statements (`if/else` and `switch`).
- Use Java iteration statements (`while`, `do/while` and `for`).

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is an if statement used for?

2. Why should the try, catch, finally construct be with exceptions?

INTRODUCTION

Control statements are one of Java's most fundamental elements and are used in almost all programs. A programmer must learn how to use them well. The `if`, `else`, `while`, and `for` statements are used most often, but it is a good idea to have a working knowledge of all the control statements.

CODE BLOCKS

Java uses braces ({,}), sometimes referred to as curly braces, to define the beginning and end of a code block. A block is used to group the contents of a class or a method, and to designate all conditional and iteration statements. Code blocks are also used for exception handling, which will be discussed in a later section. Blocks can be used at any time to set aside a piece of code. Java ignores white spaces between statements, so the placement of the curly braces is a matter of style. Table 3-1 lists several styles.

Brace Style

`class MyClass {`
`    // Many IDEs`
`    // default`
`    // to this`
`    // style.`
`}`

`void myMethod()`
`{`
`    // Considered`
`    // the`
`    // easiest to`
`    // read.`
`}`

`if(age >= 50)`
`  {`
`    // Classic`
`    // style.`
`  }`

`for(int i; i < 10; i++) {`
`    // Mixed`
`    // style,`
`    // not preferable.`
`    }`

EXPRESSIONS

An expression is a series of variables, operators, and method calls that evaluate to one distinct value. All of the following are expressions:

```
int count = 0;
```

This expression assigns the value of zero to the variable `count`.

```
StringTokenizer st = new StringTokenizer( “string” );
```

This expression creates a new object of the type StringTokenizer.

```
count++;
```

This expression increments the value of `count` by one.

The most simple form of an expression is that of an assignment statement. This is when a variable, method, or a static value assigns its value to another variable. An assignment statement could have this form:

```
int count = counter.getCount( );
```

One type of expression is the boolean expression. The boolean expression will be used quite a bit in this chapter. A boolean expression evaluates to a boolean value, either `true` or `false`. A boolean value is used to direct the behavior of a control flow statement. A control flow statement allows a program to do one of two options based on a boolean control value. A value may be generated either by placing `true` or `false` in the control statement through a method with a boolean return type, or by the evaluation of a comparison. The following chart lists the comparison operators and their definitions.

Table 3-2: Comparison operators

Operators	Definition
<	Strictly less than
>	Strictly greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to
&	Logical AND

Table 3-2: Comparison operators

Operators	Definition
&&	Short circuit AND
|	Logical OR
||	Short circuit OR
==	Is equal to

Most of these operators function in a mathematical manner. They are used to compare the numeric values of variables. The ! operator is the logical NOT operator. It is used to switch between `true` and `false` and vice versa. For instance, `!true` is the equivalent to `false` and `!false` is equivalent to `true`. The logical operators AND, short-circuit AND, OR, and short-circuit OR are used to make logical decisions.

The following shows a sample boolean expression using the equality operator:

```
number == 100
```

In this case, the expression will evaluate to `true` only if the variable `number` equals 100. If it equals anything else, the expression will evaluate to `false`. However, to "capture" this value, it must be assigned to a variable. Consider the next example:

```
boolean trueOrFalse = ( 1 > 12 ) && ( 1 < 12 );
```

In this case, the variable `trueOrFalse` will be assigned the value `false`. The unique thing about this example is how it is evaluated. The `&&` operator is the short-circuit logical AND operator. Like the logical AND operator, it evaluates to `true` only if both side of the expression are `true`. Thus, if the left hand side is `false`, the entire expression will end up being `false`.The short-circuit part means that if the expression on the left hand side is `false`, the expression short circuits, the right hand side is not tested, and the expression is evaluated to `false`. The short circuit OR operator behaves similarly if the left hand side is true.

CONDITIONAL STATEMENTS

Conditional statements are used to ask questions. The two types of conditional statements are `if` statements and `switch/case` statements.

`if` statement

The `if` statement is used to execute a code block conditionally. It surrounds a code block, and that block will only be entered when the condition in the `if` statement evaluates to `true`. The following example should demonstrate this concept.

```
int gradeBen = 76;
int gradeAndrew = 100;
int gradeCassie = 93;
if( gradeCassie > gradeBen )
{
    System.out.println( “Cassie did better on the test than Ben.”);
}
if( gradeAndrew > gradeCassie )
{
    System.out.println( “Andrew did better on the test.”);
}
if( (gradeAndrew == gradeCassie) || (gradeAndrew == gradeBen) )
{
    System.out.println( “Looks like Andrew has been cheating!” );
}
```

The `if` statement will be used throughout the rest of this chapter. A solid understanding of how the `if` statement works allows a programmer to implement complicated behavior easily.

The if statement, which was just introduced, makes a choice between two options. The syntax for the if statement is as follows:

```
if (boolean_expression1)
{
    // One or more lines of code.
}
else if (boolean_expression2)  // OPTIONAL
{
    // Still more lines of code.
}
else                           // OPTIONAL
{
    // Yet more lines of code.
}
```

Following is an example of an if statement (with all the optional flourishes):

```
if ( grade >= 90 )
{
    System.out.println("You made an A!!!");
}
else if ( grade >= 80 )
{
    System.out.println("You made a B!");
}
else if ( grade >= 70 )       // more than one “else if” can be used
{
    System.out.println("You made a C.");
}
else
{
    System.out.println("You should probably study more next time.");
}
```

The construct can contain zero or more else portions. The else if portion is not actually a seperate command, but an else block that contains an if statement. Most programmers place the two statements on one line to make the program more readable. The braces are not required if there is only one line of code for any block; however, these are sometimes included for clarity. Please note that the boolean expression, which is contained in parentheses, must be something that evaluates to true or false. An error will occur if the condition is not a boolean expression.

TECH NOTE:

Remember that a boolean expression is always true or false, not 0 or 1 as in C or Perl. Thus, a statement such as if (5-2) will fail and throw a compile time error.

switch/case statement

The switch statement allows a choice from a number of options. It will test a number of cases against a variable of type char, byte, short, or int. The following is the syntax for the switch statement:

```
switch (variable)
{
    case value0:
        // block to execute
        break;
    case value1:
        // block to execute
        break;
    default:
        // block to execute
}
```

WARNING!

If the break is not present, the next case will also be executed. This allows several case blocks to execute the same code. However, this can lead to problems if the programmer is careless.

The break statement breaks out of the switch statement. Without the break, the program would continue to execute the code block of the next case statement. A break statement should normally be used in each case.

The default option should be the last option in a switch statement. If none of the cases match, the default block is executed. This course will not use the braces in the switch statement, although it is allowable. The following is an example of a switch statement:

```
int choice = 3;
switch(choice)
{
    case 1:
        System.out.println("You chose Menu Option #1");
        break;
    case 2:
        System.out.println("You chose Menu Option #2");
        break;
    case 3:
        System.out.println("You chose Menu Option #3");
        break;
    default:
        System.out.println("You chose an illegal number");
}
```

Multiple case lines without actions or breaks can be used to represent a range of values.

ITERATION (LOOP) STATEMENTS

Iteration or loop statements are used to set up and handle processes that must be repeated. The three types of loops are `while`, `do while`, and `for`.

`while` loop (entry condition loop)

The `while` loop checks a condition before entering the loop, then executes the loop block if the condition is `true`. The condition must be a `boolean` expression contained in parentheses like the `if` statement. The syntax for the `while` loop is as follows:

```
while (boolean_expression)
{
    // loop block
}
```

The following is an example of a `while` loop:

```
int myNumber = 10;
while (myNumber >= 0)
{
    System.out.println(myNumber);
    myNumber--;
}
```

Note that this code will count down from 10 to 0, then terminate. If you want to countdown to 1 then terminate, change the boolean expression in the `while` loop to `myNumber > 0`.

The `while` loop can handle any situation where a loop is needed. However, sometimes a `do while` or a `for` loop can make things easier for the programmer.

`do while` loop (exit condition loop)

Because the condition is checked first in a `while` loop, it is possible that the loop may never execute. This second iteration type ensures that the loop block executes at least once by placing the condition at the end of the loop. The syntax for the `do while` loop is as follows:

```
do
{
    // loop block
} while (boolean_expression);
```

`for` loop

The `while` loop and `do while` loop are useful for many situations, but you must control the `boolean` condition, ensuring that the variables change and eventually make the condition `false`. If you know how many times you want to iterate, the `for` loop is probably the best choice. The syntax is as follows:

```
for (initial_condition; boolean_expression; iteration)
{
    // loop block
}
```

The following steps describe how the `for` loop works.

1. The initial condition statement (the first portion in parentheses) is executed.
2. The `boolean` expression (the second portion in parentheses) is tested to see if it is `true`. If so, the body of the loop is executed, if it is `false`, the `for` loop is exited.
3. The iteration statement (the third portion in the parentheses) is executed. It returns to Step 2 and continues until the loop is exited.

Following is an example loop that counts from 1 to 10:

```
for (int count = 1; count <= 10; count++)
{
    System.out.println(count);
}
```

The iteration variable `count` can be declared outside the loop or declared inside the `for` statement as above. In this example, the variable `count` would be out of scope and unavailable after the loop ends. If the variable is declared before the loop, then it will be available after the termination of the loop.

Nested loops (`break` and `continue`)

The `break` statement was introduced with the `switch` statement in order to keep multiple cases from executing. You can prematurely end other control blocks such as a loop by executing a `break` statement.   Because you do not want the loop to end each time, the `break` statement is placed inside a conditional statement (`if`). For example:

```
for (int countDown = 10; countDown >= 0; countDown--)
{
    System.out.println(countDown);
    if (countDown == 3)
    {
        System.out.println("ABORT");
        break;
    }
}
```

This example will count down from 10 to 3, then print ABORT and end the program. To skip one iteration of the loop, use the continue statement.

```
System.out.println("Leap years between 1896 and 1924");
for (int year = 1896; year <= 1924; year += 4)
{
    if ((year % 100) == 0)
    {
        continue;
    }
    System.out.println(year);
}
```

Because continue and break only work on the current loop, **labels** can be used to further control the loop. Labels can be placed at the beginning of a loop, and you can continue or break to the label. An example of how to use a label follows:

```
boolean found = false;
outer: for (int row = 0; row < 10; row++)
{
    inner: for (int col = 0; col < 10; col++)
    {
        // Some code which may change found to true
        if (found)
        {
            break outer;  // Jumps to outer loop
        }
    }
}
```

TECH NOTE:

Label identifiers follow the same rules as for class or variable identifiers. Many languages that use labels either capitalize their first initials or use all uppercase letters. Most Java reference books use the lowercase style shown here. Whichever style you choose, use it consistently.

Exercise 3-1: Using while and for loops

In this exercise, you will use two different iteration statements to perform the same operation, demonstrating that programming is often a matter of style. Continue with your `SectionOne` class in your `main()` method.

1. Write a program that generates the script for a NASA countdown using a `for` loop. The output should be:

```
10
9
8
7
6
5
4
3
2
1
LIFTOFF!
```

2. Repeat Step 1 using a `while` loop.
3. *(Optional)* Create an unusual countdown program using the modulus operator such that the output reads:

```
10
8
6
4
2
0
LIFTOFF!
```

4. *(Optional)* Modify Step 3 so that the output reads:

   ```
   9
   7
   5
   3
   1
   LIFTOFF!
   ```

5. *(Optional)* Create two integer variables, `firstYear` and `lastYear`. For any arbitrary `firstYear` and `lastYear`, print out all the leap years that exist between the first and last years. Remember that leap years are those years divisible by 4.

TECH TIP:

Remember that how you choose to use curly braces is a matter of style. What is important is that you use them consistently so that others are better able to read your code.

SUN CERTIFICATION

The basic Java control structures (`if`, `switch`, `for` and `while`) are tested on the Sun examination. Specifically, watch for a `switch` statement that does not contain `break` statements for each condition. If the `break` statement is not encountered, the next condition's block of code is also executed until the `break` is encountered or the `switch` statement ends. The exam will also tests concerning the `break` and `continue` statements in nested loops.

SUMMARY

This chapter covered Java's block structure and how to create code blocks with curly braces. You also learned to use conditional statements and iteration statements in your code. As you will see throughout this course, Java contains constructions similar to those in other programming languages.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What data types can be included in a switch statement?
2. In a switch statement, is the break command optional?
3. What is the minimum number of times a do/while loop will iterate?

CHAPTER 4

Methods

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Create and use static methods.
- Return a value from a method.
- Identify the method signature.
- Explain pass by value.
- Describe overloading methods.
- Determine scope of variables.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is an inner class?

2. How can command line arguments be accessed from inside a program?

INTRODUCTION

Methods are one of the most basic concepts in Java programming. To be an effective Java programmer, one must fully understand the functionality of Java methods. Methods are the building blocks of Java classes and programs. In this section, we will explore the basics of methods.

METHODS

Other languages call them functions, procedures, or subroutines, but in Java we refer to the executable parts of a class as methods. Class methods are a focus point in this section. Instance methods will be discussed in the section on object-oriented programming.

A Java application is a free-standing program that can be executed from a command line. The `main` method must be present in all Java applications. This is the starting point for all applications, and flow of control within the program begins here. As with variables, a method becomes a class method by the use of the keyword `static`. The general syntax for a class method is as follows:

```
[accessKeyword] static returnType methodName
         ([parameterType parameterName, pType2 pName2])
{
  // body of method
}
```

The two top-most lines constitute the method declaration, where the top line contains keywords denoting the characteristics of the method. The second line is the parameter list, which states the type, number, and name of variables passed into the method by the calling method. The line // `body of method` represents the method code. This is where the actual functionality of the method would be implemented.

The access keyword can be public, private, or protected. The special form of access called package is the default if no access modifier is declared; however, access keywords should always be used. For now, detailed discussion of the access keyword will be omitted. Within the realm of instructional exercises, where security is not a major concern, public may be used as the access keyword without fear of interfering with the functionality of the code.

The return type can be any Java primitive data type, an object, or `void. Void` denotes that the method has no return value. The return type indicates the type of value returned or reported back to the calling method, and it is most often used in conjunction with the assignment of a value to a variable in the calling method.

The method name follows the same naming rules as variables and will generally begin with a lowercase letter. Because methods and variables are both contained inside classes, they are called class members. As mentioned earlier, the main method must be present in every Java application, since it is this method that the Java virtual machine searches for when executing an application.

The parameter list is a comma-separated list of two-part descriptions. The first entry states the data type of the parameter, which can be either a primitive data type or an object. The second entry declares the name to be used when referencing the parameter within the method. The parameter is a variable inside the method, and the parameter name is the name used for reference.

RETURN STATEMENT

If the return type is not void, then a return statement is required. The value returned must be the same type as the return type listed in the method declaration. The following code is a simple example of a method that determines whether the first int parameter passed into the method is greater than or equal to the second int parameter. The method will return a boolean with the value true if the first int is greater than or equal to the second and false if it is not.

```
static boolean firstIntLarger (int tmpIntOne, int tmpIntTwo)
{
  boolean tmpResult;
  if (tmpIntOne >= tmpIntTwo)
  {
    tmpResult = true;
  }
  else
  {
    tmpResult = false;
  }
  return tmpResult;
}
```

The return statement directs the method to return the current value of its arguments to the calling method. As a result, in the previous example, the current value of the local boolean variable tmpResult is returned to the caller. This method might be used for a decision as in the following code.

```
if(firstIntLarger(x, y))
```

Methods with a return type other than void are most often used to assign a value to a variable in the calling method. An example would be a block of code in the calling method, such as:

```
int passedIntOne = 1;
int passedIntTwo = 2;
boolean returnedValue;
returnedValue = firstIntLarger(passedIntOne, passedIntTwo);
```

In this manner, the boolean variable `returnedValue` is assigned the value returned by the called method. In this case, the value is `false`. The same principle would hold true for other data types. Methods with a return type of void would be called without regard to a return type, execute its code, and return flow of control to its caller when finished, without an explicit return value.

It is important to note that the variable `tmpResult` is a local variable and will only exist within the method `integerLarger`. This effect is referred to as the scope of the variable. It is created within the method, and memory is allocated to hold its value. This memory is returned to the system when the flow of control returns to the calling method. This memory deallocation is referred to as garbage collection, and it will be covered in detail in a later chapter. A variable with local scope is not recognized outside of its method. Its value is returned to the calling method while the local variable itself ceases to exist.

TECH NOTE:

> *In this example, the first part of a parameter name is* tmp*, to denote its temporary nature. The same idea can be used with local variables. Both parameters and local variables go out of scope at the conclusion of the method.*

It is also possible to use a return statement in a method with a void return type. In this case, the return statement is used without an argument. This technique is most often used in conjunction with a decision structure, such as `if` or `switch`. For example:

```
if(termination condition)
  return;
else
  alternative code;
```

CALLING A METHOD

Calling a method located inside the current class can be accomplished by stating the name of the method followed by the parameters to be passed to the method. In the following code, `firstIntLarger` is called by passing the two variables `firstIntegerNumber` and `secondIntegerNumber` to it. In this case, the return value of the method is used as the boolean value for the `if` decision statement.

```
int firstIntegerNumber = 42;
int secondIntegerNumber = 12;
if (firstIntLarger(firstIntegerNumber, secondIntegerNumber))
{
  System.out.println("the first integer is larger");
}
else
{
  System.out.println("the second integer is larger");
}
```

The method can also be invoked with numeric literals, such as:

```
integerLarger(42, 12)
```

Here, the compiler will recognize the two literals as `ints` and pass their value to the parameters. Normally, the two integer variables `firstIntegerNumber` and `secondIntegerNumber` would hold values assigned by a computation elsewhere in the program.

In addition, the method's result can be assigned to a variable, since it returns a data type. In the following code, the boolean variable `larger` is assigned the return value of the method `firstIntLarger`.

```
int firstIntegerNumber = 42;
int secondIntegerNumber = 12;
boolean larger;
larger = firstIntLarger(firstIntegerNumber, secondIntegerNumber);
if (larger)
{
  System.out.println(“The first integer is larger.”);
}
else
{
  System.out.println("The second integer is larger");
}
```

In the event that the called method has a return type of `void` the method is called by simply stating its name. For example, to call a method with return type `void` and an empty parameter list named `noRetValMeth`, the line would read:

```
noRetValMeth();
```

In this manner, no return value is expected, and the called program simply performs its function and then returns control to the calling method. This type of method is usually used to change values of member variables. Because member variables have scope throughout the class, they do not have to be passed as parameters.

Methods in classes other than that of the calling method may be called under certain circumstances using the dot (.) operator. However, these techniques will be discussed later.

Another concept of calling methods is flow of control. When a method is called within a block of code, execution of the code in the calling method is suspended, and the flow of control is transferred to the called method. The flow of control then executes each statement in order within the called method until a terminating condition is met. A terminating condition can be a return statement, the end of the code within the method, or a statement that terminates the program, such as `System.exit()`. In the first two cases, flow of control is returned to the calling method to resume executing statements at the point immediately after the outside method was called. Remember: it is common to call a method from within a called method. In that case, flow of control is transferred to the third method until terminating conditions are met. It is therefore possible to develop complicated control structures within a program.

PARAMETERS

Each parameter has a type and a name, and a method may have zero or more parameters. The parameters are stated in a parenthesized, comma separated list following the stated name of the method. Parameters are then used in a manner similar to variables within the method. Once flow of control is passed to the method, the method's parameters become variables of that method. The type of a parameter can be either a primitive or an object. Although objects as parameters have already been shown in the `main()` method, objects as data types will be discussed in later chapters. The parameter `(String[] args)` is an array of type `String`, a predefined class. The brackets `[]` indicate an array of String objects. Arrays will be discussed in a later chapter. The name of the parameter is `args`. It will be referred to by this name throughout the `main` method.

We will now examine the following method declaration in more detail.

```
    static boolean firstIntLarger (int tmpIntOne, int tmpIntTwo)
```

In this case, there are two parameters passed to the method `firstIntegerLarger`. The type of both is `int`. The names are `tmpIntOne` and `tmpIntTwo` respectively. The parameters are referred to by these names throughout the method.

It also acceptable for a method to expect an empty parameter list. In this case, the parameter list would be a set of empty parentheses, as in the following:

```
public void noRetValMeth();
```

The name(s) used for the parameters by the calling method need not be the same name(s) used by the called method. Also the parameter type is not included in the parameter list used by the calling method; only the names of the passed parameters are included. In fact, the parameter list used by the calling method would be more correctly referred to as the argument list. In all cases, however, the arguments passed to the called method must match in type and number the parameters listed in the method declaration.

PASS BY VALUE

Pass by value and pass by reference refer to the way in which the passed parameters are handled by the called method. In Java, unlike other object oriented languages such as C++, there is no pointer data type, reference operator, or other means to manipulate whether parameters are passed by value of by reference. For this reason, a clear understanding of the parameter rules in Java is essential. Fortunately, the rules are simple; primitive, built-in, data types are passed by value. In other words, the value of the variable is passed but not the actual variable. With pass by value, any changes made to the parameters inside the called method will not change the value of the variables used as arguments.

As an illustration, consider this simple method that increments the integer parameter by one:

```
static void incrementInteger (int tmpInt)
{
  tmpInt++;
}
```

The call to this method is:

```
int myInteger = 42;
incrementInteger(myInteger);
System.out.println(myInteger);
```

The main program sets up memory as follows:

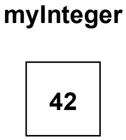


Figure 4-1

When the method is called, a temporary memory variable is set up for the duration of the method. The value of `myInteger` is placed in this temporary location:

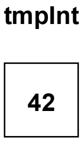


Figure 4-2

When the value of `tmpInt` is changed, it does not affect the value of the original variable `myInteger`. Therefore, what is printed after the call is `42`, not `43`. If the value of `tmpInt` is printed after the increment the 43 would be printed. However, this value is lost when the method finishes execution. This is referred to as pass by value, because the method only has a copy of the original value. In the next chapter, we will discuss pass by reference and what this means for non-primitive data types.

OVERLOADING

There are times that call for performing the same type of calculations on different types of data. In this situation, the same method name is used more than once. This practice is known as overloading and is acceptable in Java. To accomplish this, the methods must have different signatures. The method signature consists of the method name and parameter list. The return type is not part of the method signature. For example, to create several methods that added numbers together, the following methods would all be acceptable even though they are defined in the same class:

```
static int addNumbers (int i1, int i2)
{
  return i1 + i2;
}

static int addNumbers (double d1, int i2)
{
  return (int)d1 + i2;
}

static int addNumbers (int i1, double d2)
{
  return i1 + (int)d2;
}

static int addNumbers (int i1, int i2, int i3)
{
  return i1 + i2 + i3;
}
```

The Java compiler can determine which of the methods to call based on the type and number of parameters. The following methods could not be added to the class shown previously because they only differ by return type from one of the other methods.

```
static double addNumbers (int i1, int i2) { }
static boolean addNumbers (double d1, int i2) { }
static void addNumbers (int i1, int i2, int i3) { }
```

If two methods only differ by return type, the class will not compile. Two methods such as this would be referred to as ambiguous. Overloading of methods will become very important in object-oriented programming, which will be discussed in later sections.

Exercise 4-1: Writing methods

In this exercise, you will practice writing methods. Meaningful implementation of methods will improve the readability and manageability of your code. Many methods will receive parameters and values.

Continue with your `SectionOne` class in your `main()` method.

1. After entering the following code in your `main()` method, implement the three methods to make your code function correctly. Three overloaded class methods must be defined for your code to work.

```
public static void main (String[] args)
{
  // code from the previous exercises

  int i1 = 11;
  int i2 = 5;
  int iTotal = addNumbers(i1, i2);// Method 1
   System.out.println("iTotal: " + iTotal);
  double d1 = 12.0;
  double d2 = 3.9;
  double dTotal = addNumbers(d1, d2);// Method 2
  System.out.println("dTotal: " + dTotal);
  double mixedTotal = addNumbers(i1,d1);  // Method 3
  System.out.println("mixedTotal: " + mixedTotal);

}
```

2. *(Optional)* Create a fourth method called `orderedOutput()` that takes the three totals as parameters (`iTotal`, `dTotal`, `mixedTotal`) and displays them from greatest to least.

TECH NOTE:

It is good practice to choose a method name that is descriptive of the action being performed. The Java convention is to begin method names with a lowercase letter, then identify words by capitalizing the first letter of each subsequent word. For example, a method meant to set the age of a person might be named int setAge(int tmpAge) as opposed to "setage" or "SetAge".

SUN CERTIFICATION

You can expect a question asking you to identify the problem with a piece of code. One of the problems you may see is an improperly overloaded method. Questions exist where the number and type of parameters are the same as another method, but the names of the parameters are different. Questions exist that are ambiguous because of return type. On the test, local variables are used and are often not initialized. Although methods are not specifically tested on the exam, they constitute a basic piece of knowledge that you are expected to understand. Pay close attention to the parameter list, the argument list, and the variables both in the calling method and in the called method. One last thing to watch for is pass by value. Understand when the values of a variable are actually changed and when the values are not altered.

SUMMARY

Methods are one of the defining features of object-oriented programming languages. Overloading of methods allows the same name to perform different functions depending on the data used. Member methods and member variables will form the basic building blocks of objects. Knowing how these methods communicate is essential to understanding the object oriented paradigm and Java.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is the base value of the index of the command line parameter array?

2. How is a method overloaded in Java?

3. Variables declared inside a method have what scope? A) Global B) Local C) static D) Instance?

CHAPTER 5

Arrays

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Declare and initialize an array.
- Allocate space for a new array.
- Describe array indexing.
- Use the length property.
- Discuss Java's garbage collection model.
- Retrieve command line parameters.
- Discuss how arrays are effectively passed by reference.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is an array?

   ....

   ....

2. How do you declare an integer array with ten elements?

   ....

   ....

3. Is it possible to take an element of an integer array and assign the element to a variable of type long?

   ....

   ....

....

....

INTRODUCTION

Many times a programmer will have the need to operate on many similar elements which can be logically grouped together. Perhaps he or she normally has to perform the same operation on all of these elements. In this case he should use an array to hold this data.

Stop now and view the following video presentation on the Interactive Learning CD-ROM (jCert):

Java Programming Fundamentals

Java Array

WHAT IS AN ARRAY?

Arrays are our first introduction to . An array is a linear group of items of the same type. For instance, an array could be a group of integers or it could be a group of floating point numbers, but an array cannot be a mixture of the two. The way that an array is declared illustrates this property.

```
    int[] myarray = new int[10];
```

The square braces here inform the compiler that this is an array of integers and not simply an integer. In this declaration we are telling the compiler to create an array of integers called myarray. As with a class or instance variable, you can give the variable an initial value or declare its value later. The variable that stores the array can be reused to store a new array of integers later.

You may be wondering what an array is if it is not a primitive data type. An array is actually a form of an object. An object is essentially any non primitive data item in Java. Actually, the idea behind objects is a little more complex than that, and will be covered more fully in a later chapter. For now, it is only important to realize that arrays behave differently than primitive data types.

INITIALIZING AN ARRAY

Arrays can either have all of their elements initialized at the time of declaration or the elements can be individually initialized after declaration. The following is an example of an array having all of its elements initialized at the time of declaration.

```
int[] myIntArray = {1, 2, 3, 4, 5};
```

An array in Java is an object. For this example, you can think of myIntArray as acting like a reference (the value is the memory location of the data) to the data structure. A reference is similar to a pointer in C/C++, but you cannot perform pointer arithmetic on a reference. Instead, you can access individual members of the array by using the index (which always starts at 0). To change the value of the fourth element in the array, you would use the following code:

```
myIntArray[3] = 42;
```

TECH NOTE:

> *Although Java has references, pointer arithmetic is strictly forbidden. The only way to access information from an array is to reference the index.*

An array can also be initialized by using the new keyword like

```
int[] myIntArray = new int[5];
```

Note that when you create an array with the new keyword, the value of each element is automatically initialized to the default value of the array's data type. Also, at this time it is incorrect to try to initialize the elements of the array using the curly brackets, such as

```
myIntArray = {1, 2, 3, 4, 5};
```

This type of initialization can only occur at the time of declaration.

Array instantiation in Java is dynamic. Since the size of the array is bound at run time not compile time, an array need not be instantiated when it is declared. This means that you could declare an array with a statement such as,

```
int[] theArray;
```

Then perform some other operations, such as obtaining an integer value from a command line and storing it into a value named numOfItems. The previously declared array could then be instantiated with the new keyword using this value.

```
theArray = new int[numOfItems]
```

If an array is created with the new keyword then the value of the elements must be set individually. For instance

```
myIntArray[3] = 5;
```

would correctly set the fourth element of myIntArray to the value of 5. More commonly, the elements may be using a looping operation similar to

```
for (int i = 0; i < myIntArray.length; i++)
{
    myIntArray[i] = i;
}
```

Here, a for loop is used to initialize the elements of the array. In this looping operation, the termination condition uses the length property of Java arrays. The length property contains the length of an array. An important thing to remember is that Java arrays are zero-indexed, so the index of the last element of the array equals the length minus one. Length is not the only special property associated with an array.

USING AN ARRAY

Suppose you want to copy all of the elements from one array to another. In Java, arrays can be copied like an integer or floating point number. For instance,

```
int[] small = {1, 2, 3};
int[] big = small;
```

assigns the element values of small to the newly created array big, which will also be instantiated to hold three elements. This type of an array copy does not behave exactly like it would with primitives. Instead of creating a new array with separate values that were equal to the values in small, big becomes another reference to the values in small. Thus, both small and big now point to the same data. So, if we change something in one then it is effectively changed in the other.

```
big[2] = 5;
```

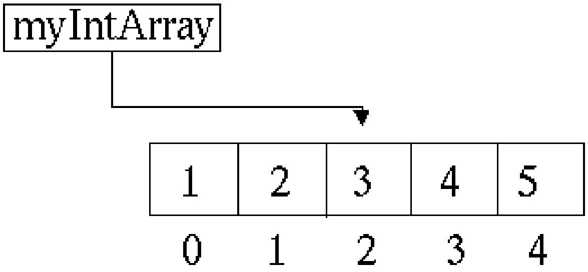
would mean that small[2] equals 5 as well. Java does, however, have a utility for copying elements from one array to another array that does not point to the same memory location. This utility is called arraycopy and it is located in the System class. The call to arraycopy is implemented like

```
System.arraycopy(from_array_name, from_array_element,
  to_array_name, to_array_element, number of elements to copy)
```

where from_array_name and to_array_name are the names of the array's being copied from and to, and were from_array_element and to_array_element are the array element positions from which copying begins. Remember that an array index begins at element 0, and the number_of_elements is the number of elements to copy. This is best explained with an example.

```
int[] odd = {1, 3, 5, 7, 9};
int[] even = {2, 4, 6, 8, 10};
```

This will set up array elements that can be pictured like



Figrure 5-1: Array elements

Now when System.arraycopy is called,

```
System.arraycopy(odd, 1, even, 1, 3);
```

we will change the elements of array even to (2, 3, 5, 7, 10). Notice that elements even[1] through even[3] have been changed to the elements odd[1] through odd[3], and that elements even[0] and even[4] have not been changed at all.

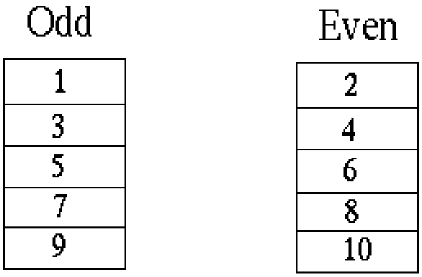


Figure 5-2: Array elements

This also brings up an important point. Copying from one array to another will not insert elements into an array because an array's size cannot change. Instead they simply overwrite any existing element values in the array being copied to.

Other array utilities can be found in java.util.Arrays class and they include a sorting function, a binary search function, a fill function, and an equals function. To use these functions you need to import the java.utils.Array class. The easiest way to do this is to import the entire utils package.

```
    import java.utils.*;
```

The following chart illustrates some of the method declarations used for these functions, as well as, the method declaration for the arraycopy method in the System class.

Table 5-1: Method declaration

java.lang.System	
	static void arraycopy(from_array_name, from_array_element, to_array_name, to_array_element, number of elements to copy)
java.util.Arrays	
	static void sort(array_name)
	static int binarySearch(sorted_array_name, value of same data type as sorted_array_name)
	static void fill(array_name, value of same data type as array_name)
	static boolean equals(array_name, object)

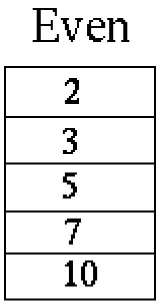
Variations of these methods with different parameters also exist.

Multidimensional Arrays

Java also supports multidimensional arrays. Multidimensional arrays are declared as follows

```
int[][] myarray = new int[i][j]
```

where i and j are the row size and column size. Any number of dimensions may be declared in this fashion. When you declare a multidimensional array in Java, it is actually an array of arrays. So in reality Java does not have multidimensional arrays, but instead fakes this by creating an array of arrays. That is, the elements of the first array are arrays and the elements of the second array are ints. A four by four array is illustrated below.



Even
2
3
5
7
10

Figure 5-3: Four by four array

Each element of the first dimension of the array is actually a 4 element array. These arrays are the second dimension and their elements contain the actual data values. Each element of those arrays may be indexed by referencing theArray[row][column].

PASSING AN ARRAY TO A METHOD

Arrays can be passed as arguments to methods. Arrays may be treated as if they were passed in a call-by-reference manner. Thus, their element values can be changed by the method. The following segment of code shows an array being passed to a method.

```
class test
{
    public static void print(int[] array)
    {
         for(int i= 0; i < array.length; i++)
             System.out.println(array[i]);
    }

    public static void main(String [] args)
    {
        int[] myarray = {1, 2, 3, 4, 5};
        print(myarray);
    }
}
```

In the previous chapter, it was explained that all parameters for methods were passed-by-value, but we just said that you could treat arrays as if they were passed-by-reference. This works because when you pass an array into a method, the method obtains a copy of the array reference. Recall that a reference contains information about where in memory the array is located. So, although the method has a copy of the reference, the method is still able to make changes to the arrays that persist after the method returns. Also, because only the reference is being copied and not the entire array, only minimal additional memory must be used. This feature gives Java the effect of all non-primitive types being passed by reference, even though the mechanism is still pass by value. Thus the same behavior occurs with all objects.

TECH TIP:

You may find it helpful to add the suffix Array or Ary to your array variables. Although not required or even mentioned in many style guides, this practice may help you initially identify what type of information is stored in a variable. Some authors recommend using a pluralized variable name to denote a collection of items.

METHODS WITH AN ARRAY RETURN TYPE

Methods can also be declared with an arrays as their return type. The below code illustrates this idea.

```
class test3
{
    public static int[] reverse(int[] array)
    {
        int[] reverse = new int[array.length];
        int count = array.length;
        for (int i=0; i < array.length; i++)
        {
            reverse[count-1] = array[i];
            count--;
        }
        return reverse;
    }
    public static void main(String[] args)
    {
        int[] myarray = {1, 2, 3, 4, 5};
        for (int i= 0; i < myarray.length; i++)
            System.out.println(myarray[i]);
        int[] myarray2 = reverse(myarray);
        System.out.println();
        for (int i= 0; i < myarray2.length; i++)
            System.out.println(myarray2[i]);
    }
}
```

Notice in the method declaration

```
public static int[] reverse(int[] array) {
```

that the return type is int[]. This method is also accepting an array named *array* as a parameter.

GARBAGE COLLECTION

When you create an array with the new keyword, you are allocating as much space in memory as necessary to store the number of elements in the array. You may wonder what happens when you are no longer using this space. Using the integer array declared previously,

```
int[] myIntArray = new int[10];
```

creates an array with space for ten integers. Now, reuse the variable myIntArray to create a three-element array:

```
myIntArray = new int[3];
```

This results in the following memory:

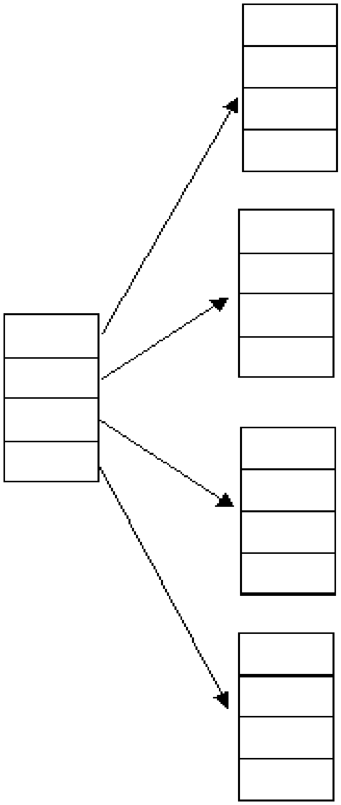


Figure 5-4

C and C++ programmers will probably see this as a memory leak. The old array now has nothing pointing to it. This data cannot be accessed. When this condition occurs, the JVM marks this object for eventual garbage collection. Garbage collection does not need to be explicitly called by the programmer; although, it may not occur immediately.

You can prevent the old array from becoming eligible for garbage collection by creating a reference or pointer to it before you create the three-element array:

```
int[] secondArray = myIntArray;
```

COMMAND LINE PARAMETERS

If you think about it, you have seen the array notation before. The main() method must always have the following method signature:

```
public static void main (String[] args)
```

The array args (or any arbitrary name you choose to give it) will contain a group of Strings. These Strings come from the command line:

```
java Practice String1 String2
```

As with other data types, you can access individual elements of this array with the bracket notation. For example, the first String, String1, would be contained in args[0], the second in args[1], and so forth. You can also test for the existence of command line parameters by testing the length property of the args array. You will learn more about manipulating Strings in the next chapter.

TECH NOTE:

Command line parameters are always String type data. It is necessary to use special String methods in order to convert these to chars, ints, doubles, and so forth. These will be discussed in later chapter. The String array may be defined as String[] args or String args[], this is true of all arrays. The common syntax is String[] args.

Exercise 5-1: Using arrays

In this exercise, you will create a 26-element array to store the characters of the English alphabet. Continue with your SectionOne class in your main() method.

1. Declare an array that can store 26 characters.
2. Write a for loop that populates each element of the array with a character of the alphabet (lowercase).

   HINT: Clever uses of casting will facilitate this.

3. Write a second for loop that prints the contents of the array created in Step 2.
4. *(Optional)* Many ways exist to populate the array with the characters of the alphabet, some more elegant than others. Be sure to experiment with various possibilities. For example, consider using double indexes in your for loop for a clever use of syntax.

GRADUATING TASK #1: CREATING A BINARY SEARCH

We have discussed the language fundamentals of Java from the primitive data types to methods and arrays. In this exercise, you will incorporate all you have learned in this section to write a binary search algorithm.

1. Create a program that will perform a binary search. The binary search will find an element from an ordered list. It uses the same process you might use to find an address in a phone book: look in the middle, then determine whether to look in the first half or last half of the book, and go to the middle of that half.

   Although, this course does not discuss algorithms, a quick look at the usefulness of this procedure is in order. If you are looking through a list of 1,000,000 items using a sequential algorithm (that is, looking at each item in turn until you find the correct one), you will (on average) need to check half the items in the list—500,000 items. With the binary search algorithm on a sorted list, the *maximum* number of items to be checked is equal to the integer greater than or equal to $\log_2(1,000,000)$, which in this example is only 20.

2. To make this algorithm work, use a sorted array of characters to search through and these other four variables. They are not initialized; initialization is your responsibility.

```
char[] searchArray =  // A sorted array of characters.
int high =            // Hint: Use a property of arrays.
int low =
int mid =             // Average the high and low.
                      // Perhaps a method to do this?
boolean foundit =
char searchFor =      // The character you are looking for.
```

3. Write code for the remainder of the binary search. Following is some pseudo code for the algorithm:

```
set low to 0
set high to the length of the searchArray - 1
find the middle with your method
while the solution has not been found
  if the searchArray[middle] is equal to the searchFor char
    print out a message of success
    quit the loop
  if the searchFor char is less than searchArray[middle]
      print out a message that the solution is less
      set the high to the middle
      compute a new middle
                  low remains the same
  else
      print out a message that the solution is greater
      set the low to the middle
      compute a new middle
                  high remains the same
repeat the loop looking for a solution
print out the index of the solution
```

4. *(Optional)* Write code that enables this program to function for other data types (integers, doubles).
5. *(Optional)* Guarantee that the end values work. If not, correct them.
6. *(Optional)* What happens if the character is not found in the array?

SUN CERTIFICATION

Array allocation

Please note that the array initialization with the curly braces can be invoked only when the array is declared, and not at any other point:

```
int[] myAry = {1,2,3};
```

You should also note that the square braces must be empty in the declaration. They can have a number of elements if the new keyword is invoked:

```
int[] myAry = new int[3];
```

Automatic garbage collection

You can expect a question on the exam similar to the following example. You should know when an object no longer has any references to it and is, therefore, eligible for garbage collection. Please note that garbage collection does not necessarily occur at the earliest opportunity. In fact, for the small programs of this chapter, may never happen at all. The garbage collection facility simply knows how to recover memory, if necessary. The garbage collection may be invoked by the programmer, but this is usually not necessary due to automatic garbage collection.

```
a.          class Test
b.          {
c.            static int[] myIntAry = {1, 2, 3, 4, 5};
d.            public static void main(String[] args)
e.            {
f.              myIntAry[3] = 42;
g.              myIntAry = new int[10];
h.              for (int ind = 0; ind < myIntAry.length; ind++)
i.              {
j.                myIntAry[ind] = ind * 2;
k.              }
}
}
```

Command line parameters

One of the questions you can expect to see on the Sun exam involves identification of command line parameters. Be aware that the length property can be used to determine whether any command line parameters have been entered. Unlike in C/C++, the name of the class is not the first element.

SUMMARY

Arrays provide you with an introduction to objects. Arrays are a feature of most programming languages. In the next chapter, you will use the language features of Java to work with object-oriented programs.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. Using the command below, how do you access the word fun and assign it to the String word?

   ```
   java TestProgram This program is fun
   ```

2. Select all the keywords
   - A. True
   - B. false
   - C. new
   - D. bool
   - E. serialized

3. How do you populate an array where the elements are sequential?

CHAPTER 6

Classes and Objects

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Identify the parts of an object.
- Create object references.
- Create and use instance members.
- Identify the differences between instance and class members.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is a class?

2. What is an object?

3. What is a method?

INTRODUCTION

Object-orientation is a software-programming paradigm that helps programmers develop efficient, error-free, reusable components. Object-orientation is based on principles of data abstraction, information hiding, inheritance, and polymorphism. The fundamental building block for all object-oriented designs is the class and its associated object. This section will examine the basics of classes and objects in the Java programming language.

Stop now and view the following video presentation on the Interactive Learning CD-ROM (jCert):

Java Programming Fundamentals

Subclasses

OBJECT-ORIENTED PROGRAMMING

Object-oriented programming refers to the building of blueprint designs called classes, and the creation of these classes into objects that can communicate with each other. Each object contains data elements (variables), and the communication occurs through method calls from one object to another. In this section, we will discuss some of the principles of object-oriented programming and how Java implements those principles. You will learn about inheritance, abstraction, encapsulation, and polymorphism.

WHAT IS A CLASS?

Java code is contained in either a class or an interface. Classes will be discussed here in greater depth.

Before the advent of object-oriented programming languages, programmers used structured programming languages such as C and COBOL. Software developers with experience in these languages think of data, and the code that operates on data, as distinct entities.

In a C program, all the data used to define an employee might be stored in a `struct` that combines multiple simpler data types, such as a character array (to hold the employee's name) and an integer value (to hold his or her annual salary), into a single data type, and employee.

Object-oriented languages such as Java take this idea one step further by encapsulating both the data and code. In this way, a class stores both the data that defines an employee, such as an integer value to hold his annual salary, and methods that operate on this data, such as a method that gives the employee a 5-percent raise. In this way, a class may act as a blueprint for building an employee. It sets the rules and the dimensions, but it does not actually provide the concrete building.

WHAT IS AN OBJECT?

If a class is a blueprint, then an object is the instantiation or realization of this blueprint. Imagine that you have the blueprint for a house (your class). You cannot live in a blueprint. Therefore, you need to build something from this blueprint in order to live in your house. In object-oriented terminology, you instantiate the house class to create a house object. In other words, an *instance* of a house object is created by instantiating the house class. When a class is instantiated, memory is allocated to hold the data that will define the object, and these values are initialized.

To instantiate an object with Java, use the `new` keyword. For example, if you have created a class called `House`, you can instantiate it as follows:

```
House myHouse = new House();
```

This code is more complex than it appears. For now, it is important to understand that you must instantiate an object using the `new` keyword to create a functional object from the class. Once you have an instance of the class, you can work with its instance members.

INSTANCE AND CLASS MEMBERS

The variables and methods that compose a class are its members. Members can be divided into two groups: class members and instance members.

Class members will always have the modifier `static` in front of them. For example:

```
static int age = 30;  // a class variable.
```

The same variable defined as an instance variable would be:

```
int age = 30;         // an instance variable.
```

The following sections describe the difference between class members and instance members.

Instance members

Although, you have already worked with class members in Section I, instance members will be discussed first because they are more commonly used. Using class members is the exception in Java; using instance members is more common.

For example, to create a class structure that may represent the structure of a hospital or other medical organization, you will begin with a class called `Employee`. An `employee` has several characteristics that you can represent as variables, such as `name`, `salary`, and `sickDays`. You will also create a method that will print all the information about each `Employee`. This method consists of four `println()` statements that print the values of instance variables. Your `Employee` class can be implemented as follows:

```
class Employee
{
    // Instance Variables
    String name;   // person's name
    double salary; // salary in dollars
    int    sickDays; // allotted sick days

    // Instance Method
    void printAll()
    {
        System.out.println("Name:   " + name);
        System.out.println("Salary: " + salary);
        System.out.println("Sick Days: " + sickDays);
    }
}
```

You will create another class that contains your `main()` method and you will create objects (instances of `Employee` class). Create two variables: `sam` and `rachael`. Each of these will be an object of type `Employee`. After creating the `Employee` type variables, create an `Employee` object with the `new` keyword. Like primitive variables, a value can be assigned either when the variable is declared or later in the code.

```
class Main
{
    public static void main(String[] args)
    {
        Employee sam = new Employee();
        Employee rachael;            // Instantiating a class
        rachael = new Employee();   // may be done on two lines.
    }
}
```

TECH NOTE:

You have come across an essential point in distinguishing instances from classes. In this code, you instantiated the `Employee` *class twice in order to obtain* `rachael` *and* `sam` *objects. Although,* `rachael` *and* `sam` *are both instances of* `Employee`*, each has its own copy in memory. Therefore, the* `sickDays` *instance variable in* `rachael` *is completely independent of the* `sickDays` *instance variable in* `sam`*. We will later show that this is not the case with class members.*

Accessing instance members

You can assign or reference the instance members by using dot notation (i.e., stating the name of the object, followed by a dot, followed by the member name). You will add names and number of sick days for `sam` and `rachael`, then print the results. You will print the instance variables from the `main()` method first, then use the `printAll()` method.

```
class Main
{
    public static void main(String[] args)
    {
        Employee sam = new Employee();
        Employee rachael;
        rachael      = new Employee();
        sam.name     = "Sam";
        sam.sickDays = 8;
        rachael.name = "Rachael";
        rachael.salary   = 32000;
        rachael.sickDays = 5;
        // Print out sam's instance variables first.
        System.out.println(sam.name + " has " +
        sam.sickDays + " sick days accrued.");

        // Now print out rachael's information using printAll()
        rachael.printAll();
    }
}
```

Notice the values for the uninitialized instance variables in `rachael`. All instance variables initialize to `0` or `0.0` for numeric types, `false` for `boolean` types, `'\u0000'` for `char` types, and `null` for all non-primitive types. Each object (`rachael` and `sam`) has its own set of values for the instance variables. The object reference determines which instance variables are referenced.

When class members are discussed, you will see many similarities, but the differences are very important. The greatest differences between class and instance members will be apparent when you run an example.

Class members

Class members can function quite differently from instance members. First, some syntactical concerns must be noted.

- Class members (variables and methods) are prefaced with the keyword `static`.
- Class methods can access only class members, but instance methods can access either class or instance variables. Therefore, if you are in a `static` method, you can only have access to other `static` methods or variables.
- If you want to access a class member from another class, you may use either the class name or the object name.

TECH NOTE:

> *Class members maintain only one copy in memory. Therefore, even though you may have instantiated many instances of a class, each member that is declared as static will be shared across all instances of that class. This is similar to the concept of global variables, but the word "global" is not acceptable in OO concepts.*

Accessing class members

An excellent example for working with class members is the `Math` class in the Java 2 API. The `Math` class is a collection of utility methods such as the `sin()` method, which returns the sine of an angle. Because the `sin()` method does not need to access any instance member variables in the `Math` class to operate, `sin()` is defined as a `static` method, or a class member method. In this way, the `Math` class does not have to instantiated to make use of the `Math.sin()` method:

```
double mySin = Math.sin(3.14);// By accessing the Math class directly
```

TECH NOTE:

> *You do not have to instantiate the `Math` class to use its member methods. You can reference it directly using the name of the class.*

An effective route is to use class variables to count the number of times a class has been instantiated. For example, if you want to add an `employeeNumber` to the instances of `Employee`, create a class variable.

```
class Employee
{
    // class variable
    static int numberOfEmployees = 0;
    // instance variables
    int employeeNumber;
    String name;
    double height;
    int    weight;
    int    age;
    double salary;
    int sickDays;
    void printAll()
    {
        System.out.println("ID:     " + employeeNumber);
        System.out.println("Name:   " + name);
        System.out.println("Height: " + height);
        System.out.println("Weight: " + weight);
        System.out.println("Age:    " + age);
        System.out.println("Salary: " + salary);
        System.out.println("Sick days: " + sickDays);
     }
}
```

```
class Main
{
    public static void main(String[] args)
    {
        Employee di = new Employee();
        di.employeeNumber =++Employee.numberOfEmployees;
        Employee ken = new Employee();
        ken.employeeNumber = ++Employee.numberOfEmployees;

        System.out.println(di.employeeNumber);
        System.out.println(Employee.numberOfEmployees);
    }
}
```

Study the preceding code carefully. The class `Employee` maintains a class-wide copy of `numberOfEmployees`.

ABSTRACTION

Abstraction is an important concept to understand when learning Java. Abstraction is a way of looking at objects in terms of what you want them to do, rather than how they do it.

Abstraction is a process in which a software developer views a class as a black box, focusing on its input and output, rather than on the details of its implementation. For example, suppose you are designing an application that allows users to generate a graph depicting sales growth. You might develop a `graph` class that is responsible for taking in raw data and generating a graphic image to be displayed in a window.

Using abstraction, you can imagine this class as having a number of inputs, including the number of points to plot, the data series, and the size of the resulting graphic image on the screen. You can also expect a certain result: a graphic image of a certain size. However, the implementation details can be ignored during the design process.

Abstraction provides several advantages. During the design process, abstraction allows software developers to focus on the design of complete applications and systems without pausing to consider implementation details. Also, by focusing on the interface to a class, abstraction allows classes to be more easily reused, modified, or replaced. If at some later time you decide to replace your `graph` class with a new class that generates a three-dimensional graph, the process will be simpler because the implementation details were not finalized during the design process.

OBJECT REFERENCES

The preceding chapter introduced you to references. Similar to a pointer in C and other languages, a reference is a variable that holds information about the location in memory of other information. When you work with objects, you are actually working with references.

Suppose you create two `Employee` variables: `rachael` and `tmpEmp`. When you instantiate a new `Employee` object using the `new` keyword, the object reference `rachael` points to the newly created `Employee` object. Since `tmpEmp` and `rachael` are references, if you assign `tmpEmp` to `rachael`, then both references will point to the same object. Thus, any changes in one object will be reflected in the other. The result is shown in Figure 6-1.

```
Employee rachael = new Employee("Rachael", 32000, 5);
Employee tmpEmp = rachael;
```

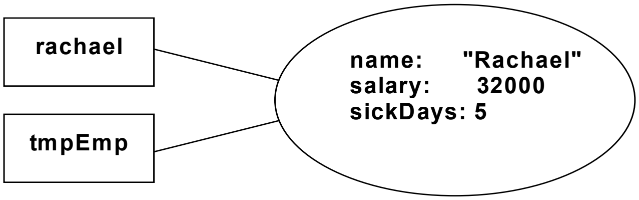


Figure 6-1: tmpEmp variable

If you have experience developing in other programming languages, you are probably familiar with the problem of memory leaks. Memory leaks occur when data that is no longer needed continues to be held in memory, even after all pointers or references to the data are gone. Java provides a service called garbage collection that helps to eliminate memory leaks. Instead of manually freeing memory (as in languages such as C), the Java Virtual Machine automatically frees the memory used by objects to which no reference points. Garbage collection is a low-priority process. The programmer need not manually invoke the garbage collector; although, the garbage collection thread may be notified by the `System.gc()` method.

TECH NOTE:

The programmer should make no assumptions as to when the garbage collector will run. It is guaranteed to run only sometime after the object no longer has a valid reference to it.

Exercise 6-1: Creating your own classes

In this exercise, you will create your own classes, instantiate them, and practice accessing their members using dot notation.

1. Create an `Employee` class with the following (minimal) instance variables. (You may add more.)

```
String name;
double height;
int    weight;
int    age;
double salary;
int sickDays;
```

2. Add a `printAll()` method that will print these instance variables to the command line.
3. Create the starting class and call it `SectionTwo`. Add a `main()` method. Then instantiate two instances of the `Employee` class, set their variables, and print them using the `printAll()` method that you implemented.
4. (*Optional*) Create a two-element array that can store elements of type `Employee`. Populate the array with the two `Employee` instances created in Step 3, then attempt to invoke the `printAll()` method of each instance.
5. (*Optional*) Use a `for` loop to loop through the elements of the array created in Step 4, then invoke the `printAll()` method.

It is often helpful to keep the different types of members together in your class (class methods, class variables, instance variables, and methods). Comments are useful for making these separations.

TECH TIP:

Remember that all members (class and instance, variables, and methods) begin with a lowercase letter. You can distinguish between variables and methods by the presence (or absence) of parentheses. The exception to this guideline is final variables (constants), which are typically all capitalized.

SUN CERTIFICATION

Static members (Math)

Before taking the Sun certification exam, you should memorize several `static` methods from the class `java.lang.Math`. These methods are:

```
ceil()   // Returns the next higher integer.
floor()  // Returns the next lower integer.
random() // Returns a random double between 0 and 1.
abs()    // Returns the absolute value.
min()    // Returns the smallest of two values.
max()    // Returns the largest of two values.
round()  // Finds the closest integer to a floating-point
         // number.
sqrt()   // Returns the square root of a number.
sin()    // Returns the sine of an angle in radians.
cos()    // Returns the cosine in radians.
tan()    // Returns the tangent in radians.
```

We will discuss packages later in this section. You first need to know how to use these methods. The class `Math` has no instance methods or variables. Therefore, you will never create an instance of `Math`. For example, if you wanted to use the following method:

```
public static int max(int x, int y)
```

you would need to reference the class name (`Math`) followed by the dot and the method name. The statement would be:

```
int biggerNumber = Math.max(15, 24);
```

Garbage Collection

Review the discussion of garbage collection. Automatic garbage collection is one of the key features of the Java language, and it is important to be thoroughly familiar with how it operates.

Initial Values

Remember that the initial values for variables are applicable only for instance and class variables. Local variables (those declared inside a method) are not automatically initialized.

SUMMARY

In this chapter, you learned how, in an object-oriented design, a class functions as a blueprint for the concrete instantiation of an object. Next, you learned the difference between instance membersand class members. Instance members are separate for each instance. Class members are kept in common throughout all instances (if any exist). With this introduction to classes and objects, you can begin to appreciate the value of abstract design of object-oriented systems in Java. More complex uses of classes and objects will be covered in a later chapter.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is the difference between an object and a class?

2. How would an instance of class `Employee` be created?

3. How would static method `getPay()` be called for an object `Dave` of class `Employee`, `if getPay()` returns a `double`?

CHAPTER 7

Inheritance

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Create a new class that uses inheritance.
- Create an overridden method.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is encapsulation?

2. Suppose you have a class Employee and a class Nurse. The Employee class has a method called getPay() which defines a function to calculate the salary of the employee. The class Nurse is a subclass of Employee but needs a different function to calculate the salary. How would this be implemented?

INTRODUCTION

Inheritance is an important aspect of object-oriented programming. This chapter will provide the fundamentals needed to implement inheritance in Java programs. An explanation of inheritance and how it is used is provided. This is followed by a discussion on extending classes, polymorphism, and overriding methods. Lastly, there is a section that discusses what is expected on Sun's Java certification exam.

Stop now and view the following video presentation on the Interactive Learning CD-ROM (jCert):

Java Programming Fundamentals

Inheritance

WHAT IS INHERITANCE?

Inheritance is a very useful concept in object-oriented programming that allows the reuse of methods across different classes. Inheritance allows this code to be reused unchanged or to be tailored to meet specific needs. The tailoring of that code is called overriding a method and will be discussed later in the chapter.

Inheritance can be described in familiar terms. One common approach is to start with an `Animal` class, and from the `Animal` class derive a `Mammal` class (a more specialized `Animal`). From `Mammal`, you can derive a `Dog` class, and so forth. This example is conceptually practical in many ways. In this case, a Mammal class would inherit all of the properties of the Animal class, just as a Dog class inherits all of the properties of Mammal.

Recall our brief introduction to objects in Section I. We described how a window on a computer screen can be considered an object with properties and methods. If you want to create your own window, one with buttons and an editing field, where would you begin? You could start from scratch and write your own `Window` class, but why? One already exists. You can use what is already available and modify it to suit your needs. This style of code reuse is the principle of inheritance. In the next section, we demonstrate how one might go about making use of inheritance.

Using inheritance

A classic example of inheritance is the staff of a business. In this case, everyone on staff is an Employee. Every employee shares common traits such as salary, benefits, and age. Most likely, the staff will be made up of multiple job titles each of these having properties, specifically the job. Each of these job titles will inherit the properties of an Employee.

TECH TIP:

Inheritance can speed development time by decreasing the amount of redundency in the code as well as increase the overall performance of a program by localizing the information stored.

A Programmer is an Employee who has a specific profession and salary. You can also define a method called calculateSickDays(), which uses the profession to determine the number of sick days that an employee has available. One way to define a Programmer is to create a class that contains all of the properties of a programmer. This would include all properties, the properties that an average employee would have, and all of the properties the are unique to a programmer. Consider the following class definition.

```
class Programmer
{
   // class variable
  static int numberOfEmployees  = 0;

  int    employeeNum; // key field
  String name;        // the employee's name
  double height;      // height in inches
  int    weight;      // weight in pounds
  int    age;         // age in years
  double salary;      // salary in dollars
  // instance variables specific to Programmer
  String language;    // language he/she programs in

// methods of Employee such as calculateSickDays()
public int CalculateSickDays()
  {
     // body
  }

// ...
}
```

EXTENDING CLASSES

The problem with previous `Programmer` classes is its redundancy. The `Employee` class contains many of the same properties that the `Programmer` class contains. Inheritance allows you to reuse the properties in the `Employee` class that have already been defined. A better approach to creating the `Programmer` class would be to extend the `Employee` class. By doing this, a `Programmer` object will inherit all the methods and variables from `Employee` . In the example below, the `Programmer` class is a subclass of `Employee` which in turn is a superclass to `Programmer`. The new `Programmer` class will only need to describe how it is different from the superclass.

```
public class Employee
{
  static int numberOfEmployees  = 0;
```

```
      int    employeeNum; // key field
      String name;        // the employee's name
      double height;      // height in inches
      int    weight;      // weight in pounds
      int    age;         // age in years
      double salary;      // salary in dollars

    public int CalculateSickDays()
      {
         // body
      }

    }



    class Programmer extends Employee
    {
      // instance variables specific to Programmer
      String language;   // language he/she programs in
      // Specific methods
        public String CurrentCertifications()
        {
          // body
        }

}
```

By using this method, a new type of employee called janitor can now be defined. A janitor's job is completely unrelated to that of a programmer. However, the janitor will include all of the properties of an employee.

```
class Janitor extends Employee
{
  // instance variables specific to Janitor
  String chemicals;
  // Specific methods
  public String AreasToClean()
    {
      // body
    }

}
```

Now, if we were to create an instance of either the `Programmer` class or the `Janitor` class, then both would have the properties of an `Employee`. In the following example, an instance of the `Programmer` class called `bill` is created. The `bill` object is then configured for this employee's specifications.

```
Programmer bill   = new Programmer();
bill.name       = "Bill";
bill.employeeNumber  = ++Employee.numberOfEmployees;
bill.language = "Java";
System.out.println(bill.language);
bill.sickDays   = 5;
```

In the example above, the `Programmer` class extends the `Employee` class. Also, `Employee` implicitly extends the `Object` class. Every class is a subclass of the top most superclass, `Object`.

> *TECH TIP:*
>
> *Java does not support multiple inheritance. Only one class can be listed in the extends clause. Interfaces provide a way of accessing methods in multiple classes. A class that implements an interface has an IS-A relationship with that Interface.*

Using this and super()

A reference to the current object is done by using `this`. The `this` reference is used for self-assignment purposes. In other words, `this` is an internal reference by an object to itself. The `this` reference can be used inside a constructor to call one of the other constructors in the same class. As an example, examine the following code.

```
public Program()
final int START=0;
final int FINISH=0;
{
  this(START, FINISH);   //call the 2-parameter constructor
}
public Program(int x, int y)
{
 //code for constructor
}
```

The zero parameter constructor Program() contains code that calls the two parameter constructor in the same clastor `Program`() contains code that calls the two parameter constructor in the same class. This is a common method for providing a default constructor while at the same time being able to initialize class variables.

It is common to make calls to the super class. Often, the super class contains methods that the sub-class needs. The call to these methods is accomplished by using the `super` keyword. Since the sub-class is an object of the superclass, it will have access to all of the methods in the superclass. Here is an example:

```
public class  Teacher extends Worker
{
    public void action()
    {
       teach();                           //teach like a Teacher
       super.getPayCheck();  //get pay check like a Worker
   }
}
```

The instanceof Operator

Due to the properties of inheritance, it is also a valid operation to cast a subclass up to a superclass. In this case, you could use the `instanceof` operator to verify that a given expression is an instance of some class. The line

```
object instanceof SomeClass;
```

evaluates to `true` if `object` is an instance of `SomeClass`, and is `false` otherwise. The `instanceof` operator can be used with any object and any class. It is important to realize that all classes are subclasses of `Object` and as such, `instanceof` will always be true when used with `Object`.

OVERRIDING METHODS

The printAll() method that was inherited from Employee does not know about the Programmer variable language. This is because the variable does not exist in the Employee class. The printAll() method of Employee will not print the additional instance variables that are located inside the Progammer class. There are two possible solutions for this problem.

1. Completely, rewrite all of the code for the printAll() method inside the Programmer class.
2. Reuse the printAll() method from Employee and add the functionality that is missing.

The best solution is to use the printAll() method inherited from Employee, and modify it just enough to print the additional variables. The super keyword allows access to members of the superclass from the subclass. The following code demonstrates this technique.

```
void printAll()
{
  super.printAll(); // call the parent object's method
  System.out.println("Language: " + language);

}
```

This method has the same method signature as the printAll() method in Employee. A signature is composed of a methods name and parameter list. If the same method signature exists in both the super and sub classes, the method is overriden. If the printAll() method is called on an object of type Programmer, the printAll() method in th Programmer class that will be executed. Overriding a method allows a subclass to customize a method for its particular needs. However, if the printAll() method on an object of type Employee is called, the printAll() method from Employee will be executed.

TECH TIP:

Overridding is one of the key features that makes polymorphism available. The Java Virtual Machine will determine at run time what the actual type of the object is and execute the appropriate method.

Exercise 7-1: Implementing inheritance

In this exercise, you will see the power that inheritance provides programmers in reducing code redundancy as you practice subclassing superclasses.

Implement the following hierarchy of classes:

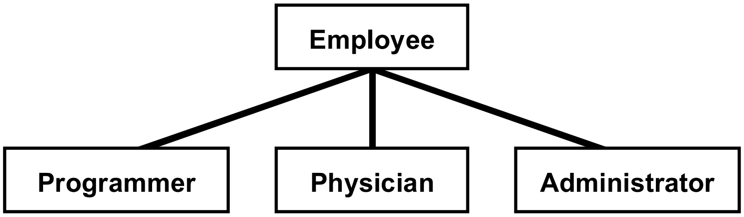


Figure 7-1: Class Hierarchy

1. The foundations of the `Employee` and `Programmer` classes have already been given. Create two additional classes named `Physician` and `Administrator` that meet the following requirements:
   - A Physician is an Employee with a DEA# for writing prescriptions.
   - An Administrator is an Employee with a specific title.
2. (*Optional*) Extend either `Programmer`, `Physician`, or `Administrator` one level further. For example, a `Pediatrician` is a special type of `Physician`.

In all three classes, the `printAll()` method will need to be properly overridden.

SUN CERTIFICATION

One of the concepts tested in the Sun certification exam is your ability to determine the outcome of overridden variables and methods in superclasses and subclasses. You are also expected to know how to store subclass objects in superclass types. The following example demonstrates both skills:

```
class Super
{
  int myInt = 42;

  String test()
  {
    return "Super";
  }
}

class Sub extends Super
{
  int myInt = 11;

  String test()
  {
    return "Sub";
  }
}
```

Working with the following two objects is familiar:

```
Super mySuper = new Super();
Sub   mySub   = new Sub();

System.out.println(mySuper.test()); // prints "Super"
System.out.println(mySuper.myInt);  // prints 42
System.out.println(mySub.test());   // prints "Sub"
System.out.println(mySub.myInt);    // prints 11
```

Apply the skills you have learned to the following object:

```
Super superSub = new Sub();
```

Because `Sub` extends `Super`, this declaration is valid. The `Sub` object created by `new Sub()` must also be a `Super` object. The behavior of the overridden variables and methods is unusual. The variable depends on the type of the object reference `superSub`, whereas the method depends on the type of the object that was created. For example:

```
System.out.println(superSub.test());  // Prints "Sub"
                                      // from new Sub().
System.out.println(superSub.myInt);   // Prints 42
                                      // from Super type.
```

The rules of casting say that you can always put an object into a more general type (that is, you can implicitly cast *up* the hierarchy). For this reason, you can place the object created by `new Sub()` into a `Super` type object reference. Another way to think about this is that a Programmer object is an Employee, so you could assign a Programmer object to an Employee type reference without explicitly casting. If this is reversed, explicit casting is required.

```
Employee e   = new Programmer();      // Legal because
                                      // Employee is
                                      // derived from Programmer.
Programmer p = (Programmer) e;
```

Because the Employee object e actually contains a Programmer object, it can be cast to a Programmer variable. However, the following code would cause a ClassCastException to be thrown.

```
Employee e   = new Employee();
Programmer p = (Programmer) e;        // ClassCastException
```

SUMMARY

This chapter covered how inheritance allows you to reuse code to meet specific needs. You also learned how to override methods and use the overridden method in its own implementation. Inheritance allows you to intelligently reuse classes that have already been written, thereby increasing the speed of development in an object-oriented language. While many design issues must be considered, you can create very sophisticated programs with few lines because much of the work will be done already.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. How would you create a new instance of an Employee?

2. How would you call a super class constructor with a String and int as arguments respectively inside of a subclass?

3. What (JRE)wouldwould(JRE) print if the following code executed?

```
int number; double num;
calculate(number);  calculate (num);
public void calculate (int number) { System.out.print(“LAST”);}
public void calculate( double num) { System.out.println(“FIRST):}
```

CHAPTER 8

Constructors

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Use the default constructor.
- Describe what occurs when a constructor is called.
- Create a constructor to set values.
- Call constructors from the same class (this).
- Call constructors from the parent class (super).
- Create a no-arguments constructor.
- Discuss String characteristics and define the common methods of String.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is a constructor?

   ....

   ....

2. How would you invoke a method that has been overridden?

   ....

   ....

3. If a string is passed to a function, will it be altered?

   ....

   ....

INTRODUCTION

Constructors are a useful tool in object oriented programming. They save programming effort, and if done well, lead to clean, easily maintained classes that can be used in a variety of situations. In this chapter you will be introduced to the fundamentals of constructors and you will be re-acquainted with method overloading, which was introduced to you in chapter 4. You will also learn the keywords `this` and `super` and be introduced to the `String` class which implements constructors extensively.

WHAT IS A CONSTRUCTOR?

A constructor is a special method that is used to instantiate an object. You have already been using constructors with statements, such as the following:

```
Employee sam = new Employee();
Programmer ken = new Programmer();
```

The syntax after the `new` keyword closely resembles a method. This format is a constructor, a method that has the same name as the class in which it exists. This constructor method does not have a return type. While it is generally said to return nothing, it does not have `void` listed as its return type as other methods that return nothing would. However, the combination of the `new` keyword with a constructor returns a reference to an object.

What can constructors do?

Constructors are used to initialize an object that you are instantiating. First, initialize a new object without using constructors. With a class `Programmer` that had member variables `name`, `height` and `weight`, and has corresponding methods. Create a new instance of this class for a new programmer named Rachael, who is 62 inches tall and weighs 120 pounds. Use the following syntax:

```
Programmer rachael;
rachael = new Programmer();
rachael.setName(“Rachael”);
rachael.setHeight(62);
rachael.setWeight(120);
```

This can be tedious when creating many different objects. A better solution is to create a constructor for the `Programmer` class, as in the following:

```
class Programmer
{
    private int height;
    private int weight;
    private String name;
    //The constructor starts here
    Programmer( String n, int h, int w )
    {
        height = setHeight(h);
        weight = setWeight(w);
        name = setName(n);
    }
    //end of the constructor

    ...Rest of Programmer methods go here...
}
```

The constructor has the same name as the class it is constructing, and it does not have a return type associated with it. The constructor is guaranteed to return an object of the type that it is constructing. In other words, the `Programmer` constructor returns an object of type `Programmer`. To use the constructor, simply use the following:

```
Programmer rachael;
rachael = new Programmer("Rachael",62,120);
```

The first line declares a `Programmer` object named `rachael`, and the second line instantiates it and initializes the name, height and weight variables. Often the two previous lines are replaced by the single line:

```
Programmer rachael = new Programmer(“Rachael”,7,48);
```

The four original lines have now been replaced with one simple line of code.

One of the primary goals of object-oriented programming is to establish an effective and architecturally sound means of interobject communication. Constructors accommodate this need with a technique referred to as callback. If an Object B needs to communicate with an Object A, then B can issue a callback to A.

USING CONSTRUCTORS

Once a class is created, instantiate it into an object. An object of type `Programmer` can be created using the following statement:

```
Programmer rachael = new Programmer();
```

The statement `new Programmer`() calls the default constructor, a constructor with no parameters. The constructor returns an object of type `Programmer`. The format for a constructor is as follows:

```
className(parameter_list)
{
    // executable code
}
```

Constructors look and behave like methods, with two differences: constructors have the same name as the class, and they do not have a return type. A constructor is always called with the keyword new. The default constructor for a class has no parameters:

```
Programmer()
{
    name   = “null“;
    height = setHeight(0);    // initailize everything to zero
    weight = setWeight(0);
    age    = setAge(0);
}
```

The above constructor uses the class's methods, such as SetHeight, and SetWeight, (which would have to be added by the programmer) to initialize its variables height and weight. However, overloading, another feature of methods, can be utilized to perform a variety of different actions at initialization. Overloading occurs when two or more methods exist that have the same name and pass different parameters. For example, to set the programmer's name and height at initialization, make an overloaded constructor, as in the following code:

```
Class Programmer
{
    Programmer()              //Default constructor
    {
        name   = “null“;
        height = setHeight(0);    // initailize everything to zero
        weight = setWeight(0);
        age    = setAge(0);
    }
    Programmer(String tmpName, int h)  //An overloaded constructor
    {
        name   = setName(tmpName);
        height = setHeight(h);    // the height will be h
        weight = setWeight(0);
        age    = setAge(0);
    }
  //........rest of Programmer methods would go here....
}
```

The age and the weight are still not initialized by the constructor. These could, of course, be added to the constructor or set later by explicit method calls.

The Java compiler differentiates between calls to either method by examining the parameters passed.

```
Programmer(String tmpName);
Programmer(String tmpName, float tmpHeight, int weight);
```

The constructors' parameters cannot have identical data types. For instance, the following two constructor formats could coexist:

```
Programmer(int h, int w, String name);
Programmer(short age, int w, String name);
```

The following two constructor formats would not be allowed:

```
Programmer(int h, int w, String name)
Programmer(int age, int w, String name)
```

In the Java 2 API, overloaded constructors are used frequently. This concept will become apparent in Section III when you study the AWT.

THIS

The keyword `this` has several uses in Java, even though it officially refers to the current object. The first use of `this` is when referring to a constructor, specifically one in the current class. The second function is to avoid namespace conflicts between a method's parameter list and its variables.

this() as a constructor

When adding different combinations of parameters to the set of constructors, you discover that much of the code is similar. The second constructor can be rewritten to use the first constructor. The keyword `this` allows another constructor to be called that has the same name but a different signature. The following is the rewrite of the second constructor:

```
Programmer(String tmpName)
{
    this(tmpName, 68, 170, 35);
    // ..........rest of constructor follows...
}
```

If the constructor `this`() is used, it must be the first statement in the constructor. The `this`() constructor is also used to execute the same code for all constructors. In a previous chapter, you added a statement after you created an `Employee` object to set the `employeeNumber`. To apply that operation in the constructor, the constructor might be as follows:

```
Employee()
{
    employeeNumber = ++numberOfEmployees;
}
```

To ensure that this operation is always called, add the statement `this()` at the beginning of all other constructors. Now every `Employee` will have an `employeeNumber`, regardless of which constructor is called.

TECH TIP:

One programming style is to define the default constructor first, which is typically the most common constructor. All other constructors point to this default constructor using the this keyword.

Avoiding namespace conflicts

Consider the following constructor:

```
Programmer(String tmpName)
{
    name   = tmpName;
    height = 68;    // The height will be 5'8".
    weight = 170;
    age    = 35;
}
```

The variable name `tmpName` was used in the parameter list, and later assigned it to the instance variable name. Two different names were used so that the local variable `tmpName` would not conflict with the instance variable `name`. While this makes a class very clear, it is not the common practice.

A second approach is to use the keyword `this`. Recall that `this` refers to the current object. Therefore, the following modification is acceptable:

```
Programmer(String name)
{
    this.name   = name;
    this.height = 68.0;    // The height will be 5'8".
    this.weight = 170;
    this.age    = 35;
}
```

By using the keyword `this`, the same variable name can be used without conflict. For clarity in this course, the first method (appending `tmp`) will be used to generally define parameters. Much of the previously written code uses the second method (`this`).

Super

Assume, as in earlier chapters, that `Programmer` extends the `Employee` class. If two constructors are added to the `Employee` class, what happens when a `Programmer` is created? The constructors for `Programmer` will call `Employee`(), and you will get an error message informing you that `Employee`() does not exist. Once a constructor is added, the default constructor is no longer automatically provided by the compiler. This functionality is replaced by defining a no-arguments constructor:

```
Employee ()
{
}
```

Another constructor of the superclass can be called with the keyword `super`. As with the constructor `this()`, when used `super`() must be the first statement of the constructor. For example, to call the `Employee(String)` constructor from the `Programmer(String)` constructor, use the following:

```
Programmer(String tmpName)
{
    super(tmpName);
}
```

TECH TIP:

> *If there is a behavior that all constructors must perform (such as the creation of the employeeNumber), it should be done in the no-arguments constructor. Then all the other constructors in the class can call the this() class.*

Inheritance does not occur with constructors. The fact that an `Employee` constructor takes a `String` does not guarantee that a constructor exists for `Programmer` with a `String`.

TECH TIP:

> *If any constructors are defined for a class, always define a no-arguments constructor, even if it has no behavior. If the class will be extended, a no-arguments constructor must exist so that the extended class will operate properly. In other words, Java will NOT create a default constructor if any constructor exists.*

CONSTRUCTOR PROCESS

When a constructor is called in a class, a call is immediately made to the constructor of its superclass. This process repeats itself up the class hierarchy to the class `Object`. This concept can be readily demonstrated as follows:

```
class Parent
{
    Parent()
    {
        System.out.println("Creating the Parent");
    }
}

class Child extends Parent
{
    Child()
    {
        System.out.println("Creating the Child");
    }
}

class GrandChild extends Child
{
    GrandChild()
    {
        System.out.println("Creating the GrandChild");
    }
}
```

```
class Main
{
    public static void main(String[] args)
    {
        new GrandChild();
    }
}
```

This produces the following output:

```
Creating the Parent
Creating the Child
Creating the GrandChild
```

CONSTRUCTORS AND CALLBACKS

Constructors are a major facilitator in establishing interobject communication. For example, they can be used to create a `Timer` class, whose purpose is to invoke a method in the class that instantiated it (after a certain interval of time).

Students often have difficulty adjusting to the idea of using callbacks as a regular part of their programming skill sets. Remembering the reasons for using callbacks is helpful. Every object in Java should be specialized. Callbacks allow an Object A to assign an appropriate task to Object B. Object B can then notify Object A (asynchronously) when the task is completed. This activity supports separation of responsibility between objects, and efficient use of resources.

You will create a `Client` object that creates and uses a `Timer` object. The `Timer` object will invoke a method of the `Client` object using a callback. Study the following example carefully:

```
class Timer
{
    int interval;
    Client client;
    Timer(Client Client, int tmpInterval)
    {
        this.client = Client;
        // client is a reference to the
        // Client object instantiated by
        // main()
        interval = tmpInterval;
    }

    void run()
    {
        while(true)
        {
            for(int i = 0; i<interval;i++);
                client.timerFired();// This is the callback.
        }
    }
}

public class Client
{
    public static void main(String[] args)
    {
        new Client();
    }
    Client()
    {
    Timer t = new Timer(this, 100000000);
    t.run();
    {
}

public void timerFired(
{
    System.out.println("Timer method fired.");
}
```

This example is used commonly in OO programming. One goal of this course is to make this approach to interobject communication the standard. By the end of this section, you will establish a way to implement the above code in a more reusable fashion with the use of interfaces.

STRINGS AND STRINGBUFFER

You have been using `Strings` throughout these chapters. In Java, `Strings` are not arrays or characters, nor do they need special terminators, as in C and C++. In Java, `Strings` are treated as objects. By using the methods of `String`, programmers are provided with comprehensive `String` functionality from the offset.

String constructors

`Strings` can be created in two ways. First, `Strings` are objects, thus there exists a predictable constructor.

```
String s = new String("Hello");
```

This declares a `String` object initialized to `"Hello"`. However, so does the following:

```
String s = "Hello";
```

`Strings` are used so frequently in this manner that Java interprets the quotation marks as a constructor, which returns an instance of a `String` object initialized to `"Hello"`. In other words, the first and second methods of declaring a `String` are equivalent.

String characteristics

`Strings` possess several characteristics that merit explanation.

Strings are immutable

Strings are immutable, meaning that once a `String` has been instantiated, it cannot change. For example:

```
String s1 = "Hello";
s1 = "Goodbye";
```

It would seem that this code shows one `String` being assigned two values. However, two different `Strings` were created. However, the first `String`, `"Hello"`, is no longer being referenced and is available for garbage collection.

`Strings` are immutable, for increased speed. If `Strings` were dynamic, `String` processes would be slower.

Concatenating Strings

Concatenating `Strings` means adding or combining `Strings` together.

If `Strings` are immutable, how can you explain the following code?

```
String s = "Hello, " + "world!";
```

Remember that "`Hello`," and "`world`" are immutable `Strings` themselves, so there should exist no `String` operation that would add the two `Strings` together. However, Java uses another class to perform the concatenation: the `StringBuffer` class, which will be discussed shortly.

Comparing Strings: equals() and ==

Because Strings are also objects, you need two ways to compare Strings.

The equals() method of String compares two strings, to discern if they contain the same characters.

```
String s1 = "keyboard";
String s2 = "keyboard";
if(s1.equals(s2))
    System.out.println("true");
else
    System.out.println("false");
```

This code would print "true" because the two Strings contain the same characters.

The == operator determines whether the two Strings refer to the same object.

```
String s1 = "keyboard";
String s2 = new String(s1);
if(s1 == s2)
    System.out.println("true");
else
    System.out.println("false");
```

This code would print "false". By creating a completely new String with the new operator, s1 and s2 are completely different objects (albeit with the same characters).

Methods of String

Because Strings are objects, many methods are available. Table 8-1 lists the common methods of Strings.

Table 8-1: Common methods of Strings

Method	Description
length()	Returns an int of the number of characters
toUpperCase()	Returns a new String with all uppercase letters
toLowerCase()	Returns a new String with all lowercase letters
equals()	Returns true if the Strings have the same length and same characters (case-sensitive)
equalsIgnoreCase()	Same as equals(), but not case-sensitive
charAt()	Returns the character at the index

indexOf()
lastIndexOf()
	Returns the index of the first or last occurrence of a character or substring

substring()	Returns a substring from a String
trim()	Returns a new String with leading and trailing white space removed

These methods are not difficult. For example:

```
String s = "Mother";
char c = s.charAt(2);
```

The character c now has the character value t.

However, Strings are immutable. Consider how to change a String from lowercase to uppercase.

```
String s = "hello";
s.toUpperCase();  // This won't work - Strings are
                        // immutable.

s = s.toUpperCase(); // We would have to do this, i.e.
                       // create a new String.
```

The above code could also have been written as follows:

```
String s = "hello".toUpperCase();
```

Remember that a literal String in double quotation marks is considered a String object.

StringBuffer

The StringBuffer provides the missing functionality of the String class, because the StringBuffer class is not immutable. Therefore, additional characters can be inserted into a StringBuffer, or modifying specific characters within a StringBuffer.

Declaring a StringBuffer is just like declaring a String, except you cannot automatically call a constructor with the quotation marks.

```
StringBuffer sb = new StringBuffer("Hello people"); // OK
StringBuffer sb = "Hello people"; // Won't work.
```

Because a StringBuffer is not immutable, you can take advantage of its many methods that change the contents of the StringBuffer.

```
StringBuffer sb = new StringBuffer("Hello people");
sb.insert(6, "nice ");
```

This code would set the variable sb to the following:

```
"Hello nice people"
```

StringBuffers are slower than Strings. It is therefore common to convert a StringBuffer into a String using the toString() method, or by declaring a new String once the StringBuffer manipulations are complete.

```
StringBuffer sb = new StringBuffer("Hello people");
sb.insert(6, "nice ");
String s1 = new String(sb); // Here's one way.
String s2 = sb.toString();  // Here's another way.
```

Exercise 8-1: Building constructors

In this exercise, you will demonstrate several uses of a constructor, such as setting the initial state of an object. The this constructor can also make your less redundant and more readable.

1. Add a constructor to the Programmer class that allows you to specify the following default information: name, height, age, weight, and language.
2. Add another constructor to the Programmer class, so that if name, height, age and weight are supplied but language is not, then a language is automatically supplied. The constructor defined in Step 1 will be called.

HINT: Use the this *constructor.*

3. *(Optional)* Add appropriate constructors for the Physician and Administrator classes.

TECH TIP:

Constructors should be grouped together in the class and identified by a comment statement, just like instance variables, instance methods, class methods, and class variables.

Exercise 8-2: Implementing callbacks

In this exercise, you will develop a `Timer` class that makes a callback to the object which instantiated the `Timer`.

Write a `MyClock` class that will use this `Timer` object to display the date and time of day every second via a callback. Later in the course, you will learn a more effective way of creating this `Timer` system. However, it is important to first master the concept of callbacks.

1. Create a class called `MyClock`.
    - This class contains a `public static void main(String[] args)` method and instantiates a class of type `Timer`. The `Timer` constructor takes an object of type `MyClock`, and an integer (the length of the delay interval).
    - The `MyClock` class also contains a method called `timerFired()`, which is void. Step 4 will explain how to display the current date from within this method.
2. Create a second class called `Timer`.
    - The constructor receives the `MyClock` object and length of the interval as parameters. This class also contains a method called `run()`, which is void. This method contains an infinite loop that repeatedly calls the `timerFired()` method of the `MyClock` object.
    - The `run()` method of the `Timer` object must be invoked to start the repeated callbacks.

3. In the `timerFired()` method, use a `System.out.println()` to display the time of day every second. Consider the following example using the `Date` class:

```
Date date = new Date();
System.out.println(date.toString());
```

TECH NOTE:

The java.util. package must be imported.*

4. (*Optional*) The `Date` class has been largely deprecated in favor of the `Calendar` class. Consult your API documentation, then replace the `Date` class with the `Calendar` class.

SUN CERTIFICATION

Strings

The `String` class is another class in the predefined API that you will must know for the Sun examination.

You are expected to know the following methods for the exam:

```
length()      // Returns an int of the # of characters.

toUpperCase()// Returns a new String with all CAPS.

toLowerCase()// Returns a new String with all lowercase.

equals()      // Returns true if the Strings have the same
              // length and same characters
              // (case sensitive).
```

```
equalsIgnoreCase() // Same as equals, but not
             // case-sensitive.

charAt()     // Returns the character at the index.

indexOf()
lastIndexOf()// Returns the index of the first or last
             // occurrence of a character or substring.

substring()  // Returns a substring from a string.

trim()       // Returns a new String with leading and
             // trailing white space removed.
```

Refer to the information on `Strings` earlier in this chapter.

Constructors

You can expect a few questions asking you to determine which statements run when constructing inherited objects. Following is an example:

```
class Super
{
  int myInt;

  Super()
  {
    System.out.println("Super - no args");
  }
  Super(int tmpInt)
  {
    myInt = tmpInt;
    System.out.println("Super - int");
  }
}

class Sub extends Super
{
  int myInt;

  Sub()
  {
    System.out.println("Sub - no args");
  }

  Sub(int tmpInt)
  {
    myInt = tmpInt;
    System.out.println("Sub - int");
  }
}
```

Consider what will happen when the following statement is executed:

```
Sub s = new Sub(42);
```

It is important to understand that the constructor for `Sub` will call the no-arguments constructor for `Super`, rather than the constructor with one argument. To call that constructor, you must invoke `super(tmpInt)` as the first line in the constructor for `Sub`.

SUMMARY

This chapter discussed the use of constructors, callbacks, and the keyword `this`. You also built constructors to supply default information for an instance of a class, and learned how to use the keyword `super` to call the parent class's constructor. The body of a constructor is an executable piece of a Java program, and will be called each time a class is instantiated to create an object. Although we concentrated on default values and callbacks, you can perform any sort of function in a constructor, although it is called only once for each object.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What would print if this code were executed?

```
static void addstring(String s){s+=” World”;}
public static void main(String (JRE)~~atgs[]~~args[](JRE)){
  String s = “Hello”;
  addstring(s);
  System.out.println(s);
}
```

2. What would print if this program executed?

```
 static String addstring(String s){
   s+=” World”;
   return s;}
public static void main (String args[]){
  String s=”Hello”, a;
  a= addstring(s);
  System.out.println(a);
}
```

3. What would be printed if the following code was executed?

```
class Super
{public Super(){ System.out.print(“Sub”);} }
class Sub (JRE)extends Super(JRE)
{ public Sub(){ (JRE)~~this.~~(JRE)super();
  System.out.print(“Super”);}
public static void main (String sting[]){
 Sub sub = new Sub();
}
```

CHAPTER 9

Interfaces and Abstract Classes

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Define the parts of an interface.
- Use interfaces to implement multiple inheritance.
- Create an abstract method.
- Use abstract classes.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. How does a class implement an interface?

2. What is polymorphism?

3. What is an abstract class?

INTRODUCTION

This section will examine the specifics of interfaces, abstract classes, polymorphism, and how interfaces and abstract classes aid in the realization of polymorphism by analyzing example implementations in terms of object-oriented design.

Stop now and view the following video presentation on the Interactive Learning CD-ROM (jCert):

Java Programming Fundamentals

Java API Interface

WHAT IS AN INTERFACE?

An interface can provide a solution for objects that have multiple characteristics. In your fictional company example, consider the role of a contractor. A contractor is not an employee, but you must pay the contractor a salary. Where do you place the contractor? If you place him or her in the `Employee` category, some attributes and operations may be inherited that you do not want to apply to a contractor. How do you give both types of data similar characteristics, so that you can treat them similarly? Interfaces provide a solution for this problem.

Contents of an interface

In Java an interface is basically an abstract class. The characteristics of an abstract class will be explained in more detail later in this chapter. For now, remember that an interface can contain only abstract methods and final variables. A class may use an interface by having the implements keyword followed by the interface name in the class header declaration. Any class that implements the interface can be consider to have effectively inherited the methods and the variables of that interface. Consider the following example:

```
public interface Payable
{
    final double NOTHING = 0.0;
    abstract double calculateSalary();
}

abstract class Employee implements Payable
{
    // Remainder of class definition
    // must either contain an abstract calculateSalary()
    // or a concrete calculateSalary().
    abstract double calculateSalary();
}
class Contractor implements Payable
{
    // instance variables
    Boolean contractCompleted = false;
```

```
        double calculateSalary()
        {
            double salary;
            if (contractCompleted)
            {
                salary = 15000.00;
            }
            else
            {
                salary = NOTHING;
            }
            return salary;
        }
    }
```

You can add another class called `BillableService` that implements `Payable`. This class would be completely external from the `Employee` hierarchy. Again, as in the preceding `abstract` class example, you can create an array of `Payable` and execute the `calculateSalary()` method on each element. Java can simulate multiple inheritance in this manner.

Interface functions

Consider the earlier discussion on objects talking among/between objects.

Interfaces and coupling

As your object-oriented programming expertise grows, you will learn that designing by interfaces is generally a superior approach to designing by inheritance. Although inheritance is a powerful and useful feature of OO programming, it has one strong drawback, coupling.

Coupling refers to the level of interdependency between objects. In reference to inheritance, a subclass is tightly coupled with its superclass. Should the design of a superclass change, it will most likely affect its subclasses, possibly in unpredictable and detrimental ways.

Coupling can be reduced with interfaces. The only responsibility an object has to its interface is the methods it must implement. This way, the implementation is restricted to the classes that implement the interface, but the classes may still be treated the same because they are linked together by the common interface.

Interfaces and object types

Assigning a type to an object, engages a contract stating minimal behavior of that object. This notion is a very powerful idea, because it extends the basic idea of the "is-a" relationship associated with inheritance, to guarantee that an object can be treated in many fashions. Consider it in relation to the following code:

```
abstract class Employee implements Payable
```

According to this statement, the `Employee` object can be treated as at least three different types:

- All classes in Java are implicitly a subclass of the `Object` class. Thus, the methods of `Object` such as `toString()` and `notify()` are available to the `Employee` class and all other classes.
- It is an object of type `Employee` explicitly.
- It is an object of type `Payable` because it implements the `Payable` interface. Thus, the `calculateSalary()` method is available.

The ability to make an object behave as another type by simply adding an interface is an extremely powerful mechanism for establishing interobject communication. If Object A has implemented a certain interface, and Object B knows this, then Object B knows how to communicate to Object A.

POLYMORPHISM

Polymorphism means many shapes; this is simple to apply by thinking of an object as a combination of different types of objects. That is a single object that has an is-a relationship with various other objects. Polymorphism is visible in two different cases:

1. One method is called invoking different methods on different objects.
2. One object is taking on different types.

Polymorphism occurred when you created an array of type `Employee` that contained objects of different types, and then invoked the `calculateSickDays()` method—a method name common to all these objects. Do not be confused by the fact that they all use the same method name. Every different type of `Employee` contains a different method for calculating sick days. The actual type of `Employee` is determined at run time and the appropriate method is called. Although each object has a `calculateSickDays()` method, each individual object had to implement that method in a meaningful (and probably unique) way.

TECH NOTE:

The example of using polymorphism with the abstract class Employee could also have been performed by using an array of a particular type of interface. For example, if the Employee class implements the Payable interface, then an array of Payable objects can be created to invoke the calculateSalary() method of Payable.

The `Clock/Timer` system you have been developing is a strong candidate for polymorphism via interfaces if you make the following observations:

```
// The following is similar to your previous work:

import java.util.*;
public class DigitalClock
{
    public static void main(String[] args)
    {
        new DigitalClock();
    }

    DigitalClock()
    {
        Timer t = new Timer(this, 10000000);
        t.run();
    }

    public void timerFired()
    {
         System.out.println(new Date().toString());
    }
}
```

```
// You had a Timer class that could perform the callback:

public class Timer
{
    DigitalClock dClock;
    int delay;
    Timer(DigitalClock dClock, int delay)
    {
        this.dClock = dClock;
        this.delay  = delay;
    }
    public void run()
    {
        while(true)
        {
            for (int i = 0; i < interval; i++);
            dClock.timerFired();
        }
    }
}
```

The preceding code worked well. However, suppose you wanted to use your Timer class again. Instead of using another instance of the DigitalClock class, use an altogether different class. For example:

```
public class StockUpdateServer
{
    public static void main(String[] args)
    {
        new StockUpdateServer();
    }

    StockUpdateServer()
    {
        Timer t = new Timer(this, 10000000);
        t.run();
    }

    public void timerFired()
    {
        // code with latest stock updates
    }
}
```

Do you see the dilemma here? Currently, your Timer class is designed to receive objects of type DigitalClock. However, with this StockUpdateServer class, you are asking it to also receive an object of type StockUpdateServer.

What options are available?

Following are two suggestions often given by students. Both are incorrect, but they demonstrate good thinking processes.

Solution 1: Overload the constructor to accept objects of both `DigitalClock` and `StockUpdateServer`.

Response 1: This solution would work, but suppose you had 100 different typed objects that would like to use this `Timer` object. It would not be reasonable (or flexible) to overload a constructor 100 times.

Solution 2: Because `DigitalClock` and `StockUpdateServer` both extend from `Object`, redesign the constructor of the `Timer` class to accept objects of type `Object`.

Response 2: Although, both objects share the `Object` type, the `Timer` class is trying to invoke a `timerFired()` method as a callback. Objects of type `Object` do not have a `timerFired()` method.

These scenarios motivate the need for constructors, because if the `DigitalClock` class and `StockUpdateServer` class have nothing appropriate in common, you can supply an interface as a common element.

The correct solution follows:

```
public interface TimerInterface
{
    public abstract void timerFired();
}

public class DigitalClock implements TimerInterface
{
    public static void main(String[] args)
    {
        new DigitalClock();
    }
    DigitalClock()
    {
        Timer t = new Timer(this, 10000000);
        t.run();
    }
    public void timerFired() //associated with TimerInterface
    {
        System.out.println(new Date().toString());
    }
}
```

```
public class StockUpdateServer implements TimerInterface
{
    public static void main(String[] args)
    {
        new StockUpdateServer();
    }
    StockUpdateServer()
    {
        Timer t = new Timer(this, 10000000);
        t.run();
    }
    public void timerFired() //associated with TimerInterface
    {
         // code with latest stock updates
    }
}

public class Timer
{
    TimerInterface tiObj;
    int delay;
    Timer(TimerInterface tiObj, int delay)
    {
        this.tiObj = tiObj;
        this.delay = delay;
    }
    public void run()
    {
        while(true)
        {
            for (int i = 0; i< delay; i++)
                tiObj.timerFired();
        }
    }
}
```

Run both the DigitalClock and StockUpdateServer simultaneously to see polymorphism in action.

WHAT IS AN ABSTRACT CLASS?

Consider the following example to help explain abstraction and its relation to an abstract class. Consider a small animal set consisting of fluffy the Cat, fido the Dog, spot the Dog, and tweety the Bird. You can collect fido and spot into the Dog class, and you might recognize that both Dogs and Cats are Mammals, but what would it mean to create a Mammal object? Mammal is an abstraction that you can use to describe common properties of Dog and Cat, but you never actually create a Mammal. Therefore, Mammal is abstract.

With Java, you can make either a class or a method abstract. In your example system, define the Employee class to be abstract. In the company example you will create only Administrators, Programmers, and Physicians, but never Employees. Of course, Administrators, Programmers, and Physicians are also Employees, but you cannot create an Employee object. The new Employee class would be:

```
abstract class Employee
{
    // Employee constructors all go here.
    Employee()     // There will be other constructors.
    {
    }

    // instance variables
    int sickDays;
    String name;   // person's name
    double height; // height in inches
    int    weight; // weight in pounds
    int    age;    // age in years
    double salary; // salary in dollars

    // instance methods
    void printAll()
    {
        System.out.println("Sick days:    " + sickDays);
    }

    abstract void calculateSickDays();
}
```

Notice that the calculateSickDays() method was also defined as abstract. To make a method abstract, simply include the keyword abstract and do not declare a body block. If any method is declared abstract in a class, then the class must be declared abstract. You can, however, make the class abstract without having any abstract methods. If the class is abstract, it cannot be used to create an object, but must be extended to a subclass with all methods defined. You can also make the subclass abstract if you do not define all the abstract methods.

Remember that classes also define a data type, so an abstract class can be used for a type. Suppose that you want to create an array of Employees, some of them Administrators, some Programmers, and some Physicians. If you want to have all the Employees execute the calculateSickDays() method, you could do the following:

```
Employee[] emps = new Employee[6];

Physician p1 = new Physician();
// Declare the other Employee subclass objects.

emps[0] = p1;
// Store the other Employee subclass objects
// to the array.

for (int i = 0; i < emps.length; i++)
{
    emps[i].calculateSickDays();
}
```

Even though, you cannot create Employee objects, you can store Employee subclass objects into an array of Employees. You know that the calculateSickDays() method will be defined for all of the elements of the array because they are all extending Employee, guaranteeing that they have this method.

GRADUATING TASK #2: INTERFACES AND POLYMORPHISM

Currently, your `Clock/Timer` class system is referred to as tightly coupled because the `Clock` and `Timer` classes are uniquely dependent on each other to create their callback scenario. In this exercise, you will remedy this dependency with the use of an interface.

1. Create and compile an interface called `TimerInterface` that prototypes a single method `public void timerFired()`.
2. `Decouple` your current `MyClock` and `Timer` classes by implementing the `TimerInterface` with your `MyClock` class.
3. Modify your `Timer` class to accept this new `TimerInterface` type as its callback object.
4. Test your new system.
5. (*Optional*) Create one or two additional classes that also implement the `TimerInterface`. Start up all your systems simultaneously (the `MyClock` system and any other classes you have created here). Observe polymorphism in action.

HINT: The effect is most noticeable if each class that uses a `Timer` object specifies a unique delay and output.

SUN CERTIFICATION

In the Certified Java Developer exam following the Certified Java Programmer exam, you should be able to use abstraction and interfaces in the design of a system. You will be expected to know the tradeoffs in creating and using these concepts. In the Certified Java Programmer exam, you should be familiar with the idea that a class with an abstract method must be declared to be abstract. Also, you should understand that if the subclass does not have a concrete definition for each abstract method, then the subclass must also be declared to be abstract. You should recognize that abstract classes and interfaces can both be used as types.

SUMMARY

This chapter covered how to use abstract classes and abstract methods. You also learned to use interfaces to resolve multiple characteristics and demonstrate polymorphic behavior. The topics in this chapter focused on teaching object-oriented programming.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. There is an abstract class, Student, which has two methods. One, details(), prints the students personal information. The other, debt(), returns the amount of money owed by the student to the school. This abstract class is extended by three other classes, Undergraduate, Graduate and Alumni. Based on this information, what does the following code do and why?

```
Student[] debtors = new Student[10];

Undergraduate ug1, ug2, ug3, ug4;
Graduate g1, g2;
Alumni a1, a2, a3, a4;
double owed = 0;
// assume all variables are initialized.

debtors[0] = ug1;
debtors[1] = ug2;
// the other values are entered into the student array.

for (int i = 0; i < debtors.length, i++)
{
   owed += debtors[i].debt();
}
```

CHAPTER 10

Packages and Access Modifiers

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Compile and run a class with packages.
- Compare the Java 1.1 API and the Java 2 API.
- Describe the object-oriented programming principle of information hiding.
- Identify accessor and mutator methods.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What does a listener class do?

2. What does the Math class provide?

3. What does the Math.round (JRE)~~class~~ method(JRE) do?

4. Which of the following from the collection class has unique elements?
   A. Map
   B. Set
   C. List

INTRODUCTION

You have already seen how classes allow users to group similar methods and data together in a logical fashion. In a similar fashion, similar classes may be logically grouped together ina package. A package allows a programmer to identify related classes. Along with the idea of packages comes the idea of accessibility. Accessibilty determines what class may access members of a given class. Proper use of accessibilty can promote good data encapsulation. Encapsulation allows for writing modular code. Modular code prompts code reuse and increased maintainability, and is thus a desirable goal for programmers.

PACKAGES AND ACCESS MODIFIERS

Recall that a method called `public static void main(String[] args)` was required to run an application. This is a static class method called `main` with a void return value, which takes an array of st`rings as its only parameter`. The keyword `public` requires an understanding of two Java concepts: packages and access modifiers.

Packages

Classes are grouped together because of a relationship which exists between the classes. In Java, these groups of classes are called packages.

A `package` is created by placing the `package` statement followed by the name of the package at the top of a `Java` class definition. The package statement must be the first line of code in the class code. To place the `Employee` class in a `package` called `payroll.info`, the first line of the file would be:

```
package payroll.info;
```

Java places the packages into subdirectories by using the period (.) as a directory separator. According to the above statement, the compiler would place Employee.class in the info directory, a subdirectory of payroll. To compile the revised Employee class and create the directories, enter the following:

```
    javac -d . Employee.java
```

The -d option is followed by the directory path telling Java where to place the compiled class. In this example, the directory path is dot (.), which indicates the current working directory. To execute the Employee class, assuming that the class contains a main() method, enter the following:

```
    java payroll.info.Employee
```

To use the Employee class from another package, reference the class name with the complete notation

```
    payroll.info.Employee e = new payroll.info.Employee();
```

You can also make the Employee class visible by including an import statement before the package designator:

```
    import payroll.info.Employee; // Makes only Employee visible.
    import payroll.info.*;        // Makes all classes in directory
                                  // visible.
```

For this course, you will generally use the default package, which is the directory in which the source files are stored.

The Java recommended method for naming a package is not difficult. Take the domain name of your Internet address, reverse it, and add the name for the package; use periods between the words since they will separate the folders. For example, Sun provides several classes in the com.sun.java package.

Java packages are useful for the following reasons:

- Packages can be used in complex systems to group classes that provide functionality to sections of the program. For example, a program might have a database section, a GUI section, and a networking section. Each of these sections can be assigned to its own package. Packages are often distributed as reusable program modules that can be incorporated into other programs.
- Packages help to avoid naming conflicts with other classes. Including the directory in the name of a class reduces the chance for conflict with other classes. Two classes could have the same name, but reside in different directories.
- Packages allow access control to classes. This principle relates directly to access modifiers.

Access modifiers

You can control the accessibility of a class and its members by using three keywords: `public, protected`, and `private`. These three keywords represent the three access levels supported by Java. A fourth access level, known as `package`, is assumed if no access modifier is specified; however, this level is essentially the same as `protected`. Figure 10-1 shows the level of restriction placed on classes and their members by each keyword. The least restrictive is `public` and the most restrictive is `private`.

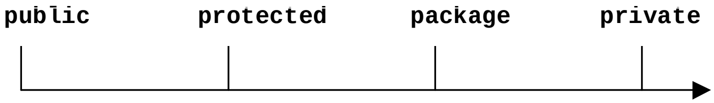


Figure 10-1: Access levels

The access levels perform as follows.

`Public:` States that the method, member, or class is accessible from any method in any other class.

`Protected:` Allows visibility to any methods of the class and subclasses or to any classes in the same package.

`Private`: Restricts access of methods and members to the class in which they are defined.

`Package` This is not an access keyword; however, the absence of an access keyword denotes this access level. This level restricts access to only classes defined in this package.

Any class member can be declared `private`, but a class can be declared private only if it is an inner class. Both a class and a class member can be declared `public`, but a `public` class may have `private` members. Private members are accessable only through accessor and mutator methods.

COMPARISON BETWEEN JAVA 1.1 AND JAVA 2

There have been many changes from the Java 1.1 API to the Java 2 API, most of which involve added and expanded packages. These packages provide large numbers of classes and methods that add greatly to Java's functionality. Some of the upgrades include security enhancements, the Java Frame Class, the Collections API, serialization enhancements, JDBC enhancements, and other performance enhancements. For a more thorough description of the differences, refer to the Web page at Sun MicroSystem's website, <http://java.sun.com/products/jdk/1.2/docs/relnotes/features.html>.

INFORMATION HIDING

Controlling the amount of access another programmer will have to your code is very important as you create reusable Java programs. It is good practice to give other programmers only the required access privileges. By restricting access to a class's members and methods, the creator ensures the code's integrety. Obscuring the inner workings of a class is called information hiding. To implement infomation hiding in your `Employee` class, make all the variables `private`. In the testing program, create accessor and mutator methods that start with the words `get` and `set`.

```
class Employee
{

// Instance Variables

    private String name;   // person's name
    private double height; // height in inches
    private int    weight; // weight in pounds
    private int    age;    // age in years
    private double salary; // salary in dollars
}
// Instance Method

public void printAll()
{
    System.out.println("Name:   " + name);
    System.out.println("Height: " + height);
    System.out.println("Weight: " + weight);
    System.out.println("Age:    " + age);
    System.out.println("Salary: " + salary);
}
```

Direct references to the variables can no longer be used outside of the class. For example, `rachael.name = "Rachael";` would not be allowed by the compiler. You must add methods to get and set these private instance variables. An accessor is a method that allows you to read a private variable**.** A mutator is a method that allows you to modify a private instance variable. The following are examples of accessor and mutator methods:

```
// accessor
public String getName()
{
    return name;
}

// mutator
public void setName(String tmpName)
{
    name = tmpName;
}
```

By using private variables with accessor and mutator methods, you can control the manner in which other programmers use the classes you create.

ENCAPSULATION

Encapsulation is an object oriented concept closely related to information hiding. In an ideal object-oriented world, the objects you create are accessible only by publicly available accessor and mutator methods. The way these methods are implemented is completely hidden from the user of that object. Encapsulation combines all of the data and methods which represent an object into one class and controlles access to that information.

One advantage of encapsulation is reduced coupling. Code that uses an object of a well defined class can continue to use that object even if the class is completely rewritten. For example, suppose a database object changes the database engine it uses. As long as it continues to support the public methods it makes available, nothing else will be adversely affected. Other objects that may have been relying on the methods of that database object may continue to do so. The only thing that has changed is the database object's implementation of those methods, which is encapsulated, and therefore completely hidden. This allows for the implementation of a system to be written for functionality, and then rewritten for optimization.

Another use for encapsulation is security. If an object is tightly encapsulated and the only access to that object is through its well-defined methods, the object is safer from tampering.

Figure 10-2 shows a diagram of object encapsulation.

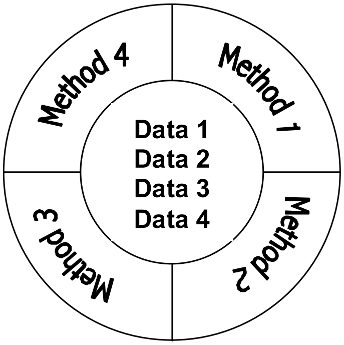


Figure 10-2: Object encapsulation

The data and variables are protected from access by the methods. Use mutators to validate data and to cause other data to be changed. Accessors allow class users to see only the required data. If you study the development of JavaBeans, the convention of naming these methods `getVariable()` and `setVariable()` will be important.

TECH NOTE:

As a general rule, design classes so that all the instance variables are declared private, and the only access to those variables is through public methods. By using accessors and mutators, you can maintain strong encapsulation.

Exercise 10-1: Using encapsulation, accessors and mutators

In this exercise, you will write highly encapsulated objects. Highly encapsulated objects should have their `private` data accessible only through the methods made publicly available.

1. Declare all instance variables of the `Employee` class `private`.
2. Create a proper accessor and mutator for each of the `private` variables.
3. Guarantee that the rest of the class structure works (for example, the `Physician` class). Make any necessary modifications.
4. Test the class structure in your `SectionTwo` class.
5. (*Optional*) Some classes are defined as read-only, meaning immutable. With your understanding of accessors and mutators, create a subclass of `Employee` that is read-only.

SUN CERTIFICATION

You will be expected to know all the keywords used for methods and variables. The keyword static designates a member as a class member. Only one copy will exist, regardless of the number of objects.

The keyword abstract will be discussed in the next section. It denotes that the method only has a specification, and must be overridden in a subclass.

The keyword synchronized will be introduced in the section on `Threads` to designate a method that can be run by only one `Thread` at a time. If the `synchronized` method is a class method, then only one synchronized class method can be run at a time. If it is an instance method, only that method will be restricted. Synchronization will be discussed in more detail later in the course.

In Java, the keyword native designates a method that is implemented in a platform-dependent language. The `native` methods are beyond the scope of this course, and you are not expected to know them for this exam.

The keyword `final` has different meanings for methods and variables. When used with variables, it designates that the value cannot be changed; other languages call this a constant. The value of a `final` variable must be declared when the variable is created because it cannot be changed. As with a method, `the keyword final` means that the method cannot be overridden in a subclass. The keyword `final` for a class means that the class cannot be subclassed.

There are more than forty keywords and reserved words in Java. You should know all of them. While certain words, such as native, are not listed as test objectives, a good Java programmers should at least know what each keyword does.

SUMMARY

This chapter discussed how to store Java classes using packages, and how to use access modifiers to control access to classes and members. The Java 2 API was also introduced. You also learned to protect programs with encapsulation. You will use the package structure of the API, but for the duration of this course, put all the programs into a single directory. When your bytecode is in one directory, there is only `public` and `private` accessibility. Default and `protected` classes and members have the same visibility as `public` in this case.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What does the protected access modifier do? What does the synchronized access modifier do?

2. What does the following code return?

```
Math.round(Math.abs(Math.ceil(-9.4) ) );
```

3. Which collection allows for duplicate items and is ordered?

4. Which object uses a key value pair and does not allow dupilcate keys?

CHAPTER 11

Swing Components

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Distinguish between the capabilities of AWT and Swing.
- Identify the general organization of the Swing class structure.
- Define and use Swing widgets and containers.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is the AWT(JRE) (JRE)and what is it used for?
2. What is Swing and how is it different from the AWT?
3. Which Swing component would you need to use if you wanted to run Java in a Web browser?
4. What is Java Beans technology?

INTRODUCTION

One of the most convenient features of Java is the presence of a set of powerful GUI (Graphical User Interface) components. The original Java language definition included the AWT, which provided for many of the basic interface needs. However, with the release of the Java 1.2 SDK, the Swing package was incorporated. Swing provides a much more powerful and expansive set of user interface components. These Swing components may be customized with minimal effort to look and behave as the programmer desires, providing Java with a powerful set of Foundation Classes for building graphical applications.

WHAT IS THE AWT?

When Sun Microsystems released the Java Development Kit (JDK) 1.0 several years ago, Java came with a platform-independent graphical toolkit called the Abstract Windowing Toolkit (AWT).

The AWT 1.0 package does not provide many prebuilt components, and creation of new AWT 1.0 components is difficult. The only way to customize some prebuilt components is to extend the original component and override methods to provide new functionality, but many components cannot be customized. These restrictions make it difficult to create large-scale GUIs using AWT 1.0, but it is useful for creating simple applets that perform animation or other graphical effects.

The two greatest restrictions of AWT 1.0 are:

- Event handling in one method.
- Use of heavyweight components.

Heavyweight components (peer pattern)

To make AWT 1.0 platform independently, Sun Microsystems chose to use native graphical toolkits (toolkits available to the underlying operating system) to create all the components in AWT. By using only those components available to the native graphical toolkits on every Java-supported OS, Sun had to choose a basic subset of widgets that are available to all supported platforms. This approach is the reason that few prebuilt components come with AWT 1.0.

AWT 1.0 uses heavyweight components. When a programmer creates an AWT button in Windows, the Java Virtual Machine actually creates a Windows button, a peer of the Java button. This process is called the peer pattern. Each time an AWT component is created, its native peer is created and displayed on the screen. To the programmer, it appears that a Java button was created; however, the Java button is a simple interface to access the methods available on the peer button. Similarly, when a button is created on Linux, a native Linux button is created and accessed through the methods and properties available to the AWT button API.

This peer pattern does not allow you to easily create complex user interfaces with advanced components, since these components may not be available on the underlying operating system's graphical toolkit.

AWT 1.1

AWT 1.1 comes with the JDK 1.1.x and higher, but still uses the peer pattern to create components. This toolkit lacks powerful components, since it is using the same types available in the AWT 1.0. The biggest difference between the JDK 1.0 and JDK 1.1 and SDK 1.2.x is not in the AWT but in the way events are handled. Event handling will be discussed in detail later in the course.

WHAT IS SWING?

Swing is a package of lightweight components, called Java Foundation Classes (JFC). The main difference between Swing and the AWT is the absence of the peer pattern. Swing does not use native peers for every component in the API; instead, it uses 100-percent Java GUI components. Swing components look the same across all supported platforms. To use the Swing components, you must import the Swing class like this:

```
    import javax.swing.*;
```

By not using platform-dependent peers or heavyweight components, Swing offers a much richer selection of user-interface components. All Swing components are lightweight, making the Swing API very large. All the classes are approximately 2 MB and require considerable memory. These disadvantages are not serious, however, because JVMs are becoming faster and more efficient. The Java 2 SDK 1.2.x is designed to address many Java speed and stability problems.

All Swing components are written completely in Java, which allows the programmer access to modify them easily; this latitude was not possible with AWT 1.1, which offered no way to radically change the functionality or appearance of standard AWT components. For example, you could not add an image to an AWT button. A separate class had to be created by extending the Panel class and implementing all desired functionality from scratch to create an image button. With Swing, you no longer need to recreate; you can save work by extending existing components.

Swing allows programmers to customize existing components without having to subclass them. This is accomplished by using event delegation and having many methods to change the behavior of each component. Most of the effort involves learning the new patterns and paradigms associated with Swing.

Model View Controller (MVC) programming paradigm

One major difference between AWT 1.1 and Swing is the way in which they are written. Every component in Swing is written using a modified version of the Model View Controller (MVC) programming paradigm. Though difficult to understand at first, this paradigm provides a formal way to create graphical object-oriented programs. In essence, MVC provides three separate parts which are used to model a component:

- The model stores the data defining the component.
- The view displays the component from the data in the model.
- The controller handles user interaction with the component.

Because of the extensive design procedures required to make all three of these parts work together, MVC programs generally have fewer bugs and are more extensible.

BASIC SWING COMPONENTS

Because the Swing API is large, we will restrict our attention to the most commonly used components that do not rely heavily on MVC methodologies and focus on the components shown in Figure 11-1:

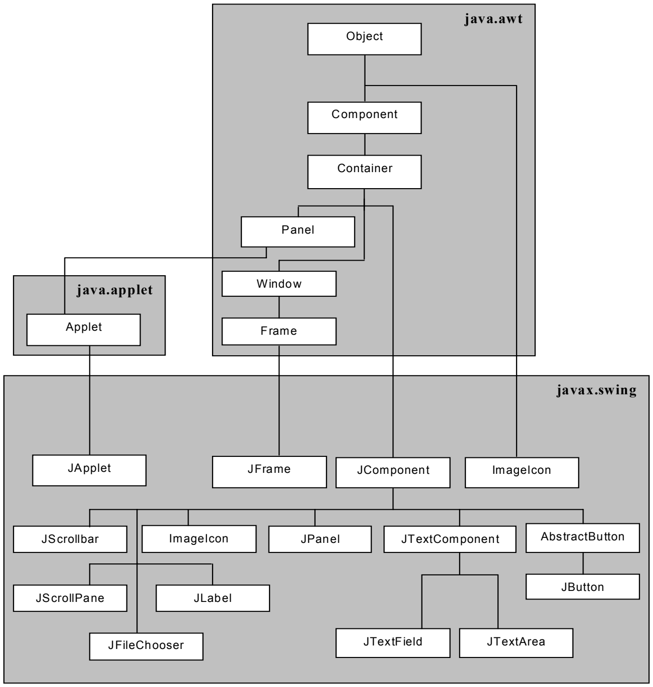


Figure 11-1: Portion of the Swing API

From this figure, we will focus on the following components in this course.

JComponent:	the parent class to many Swing components
JLabel:	displays simple text or images
JButton:	a standard clicking button

JTextField:	for single-line text entry
JTextArea:	for multiline text entry
JScrollBar:	a standard scrollbar
ImageIcon:	represents a graphic or icon. *Note: ImageIcon is not a component*
JScrollPane:	adds scrollable functionality to other components
JPanel:	a non-stand-alone container
JFrame:	a stand-alone container
JFileChooser:	a system-independent file selection dialog
JApplet:	a component to run Java in a Web browser

The preceding figure illustrates the Swing class structure. At the top of the Swing hierarchy is the JComponent class. This class has been declared abstract, but it offers numerous methods for your use. Many of the methods that offer high functionality are inherited from JComponent.

TECH NOTE:

> *Do not confuse JComponent with the term component. The former refers to the class JComponent, whereas the latter is a general term to refer to any of the child classes of the Component class.*

JComponents can be conceptually divided into three categories:

- Graphical widgets.
- Text components.
- Containers.

Graphical widgets

Widgets are the graphical components with which users interact, such as JButtons, JLabels, JScrollBars, and others. Typically, to use a widget of any sort, you need to know the nature of its constructor. As you become accustomed to the various Java components, you can often predict the nature of constructors before referring to the API since they are intuitive.

Interestingly, every Swing component listed in Figure 11-1, except ImageIcon, is a descendent of the Container class. This fact might seem to blur the line between a Container (which can contain other components) and a widget (an interactive graphical element). The ability for a Swing component to contain other components is part of its power. In this chapter, you will explore some of the more common components in detail.

ImageIcon

An ImageIcon is not a JComponent; it extends Object directly. However, ImageIcon is often used with other components, which is why we will examine it before discussing other components. The ImageIcon implements the Icon interface and its functionality. Therefore, anywhere you want to use an Icon, an ImageIcon would most likely suit your needs.

Two of the more useful constructors of ImageIcon are:

```
public ImageIcon(String filename)
public ImageIcon(URL url) // Java has a URL class
```

Because an ImageIcon is not a JComponent, it cannot be directly added to a component that is expecting another JComponent. Therefore, it is necessary to add the ImageIcon to a component that can receive an ImageIcon (or Icon), such as a JButton or JLabel. Consider the following example:

```
class ImageIconStuff extends JFrame
{
   ImageIcon icon    = new ImageIcon("btn_readme.gif");
   JLabel myLabel = new JLabel(icon);

   ImageIconStuff()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      Container c = getContentPane();
      c.add(myLabel);
      setSize(100,100);
      setVisible(true);
   }
}
```

To run this code, you need to create a dummy class whose purpose is to instantiate the necessary classes. The following is an example:

```
public class SectionThree
{
   public static void main(String[] args)
   {
      ImageIconStuff iStuff = new ImageIconStuff();
   }
}
```

The output should resemble Figure 11-2.

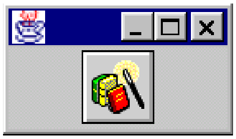


Figure 11-2: Output for ImageIconStuff

From this point on, it will be assumed that the preceding code (or its equivalent) will be used to instantiate any necessary classes.

JButton

To create a JButton, it must be instantiated in the same manner as any class. You will probably want to have access to this JButton later, therefore it should be declared as an instance variable. You need to know the nature of the constructor. It is logical to label the JButton with a functional word such as Calculate or Exit. The JButton class has a constructor that takes a String as a parameter. Additionally, recall that JButton is really a subclass of JComponent, which is a subclass of Container. Therefore, it is possible to place an ImageIcon in a JButton. Several useful constructors of the JButton can be used to accomplish these goals:

```
public JButton(String text) // text only
public JButton(Icon icon) // icon only
public JButton(String text, Icon icon) // text and icon
```

If you choose to display a JButton with both text and an icon, you will likely want to position the text and icon in a particular manner. The JButton class offers methods to do this:

```
JButton myButton = new JButton("Click",icon);
    // icon is previously instantiated

myButton.setHorizontalAlignment(JButton.LEFT);
     // text and icon are flush to the left
     // also RIGHT and CENTER can be used

myButton.setHorizontalTextPosition(JButton.LEFT);
      // text is placed to the left of the icon
      // also RIGHT and CENTER

myButton.setVerticalAlignment(JButton.TOP);
     // text and icon are flush with the top
     // also RIGHT and CENTER

myButton.setVerticalTextPosition(JButton.TOP);
     // text is flush with the TOP of the icon
     // also BOTTOM and CENTER
```

The following example displays a JButton with text and an icon. The default text and icon positions are centered with text to the right.

```
class JButtonStuff extends JFrame // Jframes are covered shortly
{
   ImageIcon icon   = new ImageIcon("btn_readme.gif");
   JButton myButton = new JButton("Click", icon);

   JButtonStuff () // Constructor
   {
      setupGUI();
   }
   private  void setupGUI() // a private internal method
   {
      Container c = getContentPane();
      c.add(myButton); // Take this code for granted for now.
      setSize(100,100);
      show();
   }
}
```

The output from this code should resemble Figure 11-3.

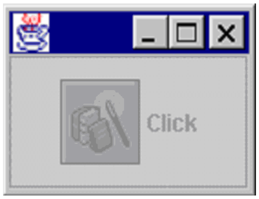


Figure 11-3: Output for JButtonStuff

Many holes exist in the code at this point. How this JFrame is being displayed has not been explained, and the JButton is not active. Nothing should happen when you click it.

Remember that a JButton class is like any other class. It has properties and methods that may be useful. These properties and methods are accessable using the dot notation, as in the following:

```
class JButtonStuff extends JFrame
{
   JButton myButton = new JButton("A Button");

   JButtonStuff()
   {
      setupGUI();
   }
   private void setupGUI()
   {
      myButton.setEnabled(false);
      Container c = getContentPane();
      c.add(myButton);  // Take this code for granted for now
      setSize(100,100);
      setVisible(true);
   }
}
```

The output from the preceding code should resemble Figure 11-4.

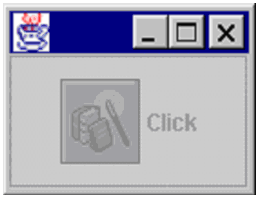


Figure 11-4: New output for JButtonStuff

JLabel

JLabel facilitates the placement of text or images within your GUI. Its constructors follow a similar pattern. One of its constructors takes an ImageIcon. This feature allows you to place an ImageIcon almost anywhere as long as it is first placed in a JLabel. The syntax is as follows:

```
public JLabel(String name) // text label
public JLabel(Icon)        // icon label
```

Like the JButton, a constructor exists for an icon and text to be added upon instantiation as well as a third parameter specifying the horizontal alignment. In addition, methods of the JLabel class allow the following:

```
JLabel myLabel = new JLabel("My Label"); // text label
myLabel.setIcon(icon); // icon previously instantiated
```

Methods can control the alignment of the icon and text of the JLabel independent of instantiation:

```
myLabel.setHorizontalAlignment(JButton.LEFT);
// text and icon are flush to the left
// also RIGHT and CENTER

myLabel.setHorizontalTextPosition(JButton.LEFT);
// text is placed to the left of the icon
// also RIGHT and CENTER

myLabel.setVerticalAlignment(JButton.TOP);
// text and icon are flush with the top
// also BOTTOM and CENTER

myLabel.setVerticalTextPosition(JButton.TOP);
// text is flush with the TOP of the icon
// also BOTTOM and CENTER
```

Consider the following example:

```
class JLabelStuff extends JFrame
{
   ImageIcon myIcon = new ImageIcon("btn_readme.gif");
   JLabel myLabel = new JLabel("A Label");

   JLabelStuff()
   {
      setupGUI();
   }
   private void setupGUI()
   {
      Container c = getContentPane();
      myLabel.setIcon(myIcon); // add the icon
      // place the text to the left of the icon
      myLabel.setHorizontalTextPosition(JLabel.LEFT);
      c.add(myLabel);  // Take this code for granted for now
      setSize(100,100);
      show();
   }
}
```

The output should resemble Figure 11-5.

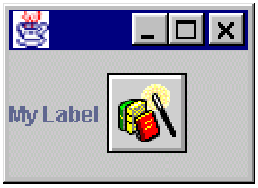


Figure 11-5: Output for LabelStuff

JScrollBar

The JScrollBar class allows you to select from a range of values in a graphical manner. What the constructor should be needs to be answered here also. You will ultimately refer to the API, but trying to determine it on your own is a productive learning exercise. JScrollBars are either horizontal or vertical. You must specify an initial value and a size for the thumb for the JScrollBar. The thumb is an indicator or elevator button located on both ends of the JScrollBar. It controls the JScrollBar, which will have to have its range of values set. After the requirements are determined, consult a Java 2 API to find an appropriate constructor. Examine the following example:

```
class JScrollBarStuff extends JFrame
{
   JScrollBar myScrollBar;
   myScrollBar = new JScrollBar(JScrollBar.HORIZONTAL,32,10,0,212);

   JScrollBarStuff ()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      Container c = getContentPane();
      c.add(myScrollbar);
      setSize(200,50);
      show();
   }
}
```

The output should resemble Figure 11-6.

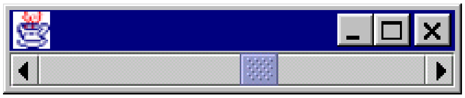


Figure 11-6: Output of JScrollBarStuff

This constructor was sophisticated but appropriate and used the following format:

```
public JScrollBar(int orientation, int initial Value,
                  int extent, int minValue, int maxValue)
// extent is the term used for the thumb
```

JTextField

A JTextField accepts an integer specifying the width according to the number of columns. Another constructor will allow the default text to be set. The following is an example:

```
class JTextFieldStuff extends JFrame
{
   JTextField myTextField = new JTextField("Some Text",25);

   JTextFieldStuff()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      Container c = getContentPane();
      c.add(myTextField);
      setSize(200,50);
      show();
   }
}
```

The output should resemble Figure 11-7.

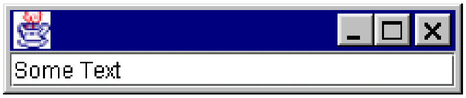

Some Text

Figure 11-7: Output of JTextFieldStuff

Other possible constructors for JTextField are as follows:

```
public JTextField(String text, int columns)
public JTextField(); // default constructor
```

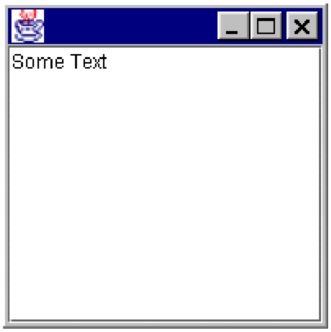
JTextArea

JTextArea has multiple uses such as a serving as a miniature editing window for entering text or displaying information. The JTextArea constructor takes two integers that represent the number of rows and the number of columns.

```
class JTextAreaStuff extends JFrame
{
   JTextArea myTextArea = new JTextArea(10,40);
   // specify number of rows and columns

   JTextAreaStuff()
   {
      setupGUI();
   }
   private void setupGUI()
   {
      Container c = getContentPane();
      c.add(myTextArea);
      setSize(200,200);
      show();
   }
}
```

The output should resemble Figure 11-8.



Some Text

Figure 11-8: Output of JTextAreaStuff

Some additional constructors for JTextArea are:

```
public JTextArea(String text)
public JTextArea(); // Default constructor
```

JScrollPane

One problem with the JTextArea is that it will not scroll to accommodate more text than can be displayed on one screen. Additionally, if a loaded image exceeds the boundaries of the container in which it is to be displayed, the container will not scroll. A JScrollPane adds this scrolling functionality to a component.

Scrolling behavior can be added to a JTextArea. The easiest way is to instantiate a JScrollPane class by passing the component you want to make scrollable in its constructor. For example:

```
JTextArea ta = new JTextArea();
JScrollPane sp = new JScrollPane(ta); // Scrollbars now added
```

This code will automatically add scrollbars to the JTextArea as needed. However, you might want to tailor the behavior of the scrollbars. Customization is possible with a series of methods available to the JScrollPane class such as:

```
// continuing with our example...

sp.setHorizontalScrollBarPolicy(int horizoptions);
// where horizoptions are:
// JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
// JScrollPane. HORIZONTAL_SCROLLBAR_AS_NEEDED (The default)
// JScrollPane.HORIZONTAL_SCROLLBAR_NEVER

sp.setVerticalScrollBarPolicy(int vertoptions);
// where vertoptions are:
// JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
// JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED (The default)
// JScrollPane.VERTICAL_SCROLLBAR_NEVER
```

Finally, do not add the JTextArea to your container. Just add the JScrollPane since it contains the component you want to scroll.

```
// using the add() of your container...
c.add(sp);
```

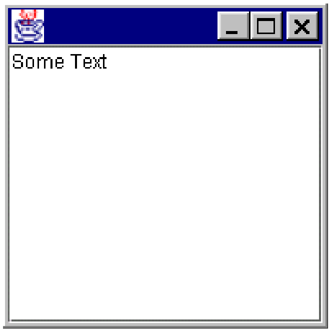
The following code is a complete example.

```
class JScrollPaneStuff extends JFrame
{
   JTextAreaStuff  myTextArea   = new JTextArea(10,40);
   JScrollPane     myScrollPane = new ScrollPane(myTextArea);

   JScrollPaneStuff()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      Container c = getContentPane();
      myScrollPane.setHorizontalScrollBarPolicy(
            JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
      myScrollPane.setVerticalScrollBarPolicy(
            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
      c.add(myScrollPane);
      setSize(200,200);
      show();
   }
}
```

The output should resemble Figure 11-9.



Some Text

Figure 11-9: Output of JScrollPaneStuff

JFileChooser

A JFileChooser is a small pop-up window that allows a user to select a file name from the local disk drive. The options for a JFileChooser allow it to be shown as an open file chooser or a closed file chooser. However, the JFileChooser does not perform the actual saving or loading of the file.

JFileChoosers are modal. If a modal component pops up, it blocks the user from interacting with other GUI components until the modal component is closed. For a JFileChooser, the modal feature can be disabled.

A programmer can instantiate a JFileChooser by using a default constructor or passing it a path to a directory to display. If a default constructor is used, the initial directory will be the current working directory. The ability of the JFileChooser is dependent on the local working environment.

Once instantiated, a JFileChooser can be displayed in either save mode or load mode with the following commands:

```
JFileChooser chooser = new JFileChooser(); // instantiated
int returnVal1 = chooser.showOpenDialog(this); // open a file
int returnVal2 = chooser.showSaveDialog(this); // save a file
```

Several considerations should be noted here. The showOpenDialog(this) and showSaveDialog(this) commands will visually display the JFileChooser. The keyword this is necessary because a modal component is associated with a parent window or frame. In this case, assume that this refers to the JFrame from which the JFileChooser was instantiated. Also note that both methods return an int. These processes are necessary because a user might select a file to save or load and then choose to cancel this operation. Even though the user cancelled this operation, the file selected will still be returned through the JFileChooser instance chooser. Even though a file name was chosen, the program must verify that the user did not click the Cancel button. This verification can be accomplished using some predefined constants of JFileChooser.

```
// continuation of above code to determine whether a Cancel
// button was clicked...

if(returnVal1 == JFileChooser.APPROVE_OPTION)
{
   // code to get the file name because the user selected
   // a file name and clicked the Load button
}
else if(returnVal1 == JFileChooser.CANCEL_OPTION)
{
   // code to respond to cancellation because the user
   // clicked the Cancel button
}
```

If the user selected a file, then it is necessary to obtain the actual name of the selected file:

```
// modification of preceding code to obtain the file name...

String fileName = "";
if(returnVal1 == JFileChooser.APPROVE_OPTION)
{
   fileName = chooser.getSelectedFile().getName();
}
else if(returnVal1 == JFileChooser.CANCEL_OPTION)
{
   // code to respond to cancellation because the user
   // selected the Cancel button
}
```

The method getSelectedFile() of JFileChooser returns a file object, whereas getName() is a method of the class File which returns the String representation of the file. A complete example follows:

```
class JFileChooserStuff extends JFrame
{
   // specify the starting directory to be the root of
   // the C drive
   JFileChooser chooser = new JFileChooser("c:\\");
   String fileName = "";
   {
      setupGUI();
   }
   private void setupGUI()
   {
      setSize(300,300); // just an empty frame right now
      show();
      int retVal = chooser.showOpenDialog(this);
      if(retVal == JFileChooser.APPROVE_OPTION)
      {
         fileName = chooser.getSelectedFile().getName();
         System.out.println("Selected file: " + fileName);
      }
      else if(retVal == JFileChooser.CANCEL_OPTION)
      {
         System.out.println("Operation cancelled");
      }
    }
}
```

The output should resemble Figure 11-10.

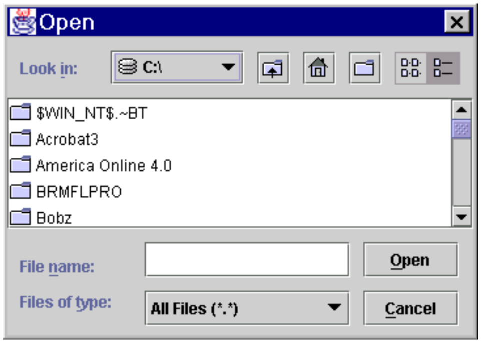


Figure 11-10: Output of JFileChooserStuff

Containers

In the introduction of JButtons, it was mentioned that some holes existed in the code. The declaration of the class used the word JFrame, and it was not yet explained how the example classes became visible. These topicswill now be approaced, starting with an explanation of the JFrame class and Containers in general.

In Java, components need to be placed into containers. Two types of containers exist: top-level and lower-level. Top-level containers like JFrame and Window cannot be contained in other containers. Lower-level containers like JPanel and JScrollPane can exist only inside another container. Containers are commonly nested within other containers to create complicated GUIs. JFrame and JPanel are the most frequently used Swing containers.

JFrame

JFrame is the primary container used to create stand-alone applications. Objects of the Window class do not contain borders or a title bar, both of which are typical prerequisites for a Window application; therefore, Window objects are not used as often as JFrames. Because JFrame is a class, it can readily be extended to inherit all of its capabilities. In this example, the JFrameStuff class extends JFrame.

```
class JFrameStuff extends JFrame
{
   JFrameStuff ()
   {
      setupGUI();
   }
   private void setupGUI()
   {
      setSize(150,150);
      // set the size in pixels
      setTitle("Practice");
      setCursor(Frame.HAND_CURSOR);// just an example;
      show();// make the Frame visible
   }
}
```

The output should resemble Figure 11-11.

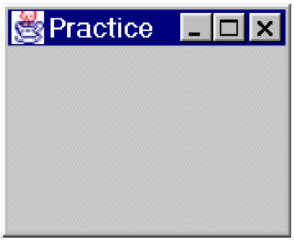


Figure 11-11: Output of JFrameStuff

Excluding the title, the Jframe display size is 0 x 0 when it is first created. To see the contents of a JFrame, you must invoke the setSize() and the setVisible() methods. This has been done throughout the previous JFrame examples.

Adding components to a JFrame is not a direct process because the JFrame is a single component that is created from a combination of several components. These components are JRootPane, JLayeredPane, ContentPane, and GlassPane. This level of sophistication is necessary for Swing to include such features as Multi-Document Interface (MDI) and drag-and-drop. If you do not need to create an MDI application, then the approach is straightforward. To add a component to a JFrame, it is first necessary to get the ContentPane associated with the JFrame. From that point, add components to the ContentPane. By adding components directly to the ContentPane, components are indirectly added to the JFrame.

Continue with the setupGUI() method, and attempt to add a single JButton to your JFrame.

```
class JFrameStuff extends JFrame
{
   JButton myButton = new JButton("My Button");
   JFrameStuff ()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      // First, associate the ContentPane with the JFrame
      Container c = getContentPane();
      // Next, add the JButton to this ContentPane;
      // the JButton is automatically added to the JFrame
      c.add(myButton)
      setSize(150,150);// Set the size in pixels.
      setTitle("Practice");
      setCursor(Frame.HAND_CURSOR);
      show();
   }
}
```

The output should resemble Figure 11-12.

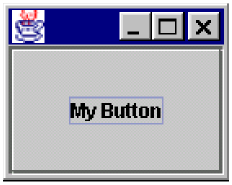


Figure 11-12: Output of JFrameStuff

JPanel

JPanels are simple Containers that do not pop up or appear as JFrames do. Instead, as low-level containers, their job is to contain Components such as JButtons, JTextFields, or other low-level Containers. JPanels also serve as convenient Components upon which to draw graphics. In the following example, you will instantiate a JPanel, but its usefulness will not become fully apparent until later.

The method add(), which is a method inherited from Container class allowing for Components to be added to a Container.

```
class JPanelStuff extends JFrame
{
   JPanel myPanel = new JPanel();

 JPanelStuff ()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      Container c = getContentPane();
      c.add(myPanel); // Add panel to frame; you will not
      setSize(125,125);// be able to see it
      setTitle("Practice");
      show();
   }
}
```

Your output should resemble Figure 11-13

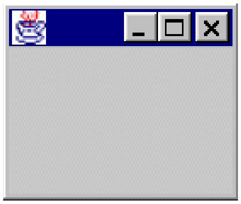


Figure 11-13: Output of JPanelStuff

JAVABEANS

The JavaBeans specification is a software component model similar in nature to Microsoft's COM. JavaBeans are individual components that are designed according to a strict specification so that they can be easily dropped into an application using visual tools. For example, a Web browser JavaBean would encapsulate all the functionality of a Web browser into an easily manageable component within a visual programming environment.In the same manner all Swing components, such as JButton or a JTextField, are JavaBeans. Libraries of JavaBeans are available from various vendors. JavaBeans can be used to rapidly develop sophisticated applications by bringing together several separately developed components.

SUN CERTIFICATION

In the Sun Certified Java Programmer examination, TextArea, TextField, and List are all tested. List implements the drop-down list box and a scrolling list box. The List constructor requires the number of rows visible and a boolean declaring whether or not multiple selections are allowed. After the List object is created, it is populated with the addItem() method. The following is code for a List:

```
List myList = new List(2, false);
myList.addItem("Pizza");
myList.addItem("Spaghetti");
myList.addItem("Lasagna");
myList.addItem("Minestrone");
```

At this point, the List object myList would be added to a Container.

The following Component methods should be understood:

```
setEnabled()
setVisible()
setSize()
setForeground()
setBackground()
show()
```

SUMMARY

This chapter covered the Swing class structure and organization. Components were defined, and an ImageIcon, a JButton, a JScrollBar, and other components were created. Many containers were defined and created. These graphical Components can be assembled in the next lesson to create complex graphical interfaces.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What does this piece of code do?

```
ImageIcon icon = new ImageIcon(“save_button.gif”);
JButton save = new JButton(icon);
```

2. What does a JScrollPane do?

3. What is a modal component?

CHAPTER 12

Layout Managers

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Define a layout manager.
- Set a layout manager for a Container.
- Effectively use each FlowLayout, GridLayout, BorderLayout, and BoxLayout.
- Nest Containers and Layout Managers to form more complex GUI layouts.
- Separate a complex design into its Component Containers and Layout Managers.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is a Layout Manager?

2. In which package are the Layout Managers located?

INTRODUCTION

Java makes the process of creating graphical user interfaces, GUI's, simple. In this chapter, the tools used for controlling the look of GUI's will be discussed. These Java tools are known as Layout Managers.

Stop now and view the following video presentation on the Interactive Learning CD-ROM (jCert):

Java Programming Fundamentals

Layout Manager

WHAT IS A LAYOUT MANAGER?

Containers can hold other Components, but they have no way to determine where these Components should be placed. This task is performed by layout managers.

The Java API provides a variety of layout managers:

- BorderLayout
- BoxLayout
- CardLayout
- FlowLayout
- GridBagLayout
- GridLayout
- OverlayLayout
- ScrollPaneLayout
- ViewportLayout

Java also allows absolute positioning. That is, specifying a components position by pixel count. However, this technique is discouraged due to variations in platforms and resolutions. It is best to let a layout manager address cross-platform issues.

Layout managers can be frustrating. Although you may have an idea for the interface's presentation, layout managers must consider many elements before your Components are actually displayed. Thus, you must often compromise.

This chapter will concentrate on FlowLayout, GridLayout, BorderLayout, and BoxLayout. These four layout managers offer great functionality while remaining relatively easy to use. Other layout managers provide greater flexibility and specific case functionality. For instance, CardLayout can be used when a tabbed index system is needed. GridBagLayout and OverlayLayout are the most flexible of the layout managers; however, they are somewhat cumbersome to implement. ScrollPaneLayout and ViewportLayout are component-specific managers.

FlowLayout

Each layout manager is named for its function. The FlowLayout centers Components on each line in a flowing manner. A line is the length of the container and the height of the tallest component on the line. Components are added to the right of the first component and the line is centered again. When no additional room is available on that line of the Container, the FlowLayout drops down to the next line. Figure 12-1 shows a Frame with five buttons set to FlowLayout. The Frame has been resized several ways.

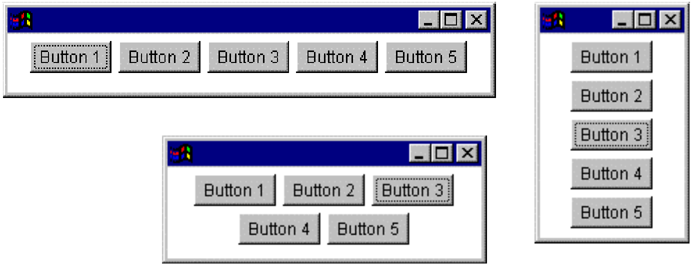


Figure 12-1: FlowLayout resized

To use the FlowLayout, follow these steps:

1. Set the layout of the Container using the `setLayout(LayoutManager mgr)` method inherited from Container.
2. Add the Components using the method `add(Component comp)`, also inherited from Container. For example:

```
class FlowLayoutStuff extends JFrame
{
   JButton b1 = new JButton("Button 1");
   JButton b2 = new JButton("Button 2");
   JButton b3 = new JButton("Button 3");

   FlowLayoutStuff()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      Container c = getContentPane();
      c.setLayout(new FlowLayout());  // Set the layout
                          // before adding the
      c.add(b1);          // components.
      c.add(b2);          // Notice that you do not
      c.add(b3);          // need a reference to the
      setSize(200,100);    // FlowLayout object. This
      show();    // is typical.
   }
}
```

Your output should resemble Figure 12-2.

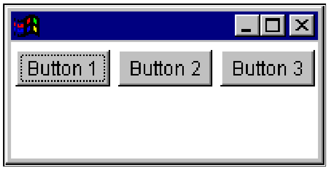


Figure 12-2: Output for FlowLayoutStuff

GridLayout

GridLayout is a layout based on a grid-like format. The constructor to the GridLayout class requires the number of rows and columns necessary to form the grid. The grid is filled in the following manner. A column with the specified number of rows is created and Components are added from top to bottom. Columns will be added one at a time as columns are filled. When a column is added, the components will be moved so that they are positioned left to right and top to bottom in the order that they were added. The number of rows will not be increased. If more Components are added than will fit, another column will automatically be added. Component size is entirely controlled by the grid, all components in the grid are the same size. As more components are added, the size of each component is reduced. Using the following code, create a GridLayout that has two rows and three columns.

```
class GridLayoutStuff extends JFrame
{
   JButton b1 = new JButton("Button 1");
   JButton b2 = new JButton("Button 2");
   JButton b3 = new JButton("Button 3");
   JButton b4 = new JButton("Button 4");
   JButton b5 = new JButton("Button 5");
   JButton b6 = new JButton("Button 6");

   GridLayoutStuff()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      Container c = getContentPane();
      c.setLayout(new GridLayout(2,3));
      c.add(b1);
      c.add(b2);
      c.add(b3);
      c.add(b4);
      c.add(b5);
      c.add(b6);
      setSize(200,100);
      show();
   }
}
```

The output should resemble Figure 12-3.

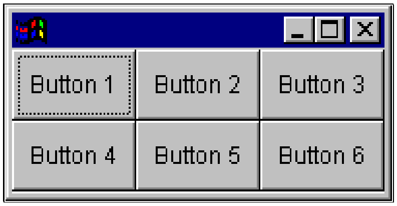


Figure 12-3: Output for GridLayoutStuff

BorderLayout

BorderLayout is an unusual, but useful layout. In this case, the layout is divided into five regions, as shown in Figure 12-4.

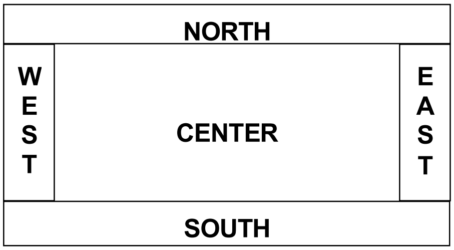


Figure 12-4: Example of BorderLayout

When components are added, you must specify the region in which the component is to be added. To set a BorderLayout and add a Button, called `myButton`, to the West region, enter the following code:

```
c.setLayout(new BorderLayout());
c.add(myButton, BorderLayout.WEST);
```

Important aspects of BorderLayout are:

- Only one component can be added to any one area. If more than one is added, you will only see the last one added.
- The West and East areas will give the added component its preferred width, but will change the component height.
- The North and South areas will give the added component its preferred height, but will change the component width.
- The Center region will fill any area between the other four areas and a component added to the Center will be this size. Center is the default are if none is specified.
- JFrames are, by default, set to BorderLayout. However, always explicitly set the layout manager, instead of assuming one.

```
class BorderLayoutStuff extends JFrame
{
   JButton b1 = new JButton("Button 1");
   JButton b2 = new JButton("Button 2");
   JButton b3 = new JButton("Button 3");
   JButton b4 = new JButton("Button 4");
   JButton b5 = new JButton("Button 5");

   BorderLayoutStuff()
   {
      setupGUI();
   }
```

```
      private void setupGUI()
      {
         Container c = getContentPane();
         c.setLayout(new BorderLayout());
         c.add(b1, BorderLayout.NORTH);
         c.add(b2, BorderLayout.EAST);
         c.add(b3, BorderLayout.SOUTH);
         c.add(b4, BorderLayout.WEST);
         c.add(b5, BorderLayout.CENTER);
         setSize(300,300);
         show();
      }
   }
```

Your output should resemble Figure 12-5.

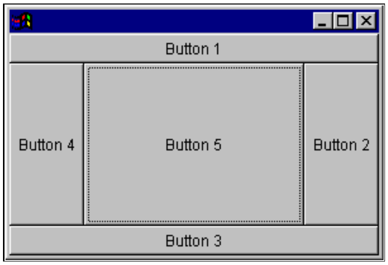


Figure 12-5: Output of BorderLayoutStuff

BoxLayout

The BoxLayout is similar to the FlowLayout. However, a BoxLayout allows either a horizontal or a vertical layout of components. In the following basic example, you will create two JPanels and populate them with JButtons to demonstrate the BoxLayout.

```
class BoxLayoutStuff extends JFrame
{
   JPanel horPanel = new JPanel();
   JPanel vertPanel  = new JPanel();
   JButton b1 = new JButton("b1");
   JButton b2 = new JButton("b2");
   JButton b3 = new JButton("b3");
   JButton b4 = new JButton("b4");
   JButton b5 = new JButton("b5");
   JButton b6 = new JButton("b6");

   BoxLayoutStuff()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      Container c = getContentPane();
      c.setLayout(new GridLayout(1,2));
      horPanel.setLayout(new BoxLayout(horizPanel,BoxLayout.X_AXIS));
      // Set the h-layout
      vertPanel.setLayout(new BoxLayout(vertPanel,BoxLayout.Y_AXIS));
      // Set the v-layout
```

```
        horPanel.add(b1);
        horPanel.add(b2);
        horPanel.add(b3);
        vertPanel.add(b4);
        vertPanel.add(b5);
        vertPanel.add(b6);
        c.add(horPanel);
        c.add(vertPanel);
        setSize(400,150);
        setVisible(true);
    }
}
```

The output should resemble Figure 12-6.

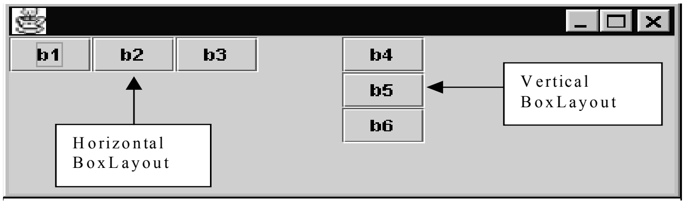


Figure 12-6: Output of BoxLayoutStuff

SWING

Swing introduces a new component called Box. This component is really just a container with the BoxLayout already set. However, it does offer some useful methods and two useful components: a strut and a glue.

Strut

A strut is an invisible component that allows either horizontal or vertical (fixed) spacing to be added to layouts. Struts are created using a factory method of the Box class. For example, a horizontal and vertical strut, each 15 pixels in width and height, are created as follows:

```
Component hStrut = Box.createHorizontalStrut(15);
Component vStrut = Box.createVerticalStrut(15);
```

Add these components to your layout as usual to achieve the desired effect. Notice the slight modification to the code regarding the struts; this approach is more typical:

```
class BoxLayoutStuff extends JFrame
{
   JPanel horPanel = new JPanel();
   JPanel vertPanel  = new JPanel();
   JButton b1 = new JButton("b1");
   JButton b2 = new JButton("b2");
   JButton b3 = new JButton("b3");
   JButton b4 = new JButton("b4");
   JButton b5 = new JButton("b5");
   JButton b6 = new JButton("b6");

   BoxLayoutStuff()
   {
      setupGUI();
   }
   private void setupGUI()
   {
      Container c = getContentPane();
      c.setLayout(new GridLayout(1,2));
      horPanel.setLayout(new BoxLayout(horizPanel,BoxLayout.X_AXIS));
      // Set the h-layout
      vertPanel.setLayout(new BoxLayout(vertPanel,BoxLayout.Y_AXIS));
      // Set the v-layout
      horPanel.add(b1);
      horPanel.add(Box.createHorizontalStrut(15));
      horPanel.add(b2);
      horPanel.add(Box.createHorizontalStrut(15));
      horPanel.add(b3);
      vertPanel.add(b4);
      vertPanel.add(Box.createVerticalStrut(15));
      vertPanel.add(b5);
      vertPanel.add(Box.createVerticalStrut(15));
      vertPanel.add(b6);
      c.add(horPanel);
      c.add(vertPanel);
      setSize(400,150);
      show();
   }
}
```

The output should resemble Figure 12-7.

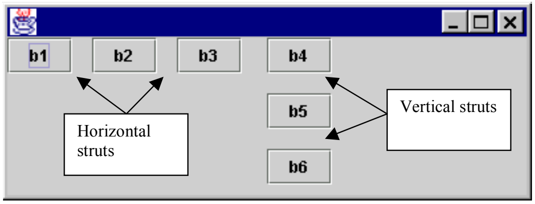


Figure 12-7: Output of BoxLayoutStuff (with struts)

Glue

A glue component allows control of the positions of other fixed-size components in the layout. This invisible component expands to absorb extra space in the container. Glue components will consume as much horizontal or vertical space as possible. Create horizontal and vertical glue components using factory methods of the Box class:

```
Component hGlue = Box.createHorizontalGlue();
Component vGlue = Box.createVerticalGlue();
```

Add these components to the layout as usual, and note the effects:

```
class BoxLayoutStuff extends JFrame
{
   JPanel horPanel = new JPanel();
   JPanel vertPanel  = new JPanel();
   JButton b1 = new JButton("b1");
   JButton b2 = new JButton("b2");
   JButton b3 = new JButton("b3");
   JButton b4 = new JButton("b4");
   JButton b5 = new JButton("b5");
   JButton b6 = new JButton("b6");

   BoxLayoutStuff()
   {
      setupGUI();
   }
```

```
      private void setupGUI()
      {
         Container c = getContentPane();
         c.setLayout(new GridLayout(1,2));
         horPanel.setLayout(new BoxLayout(horizPanel,BoxLayout.X_AXIS));
         // Set the h-layout
         vertPanel.setLayout(new BoxLayout(vertPanel,BoxLayout.Y_AXIS));
         // Set the v-layout
         horPanel.add(Box.createHorizontalGlue());
         horPanel.add(b1);
         horPanel.add(b2);
         horPanel.add(b3);
         vertPanel.add(Box.createVerticalGlue());
         vertPanel.add(b4);
         vertPanel.add(b5);
         vertPanel.add(b6);
         c.add(horPanel);
         c.add(vertPanel);
         setSize(400,150);
         show();
      }
}
```

The output should resemble Figure 12-8.

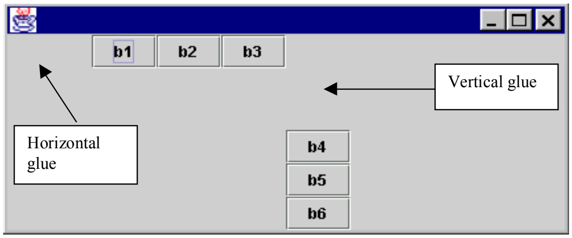


Figure 12-8: Output of BoxLayoutStuff (with glue)

Combining layouts

To create a specific GUI in spite of the apparent limitations of each layout, use a combination of layouts. This is nesting. Combine your knowledge of the layout managers with the use of different containers to achieve sophisticated results. As an example, try to duplicate the layout shown in Figure 12-9.

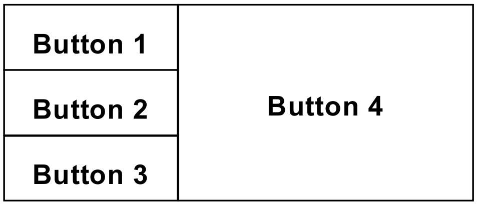


Figure 12-9: Example of a sophisticated layout

You can approach this task in multiple ways. This layout strongly resembles a BorderLayout with three JButtons in the West region, and a fourth JButton in the Center region. However, if you add multiple JButtons in any one region, only the last JButton will be displayed. Therefore, you must further separate the problem. The West region resembles a GridLayout with three rows and one column. You can summarize the solution as follows:

1. Create a JFrame with a BorderLayout.
2. Create a JPanel with a GridLayout of three by one, and add three JButtons.

3. Add the JPanel to the West region.
4. Add the fourth JButton to the Center region.

```
class NestedLayoutStuff extends JFrame
{
   JButton b1 = new JButton("Button 1");
   JButton b2 = new JButton("Button 2");
   JButton b3 = new JButton("Button 3");
   JButton b4 = new JButton("Button 4");
   JPanel west = new JPanel();

   NestedLayoutStuff()
   {
      setupGUI();
   }

   private void setupGUI()
   {
      Container c = getContentPane();
      c.setLayout(new BorderLayout());
      west.setLayout(new GridLayout(3,1));
      west.add(b1);
      west.add(b2);   // Add Buttons to the JPanel
      west.add(b3);
      c.add(BorderLayout.WEST, west);   // Add JPanel
      c.add(BorderLayout.CENTER,b4);
      setSize(250,150);
      show();
   }
}
```

Your output should resemble Figure 12-10.

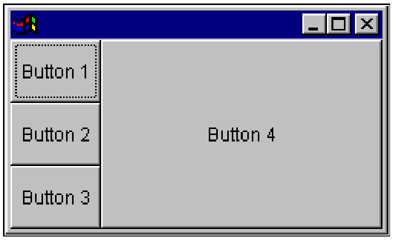


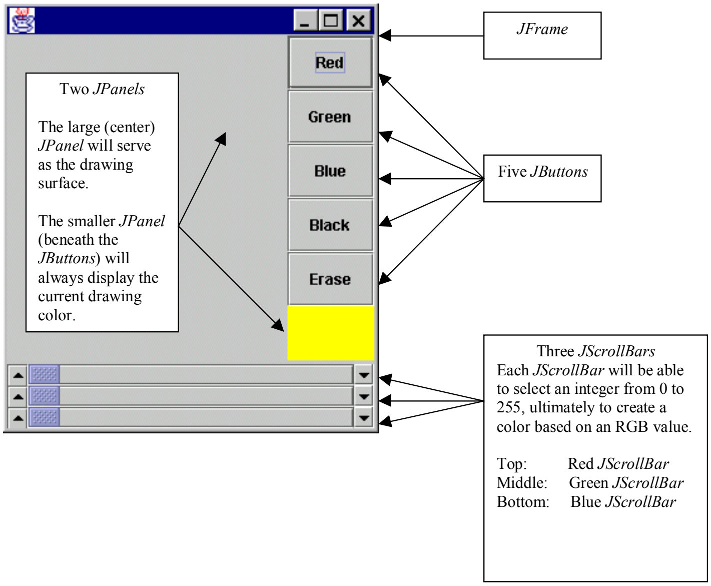
Figure 12-10: Output of NestedLayoutStuff

Other possibilities exist. Perhaps a BoxLayout in the West JPanel, or a FlowLayout, would be more accommodating. The layout managers offer a great deal of flexibility and creativity. Remember to test your layouts under various conditions, such as different screen resolutions and operating systems.

GRADUATING TASK #3: CREATING SOPHISTICATED LAYOUTS

In this exercise, you will create a moderately sophisticated GUI which will serve as the front end for a painting program that is similar to the Paint program in Windows. You will develop this painting program over the next several sections, the general form is provided here. Experiment with layout techniques to find the most suitable solution.

1. Create a GUI as shown in Figure 12-11. Ultimately, this GUI will become fully functional with the ability to draw in many predefined and user-defined colors. Note that this entire GUI consists of one JFrame, four JPanels, five JButtons, and three JScrollBars.



JFrame

Five *JButtons*

Two *JPanels*

The large (center) *JPanel* will serve as the drawing surface.

The smaller *JPanel* (beneath the *JButtons*) will always display the current drawing color.

Three *JScrollBars*
Each *JScrollBar* will be able to select an integer from 0 to 255, ultimately to create a color based on an RGB value.

Top:	Red *JScrollBar*
Middle:	Green *JScrollBar*
Bottom:	Blue *JScrollBar*

Figure 12-11: Model GUI

2. (*Optional*) Change the background colors of the three JScrollBars to red, green, and blue to their representative colors.
3. (*Optional*) Change the background colors of the red, green, blue, and black `JButtons` to their representative colors.
4. (*Optional*) Change the background color of the small JPanel to black, which is the default drawing color in Java.
5. (*Optional*) Add three JLabels to the left of the three JScrollBars, labeling the color that each JScrollBar represents.

SUN CERTIFICATION

Three layouts are covered on the Java Certified Programmer exam: BorderLayout, FlowLayout, and GridLayout. Additional aspects of the layouts should be reviewed.

- Remember that some of the layouts resize Components, but only some Components can be resized. A Button, Label, and TextField can be stretched if necessary, but the Checkbox component cannot be stretched.
- Each type of Container has a default LayoutManager. These are listed in Table 12-1.

Table 12-1: Default LayoutManagers

Container	Default LayoutManager
JFrame	BorderLayout
JPanel	FlowLayout
Dialog	BorderLayout
Window	BorderLayout
Japplet	FlowLayout

- Other layout managers can be employed in a Container using the `setLayout()` method of Container (and its derived classes). The Panel uses the FlowLayout, therefore JApplet will also use the FlowLayout by default, because JApplet is derived from Panel.

SUMMARY

FlowLayout, GridLayout, BorderLayout and BoxLayout are the four most common layout managers used in Java. A common techniques of nesting Containers and layout managers is to create more sophisticated designs. Each layout manager has a unique affect on Components and Containers. These methods and affects should be reviewed before continuing. You will learn how to place shapes, text, and images onto a graphical Component in the next chapter.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. Which Layout Manager is used by (JRE) panels (JRE)default?

2. How many placement fields are there in the BorderLayout and what are they?

3. Which Layout Manager will not change the size of its components?

CHAPTER 13

Graphics

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Identify the AWT class structure for graphics.
- Explain how the Graphics class is realized through the graphics context.
- Gain access to a Container's graphic context by overriding the `paint(Graphics g)` method.
- Use methods of the Graphics class via the graphics context, including `drawString()` and `drawRect()`.
- Effectively use the Color and Font classes.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. Where are Graphics in Java?

   ....

   ....

2. Which Java class would you use to to draw text and shapes on the screen?

   ....

   ....

3. About how many different colors is Java capable of displaying?

4. Why are there only a few fonts available to all Java Virtual Machines?

INTRODUCTION

Almost all modern programs contain some sort of graphics or animation. In response, the Java `Graphics` class provides many methods to a programmer. These methods include tools to draw almost all basic shapes. These basic shapes can be combined to create complicated designs.

With the introduction of Java 1.2, the `Graphics` class was extended by `Graphics2D`. This new class provides the programmer more sophisticated control over geometry, coordinate transformations, color management, and text layout. The tools are now advanced enough that a programmer can create and animate images with relative ease.

WHAT ARE GRAPHICS IN JAVA?

Java graphics can be thought of as any image drawn onto any Java component. These components include all `AWT Components` and all `Swing Components`. Java provides a rich set of classes to create graphics. In this chapter, four classes in the AWT package that are useful for graphics will be discussed: `Graphics`, `Graphics2D`, `Color`, and `Font`.

GRAPHICS CLASS

The **Graphics** class is an abstract class. As a cross-platform language, Java cannot implement the necessary hard code to properly handle every graphics system available. For this reason, the Java Virtual Machine must implement this abstract class. The various implementations of the Graphics class will be handled properly as long as a programmer adheres to the methods provided.

Abstract classes cannot be instantiated. Therefore, programmers must request a Graphics object from a Component, which is accomplished by using a method of the form:

```
    component.getGraphics();
```

This method is defined in Component and returns a Graphics object. Therefore, any class that is a subclass of Component can be drawn upon. The functionality of the Graphics class can be accessed even though it is an abstract class.

Programmers actually work with the **graphics context**, which is the AWT implementation of the methods defined in the Graphics class. In essence, the graphics context is a wrapper representing the surface of a component. Drawing is never performed directly to the surface of a component but to the graphics context, which draws to the surface of the component.

Two methods are used to paint components. If a component is a member of the Swing package, the method is

```
    public void paintComponent(Graphics g)
```

otherwise, the method is

```
    public void paint(Grapics g)
```

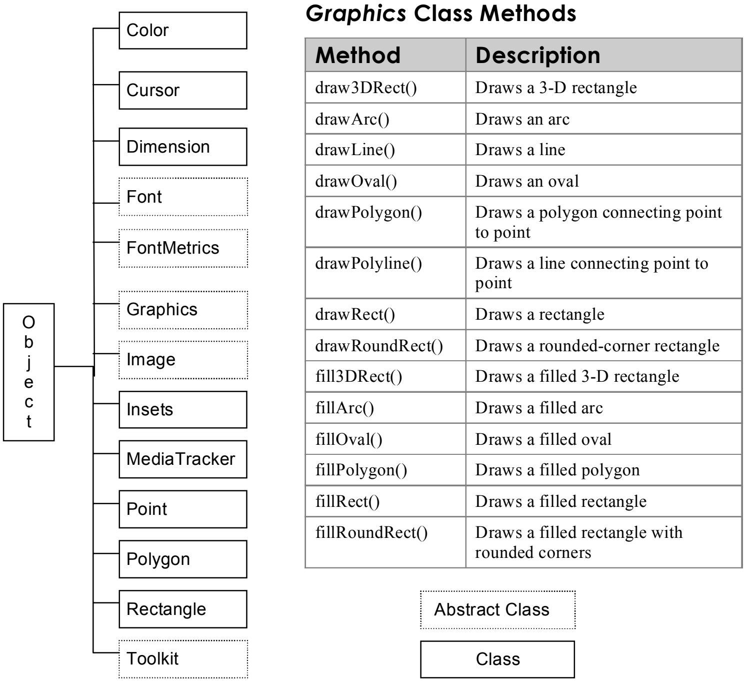
The graphics context can be requested from the `Component` or `Image` with which you want to work. Usually, you will override the painting method for that object. The following code provides a simple example.

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);// This will redraw the background
    //color; otherwise your component will be transparent, and
    // might draw to the component beneath it
    // Additional code for drawing would be included
}
```

If the component to which you want to paint is a subclass of `Container` alone, then you can override the following method:

```
public void paint(Graphics g)
{
    super.paint(g); // Frame might have problems refreshing
    // its onscreen image without calling its super method
    // first
    // Additional code to do some drawing...
}
```

Figure 13-1 lists the many classes available in the AWT for creating graphics. Some of the methods made available in the Graphics class itself.



Graphics Class Methods

Method	Description
draw3DRect()	Draws a 3-D rectangle
drawArc()	Draws an arc
drawLine()	Draws a line
drawOval()	Draws an oval
drawPolygon()	Draws a polygon connecting point to point
drawPolyline()	Draws a line connecting point to point
drawRect()	Draws a rectangle
drawRoundRect()	Draws a rounded-corner rectangle
fill3DRect()	Draws a filled 3-D rectangle
fillArc()	Draws a filled arc
fillOval()	Draws a filled oval
fillPolygon()	Draws a filled polygon
fillRect()	Draws a filled rectangle
fillRoundRect()	Draws a filled rectangle with rounded corners

Figure 13-1: AWT class hierarchy (graphics section)

Typically, to create graphics (either text or shapes), the `paintComponent(Graphics g)` method, derived from `JComponent`, will be overridden. The coordinate system used to position graphics is consistent with computer graphics coordinates of the form (X,Y) where the origin (0,0) is the upper-left corner of the component. X increases by one for every pixel moved down from the origin.

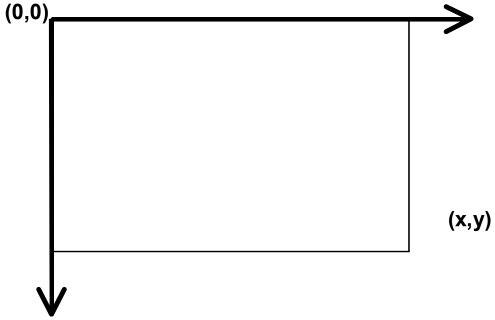


Figure 13-2: Coordinate system for positioning graphics

You can perform the `Hello World` program graphically with the following code:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Window.*;
import javax.swing.*;

class PaintingStuff extends JPanel
{
    PaintingStuff ()
    {
        // empty
    }
    // The AWT passes the graphic context of the frame as a
    // parameter to the paint method.
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g); // Avoid transparency issues.
        g.setColor(Color.blue); // The foreground must be
        // different from the background, or you will not see
        // the result (we will discuss this topic shortly).
        g.drawString("Hello World",70,100);
    }
}
```

Of course, this `JPanel` must be added to a `JFrame`. For example:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Window.*;
import javax.swing.*;

class StartPaintingStuff extends JFrame
{
    PaintingStuff pStuff = new PaintingStuff();

    StartPaintingStuff()
    {
        setupGUI();
    }

    private void setupGUI()
    {
        Container c = getContentPane();
        c.setLayout(new BorderLayout());
        c.add(BorderLayout.CENTER,pStuff);
        setSize(200,200);
        setVisible(true);
    }
}
```

The output should resemble Figure 13-3.

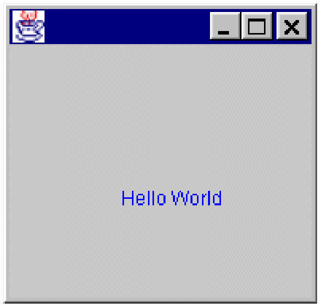


Figure 13-3: Output of PaintingStuff

Overriding `paintComponent(Graphics g)` is usually the preferred method for accessing the graphics context of a `JComponent`. Remember that for a `Frame`, the method is `paint(Graphics g)`. Sometimes programmers will try to capture the graphics context used elsewhere in their code by using an inherited method of `Component` called `getGraphics()`. This way, the programmer is not forced to remain in the specific inherited method.

This technique can be difficult. The graphics context is generated by the AWT spontaneously, and maintaining a reference to it may cause problems later. If you use this technique, be sure to attain a fresh reference to the graphics context each time you use it. For example, you could run the Hello World program without committing to overriding a specific method as follows:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Window.*;
import javax.swing.*;

class PaintingStuff extends JPanel
{
    PaintingStuff ()
    {
        // empty
    }
    // The AWT passes the graphics context of the frame as a
    // parameter to the paint method.

    private void drawSomething()
    {
        Graphics g = getGraphics(); // Request the Graphics context.
        g.setColor(Color.red);
        g.drawString("Hello World",70,100);
    }
}
```

The drawSomething() method can be invoked as follows:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Window.*;
import javax.swing.*;

class StartPaintingStuff extends JFrame
{
    PaintingStuff pStuff = new PaintingStuff();
    StartPaintingStuff()
    {
        setupGUI();
        pStuff.drawSomething();
    }

    private void setupGUI()
    {
        Container c = getContentPane();
        c.setLayout(new BorderLayout());
        c.add(BorderLayout.CENTER,pStuff);
        setSize(200,200);
        setVisible(true);
    }
}
```

The result is almost identical to the previous example; the only difference is the manner in which the graphics context was obtained.

One surprising aspect is the frequency with which the AWT calls the `paint(Graphics g)` method. Each time you drag, resize, cover, or uncover some part of a window, the AWT will update its graphics context and automatically call the `paint(Graphics g)` method. The `paint(Graphics g)` method should contain all the code to redraw the entire component. You may also force the AWT to call `paint(Graphics g)` if data affecting the output of your program has changed. In this case, a call to `repaint()` will force the AWT to call the component's `paint(Graphics g)` method. The `repaint()` method performs some updating but ultimately calls the `paintComponent(Graphics g)` or the `paint(Graphics g)` method.

> *TECH TIP:*
>
> *Noted that refreshing of the screen is not guaranteed. The virtual machine will repaint the screen as soon as possible; however, repainting is controlled strictly. The reason is to prevent painting from consuming all of the processor cycles.*

Java programmers should become comfortable with all the methods available in the `Graphics` class. The following sub-sections will introduce some of these methods.

drawString()

The method drawString() in the `Graphics` class allows you to draw text using the graphics context's current font and color. As seen in the code examples throughout this chapter, you can call the `drawString()` method in the following manner.

```
public void paint(Graphics g)
{
      g.drawString("Hello World",70,100);
}
```

The first parameter, `Hello World`, refers to the string that is to be painted. The second and third parameters refer to the desired X and Y position on the coordinate system.

drawLine()

Sometimes the `Graphics` class will need to be used to draw a line or a group of lines, which can be accomplished by calling the `drawLine()` method. The following applet is an example using `drawLine()`, to draw the word `java`.

```
import javax.swing.*;  //import Applet class
import java.awt.Graphics; //import Graphics class

public class DrawLineTest extends JApplet
{
    public void paint(Graphics g)
    {
        g.drawLine(45, 75, 25, 75);
        g.drawLine(35, 75, 35, 115);
        g.drawLine(25, 115, 35, 115);
        g.drawLine(55, 75, 55, 115);
        g.drawLine(65, 75, 65, 115);
        g.drawLine(55, 75, 65, 75);
        g.drawLine(55, 85, 65, 85);
        g.drawLine(75, 75, 80, 115);
        g.drawLine(80, 115, 85, 75);
        g.drawLine(95, 75, 95, 115);
        g.drawLine(105, 75, 105, 115);
        g.drawLine(95, 75, 105, 75);
        g.drawLine(95, 85, 105, 85);
    }
}
```

This example shows how `drawLine()` draws a line, using the current color, between the point specified by the first two x-y parameters and the point specified by the second two x-y parameters.

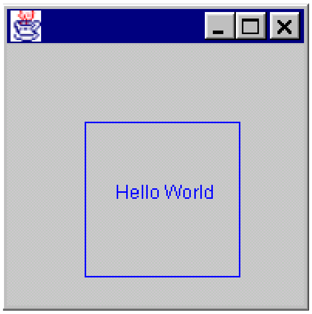
drawRect()

If you want to draw a rectangle around the text, you can implement one of the methods available to the `Graphics` class such as `drawRect()`.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Window.*;
import javax.swing.*;

class PaintingStuff extends JPanel
{
    PaintingStuff()
    {
        setupGUI();
    }
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        g.setColor(Color.blue);
        g.drawString("Hello World",70,100);
        g.drawRect(50,50,100,100);
    }
}
```

Your output should resemble Figure 13-4.



Hello World

Figure 13-4: Output of PaintingStuff (with rectangle)

drawImage()

You can also draw an entire image using the `drawImage()`. An example of how this can be accomplished is in the following code.

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

public class Test1 extends JApplet
{
    private Image i;

    public void init()
    {
        i = getImage(getDocumentBase(),"9a.jpg");
    }
    public void paint(Graphics g)
    {
        g.drawImage(i, 0, 0, this);
    }
}
```

The drawImage() method in this example attempts to draw the Image i at the specified position (0,0) with the ImageObserver Test1. For the level of this discussion, the image observer can be thought of simply as the component on which to display the image. The image is obtained from a file using a call to getDocumentBase() which returns the path to the image 9a.jpg. The drawImage() method returns immediately in all cases, even if the image is not yet loaded, in which case the drawImage returns a boolean with the value false. Checking the return value of this method is recommended. If the value is false, the program should wait for the image to be drawn. Otherwise, as more of the image becomes available, the process that draws the image will alert the specified image observer, and the component will be updated.

Color class

In the perspective of object-oriented programming, color is an object, which can be used as a property of another object. Therefore, Java contains the Color class. Color can be determined in many ways, such as an RGB value by using the method Color(int R, int G, int B) or by simply specifying the color: Color.red. Once set, the color can be adjusted using methods such as Color.brighter(). Color fits nicely into the framework of an object.

A Color object may be readily used to change the current color of the graphics context from its default color, black. The following code demonstrates three ways to set the color of the graphics context using the setColor() method of the Graphics class and a properly created Color object.

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    g.setColor(Color.red);
        // M1 - using a constant.
    g.setColor(new Color(255,0,0));
        // M2 - using a constructor.
    g.setColor(new Color(256*256*255));
        // M3 - using a constructor.
}
```

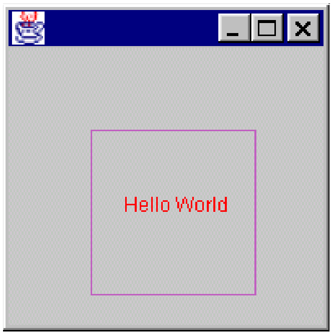
And now the code:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Window.*;
import javax.swing.*;

class PaintingStuff extends JPanel
{
    PaintingStuff ()
    {
        setupGUI();
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        g.setColor(Color.red);// M1
        g.drawString("Hello World",70,100);
        g.setColor(new Color(200,100,200));// M2
        g.drawRect(50,50,100,100);
    }
}
```

Your output should resemble Figure 13-5.



Hello World

Figure 13-5: Output of PaintingStuff (with colors)

Font class

The **Font** class is also available in Java and it works much like the `Color` class. A `Font` is defined by its `name` (Helvetica, Times Roman, etc.), its `style` (bold, italic, plain, etc.), and its `size` (10,12,14, etc.). The constructor of a `Font` object will accept these parameters.

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    g.setFont(new Font("Helvetica", Font.BOLD, 15));
}
```

As is often the case with Java, you must consider how a `Font` object translates from one system to another. To successfully cross platforms, Java makes the following `Fonts` available to any JVM:

- Serif
- Sans Serif
- Monospaced

- Dialog
- Dialog Input

You are not limited to these fonts. The Toolkit class contains a method that will retrieve all the fonts available to the system.

The final modification to your practice code will set a font.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Window.*;
import javax.swing.*;

class PaintingStuff extends JPanel
{
   PaintingStuff ()
   {
      setupGUI();
   }
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      g.setColor(Color.red);
      g.setFont(new Font("Serif", Font.BOLD +
      Font.ITALIC, 20));
      g.drawString("Hello World",70,100);
      g.setColor(new Color(200,100,200));
      g.drawRect(50,50,100,100);
   }
}
```

Your output should resemble Figure 13-6.

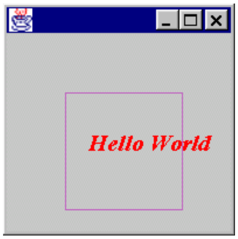


Figure 13-6: Output of PaintingStuff (with fonts)

Exercise 13-1: Drawing to your Scribble *JFrame*

In this exercise, you will draw to the surface of your `JFrame` by overriding the appropriate method. Interestingly, you are drawing to the surface of the `JFrame`, not the `JPanel`. How do you see the result if the `JPanel` is on top of the `JFrame`? In `Swing`, components are transparent by default. In effect, you are looking through the `JPanel` to see the surface of the `JFrame`, as shown in Figure 13-7.

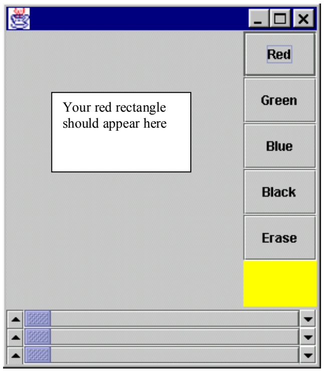


Figure 13-7: Your JFrame

1. Draw a filled red rectangle in the top center of the `JFrame` by overriding the appropriate method to obtain a reference to the graphics context.
2. (*Optional*) Add 3-D borders to your red rectangle by using the `Graphics` method of `draw3DRect(int x1,int y1,int width, int height, boolean raised)`. Try to achieve the visual effect of a raised button.
3. *(Optional)* Adjust the fonts on the `JButtons` to make them aesthetically appealing.

SUN CERTIFICATION

The methods of the `Graphics` class that appear on the exam are: `drawString()`, `drawLine()`, `drawRect()`, `drawImage()` `drawPolygon()`, `drawArc()`, `fillRect()`, `fillPolygon()`, and `fillArc()`. The `drawRect()`, `fillRect()`, `drawImage()`, and `drawArc()` methods all need the rectangle in which the graphical element will be drawn. These methods require the *x* and *y* coordinates of the upper-left corner of the rectangle and the width and height of the rectangle. The coordinate system used in Java was discussed in this chapter, and all coordinates are given in pixels. The `fillPolygon()` and `drawPolygon()` methods require a set of *x* and *y* coordinates for each point of the polygon, again using pixels in the same coordinate system.

SUMMARY

This chapter covered some of the techniques used to obtain the graphics context of a component, as well as some of the methods of the `Graphics` class. You also learned about the `Color` and `Font` classes and that the `Cursor` class works similarly to `Color` and `Font`. The ideas discussed in this chapter create the visual aspects of a Java program.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. Where is the origin in the coordinate system used in Java?

2. What would output would this piece of code produce?

```
import javax.swing.*;
import java.awt.*;

class PaintStuff extends JFrame
{
                  private Container c;
                  PaintStuff()
                  {
                  setupGUI();
                  drawSomething();
                  }

                  private void drawSomething( )
                  {
                  Graphics g = getGraphics();
                  g.setColor( Color.blue );
                  for( int i = 0; i < 75; i++ )
                  {
                  g.drawRect( i, i, i * 2, i * 2 );
                  }
                  }
```

```
         private void setupGUI()
         {
         c = getContentPane();
         c.setLayout( new BorderLayout() );
         setSize( 200, 200 );
         setVisible( true );
         }
}
```

3. What attributes would the following code give to a font?

```
g.SetFont( new Font( “SansSerif”, Font.ITALIC, 20 );
```

4. Why would you want to use this code in a program that used graphics?

```
public static void main( String args[] )
{
         GraphicApp app = new GraphicApp();
         app.addWindowListener( new WindowAdapter()
         {
         public void windowClosing( WindowEvent e )
         {
         System.exit( 0 );
         }
         } );
}
```

CHAPTER 14

The Delegation Model

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Explain the event delegation model conceptually.
- Create listener classes that can respond to events.
- Register listener classes with their Component sources.
- Capture events and deal with them in meaningful ways.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is an event?

   ....

   ....

2. What kinds of user actions can generate events?

   ....

   ....

INTRODUCTION

When creating a program that runs within a windowing environment, one of the basic necessities is event handling. Dealing with things like mouse and button clicks, and window activation and deactivation is essential to the creation of a graphical user interface. A program in this type of environment must constantly listen for events to occur, and, when they do, process them appropriately. The process through which this is done in Java is outlined in the AWT event model and is referred to as the Delegation Model. In this chapter, you will learn the basics of the Delegation Model including what an event is, and and how to implement it within a program.

The basic idea behind event handling is that there are event objects, event sources and event listeners. Each event listener is told for which type of event object to listen and when a source creates and broadcasts an event object of that type, the event listener performs its specified task.

WHAT IS AN EVENT?

An event can be described as something that happened or occurred. This definition, though basic, is appropriate. To describe an occurrence relative to Java, ask the following questions:

- What caused the event? Was it a JButton, a JTextField, or a keystroke?
- What was the exact nature of the event? Was a JButton pressed or was it released? Was a mouse clicked once, or was it double-clicked?
- Is additional information available about the event? If a mouse button was clicked, where was it clicked (the coordinates)? If a key on the keyboard was pressed, was it in conjunction with the CTRL key or ALT key?

Based on these questions, you can surmise that when an event occurs in Java, it creates an event object. This event object contains all the information you need to properly process the event:

- The source of the event (which Component generated it).
- The ID of the event (e.g., a mouse down or mouse up).
- Other important information based on the type of event (such as the label of the JButton that generated the event).

SDK 1.3 EVENT HANDLING

The JDK 1.1 model was an advance in elegance, scalability and efficiency over the 1.0 method, and remains unchanged in SDK 1.3. The JDK 1.1/SDK 1.3 model is based on a concept called the event delegation model.

> *TECH NOTE:*
>
> *Although the enhanced event delegation model was introduced in JDK 1.1, this book will refer to it as belonging to the SDK 1.3.*

The event delegation model relies on the concept of event sources (objects that generate events—often `Components`) and event listeners (objects that receive `Events`). Some questions that arise in relation to the event delegation model are:

- How does the event source generate the Event object?
- How does the event source know where to send the Event object?
- How does a listener prepare to receive the Event?
- What does the listener do once the Event is received?

The following sections will address these questions systematically.

Generating the event object

An Event object is generated automatically by actions such as a user moving the mouse or clicking a button. Although you have the ability to generate your own Events, you need not generate Events for standard AWT `components` (although this is done frequently with JavaBeans).

One disadvantage of the 1.0 model was that the Event object generated was too general (only one type of Event object was generated). The 1.3 model takes a different approach. The 1.3 model generates many different types of `Events` depending on the source. For example, `Windows` generate `WindowEvents`, and `JCheckboxMenuItems` generate `ItemEvents`. Some less obvious examples are `JButtons` and `JTextFields` generating `ActionEvents`.

Once you know which sources generate which `Events`, the question remains as to how the source knows where to send the `Event` object so it can get the proper response?

Sending the event object to the listener

The `Event` object that the source object creates must be sent to a listener object. How does it know which listener to send the `Event` object to? What if more than one listener is available? The source must register the listener to which it wants to send the `Event` object. This arrangement is common and can be demonstrated with ordinary events.

If you (the listener) want to receive a subscription to *Time* magazine (the `Event` object) from the publisher (the event source), what would you do? You would ask the publisher to put you on its mailing list (register yourself with the source). Then, when the publisher releases the next issue of *Time* magazine, you automatically receive a copy. Java handles its process the same way.

The difference in this analogy is the level of strictness with which Java adheres to this relationship. The publisher of *Time* magazine will send the magazine to anyone who pays. Java is not so liberal. `Event` sources can send `Event` objects only to listeners that have been specifically designed to receive those `Events`. Therefore, when a listener object is designed, it must be prepared to receive `Event` objects of a specific type.

Preparing the listener to receive the event

The class designed to receive the Event object must adhere to a specific contract. The contract is established by implementing the proper listener interface. Recall that an interface is a contract between the interface, the class which implements that interface, and the methods that the class must implement. An object can only accept event objects of the type it listens for. The listener interfaces available to the programmer in SDK 1.3 are:

- ActionListener
- ComponentListener
- FocusListener
- ListSelectionListener
- MouseListener
- MouseMotionListener
- WindowListener

Three basic steps are involved in the implementation. First, create an object which implements one of the above event listeners:

```
public class Hello extends JApplet implements ActionListener {
```

Next, within that object, create a method named actionPerformed() that contains the code that is to be executed whenever the listener hears something.

```
public void actionPerformed(ActionEvent e)
{
    Toolkit.getDefaultToolkit().beep();
}
```

Then create one or more source objects that broadcast events to specified listeners. Tell it what the specified listeners are via the appropriate add...Listener( ) method. For example, if you added a JButton (which broadcasts an Action event) within the Hello applet above and to register Hello with the button as a listener, you would do the following:

```
JButton beepButton = new JButton(“Click Me”);
beepButton.addActionListener(this);
...
```

Here the this keyword was used since the button was within the same object as the actionPerformed method. To add an ActionListener belonging to a separate object, simply pass that object in the addActionListener’s parameter list.

You may register more than one actionListener with an event source. If you have an object named beepCounter which also implemented the ActionListener interface, and you want to add it to the beepButton’s list of registered listeners, you could use the following code:

```
JButton beepButton = new JButton(“Click Me”);
beepButton.addActionListener(this);
beepButton.addActionListener( beepCounter );
```

Consider what you have learned for the next example.

Example: Creating a closeable JFrame

You may have noticed that when you create a JFrame, it does not close down the application but rather the JFrame itself. You will now remedy this situation.

What is the source of the event?

If you are trying to close a JFrame that is a type of Window, then your source is a Window object.

What type of event object is generated?

Consult Figure 14-2 at the end of this chapter or any good API. According to this diagram, Window objects create events of type WindowEvent.

What object types are capable of receiving WindowEvents?

According to Figure 14-2, objects that implement the `WindowListener` interface are able to receive `WindowEvent` objects.

Now you have all the information needed to design the listener class.

```
import java.awt.*;
import java.awt.Window.*;
import java.awt.event.*;
import javax.swing.*;

public class MyWindowListener implements WindowListener
{
    public void windowOpened(WindowEvent e) {}

    public void windowClosing(WindowEvent e)
    {
        Window w = (Window)e.getSource();// Step 1
        w.setVisible(false);// Step 2
        w.dispose();  // Step 3
        System.exit(0);// Step 4
    }

    public void windowClosed(WindowEvent e) {}

    public void windowIconified(WindowEvent e) {}

    public void windowDeiconified(WindowEvent e) {}

    public void windowActivated(WindowEvent e) {}

    public void windowDeactivated(WindowEvent e) {}
}
```

Note the following items concerning the preceding code:

- The implementation of the interface forces you to implement all the methods of that interface, even if they do nothing.
- The four steps listed above provide an accepted way to close your window and application. We will examine each step.

Step 1 – `Window w = (Window)e.getSource()`:

One piece of information contained in every `Event` object is the source of the `Event` (the `JFrame` object reference itself). Because a `JFrame` is a `Window` (by inheritance), you must cast the variable `w` as such. The `WindowEvent` type also contains a method called `getWindow()`, which returns a variable of type `Window` (requiring no casting).

Step 2 – `setVisible(false)`:

The current `JFrame` is visible. This step visually removes the `JFrame` from the screen.

Step 3 – `dispose()`:

Although the `JFrame` is no longer visually displayed, it is still a reference within the JVM. This step removes the reference from the JVM.

Step 4 – `System.exit(0)`:

This step shuts down the JVM. Be sure this outcome is the one you want.

Although you have created a listener that will properly respond to a `WindowEvent`, you still need to register this class with the source.

Now create a simple `JFrame` called `MyCloseableJFrame` so you can close it with your listener class. In the following code, notice the use of the `public static void main(String[] args)` to instantiate itself. This technique is common.

```
import java.awt.*;
import java.awt.Window.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class MyCloseableJFrame extends JFrame
{
    MyCloseableJFrame ()
    {
        setSize(200,200);
        show();
    }

    public static void main(String[] args)
    {
        MyCloseableJFrame myCF = new MyCloseableJFrame ();
    }
}
```

Now that you have a simple `JFrame` to display, you must register the event listener class to the `MyCloseableJFrame` class (the source). Notice the method used to register the listener. This convention is used each time you add a listener. If you were to register a listener class that was designed to handle events of type `ItemEvent`, the class would have to implement the `ItemListener` interface. The method to actually register the listener (in the source) would be called `addItemListener()`.

```
import java.awt.*;
import java.awt.Window.*;
import java.awt.event.*;
import javax.swing.*;

public class MyCloseableJFrame extends JFrame
{
    MyWindowListener myWindowListener;
    MyCloseableJFrame ()
    {
         setupGUI();
    }
    public static void main(String[] args)
    {
        MyCloseableJFrame myCF = new MyCloseableJFrame ();
    }
    private void setupGUI()
    {
        myWindowListener = new MyWindowListener();
        addWindowListener(myWindowListener);
        setSize(200,200);
        show();
    }
}
```

Summary of the event handling process

1. Determine the source of your `Event`. Is it a `Window` or a `Button`?
2. Determine the type of event the source `Component` will generate. If the source is a `Button`, then an API will tell you that it generates an `ActionEvent`. Note the type of listener the `addSomeListener()` requires. In the case of a `Button`, the API will tell you that `Button` has a method called `addActionListener()`, which means that it generates an `ActionEvent`.
3. Determine the type of listener class that must be created. Again, in the case of a `Button`, it requires a listener of type `ActionListener` because it generates an `ActionEvent`.
4. Create a listener class that implements the appropriate listener. Implement all methods of the interface. Make the necessary method(s) truly functional.
5. Register an instance of the listener class from the source using the appropriate `addSomeListener()` method.

JFrame convenience methods for event handling

You might have noticed that the JFrames you created were able to close by default (although not to the point of shutting down the JVM). This default feature of JFrames is new to Swing.

To set default closing operations on a JFrame without implementing a listener class, use the JFrame method of setDefaultCloseOperation(). The three modes are demonstrated as follows:

```
class CloseJFrameStuff extends JFrame
{
    CloseJFrameStuff()
    {
        setupGUI();
    }
    private void setupGUI()
    {
    // You would not set all three modes
    // at the same time in your own code.

    // 1. Hide and remove the JFrame reference from memory.
        this.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    // 2. Do not hide or close at all.
        this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

    // 3. Simply hide the JFrame. This is the default mode.
        this.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);

        setSize(200,200);
        show();
    }
}
```

Example: Event handling and callbacks

To further the development of the event model, consider another scenario. Suppose you need to modify something in the source class (or interface) from the listener class. This situation is common, and it relates directly to a concept discussed earlier regarding callbacks. For the listener class to modify any of the `Components` from the source, the listener class needs a reference to the `Component`.

In this example, you have two `JButtons` and a `JTextArea`. Ultimately, a message will pop up in the `JTextArea` when a `JButton` is clicked. Start with the basic framework. The interface is shown in Figure 14-1. When a user clicks a button, you want a message to be displayed in the `JTextArea`. You must pass a reference for this `JTextArea` to the `EventListener` class.

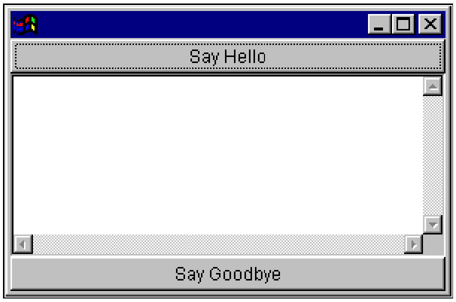


Figure 14-1: Interface with JButtons and JTextArea

```
import java.awt.*;
import java.awt.Window.*;
import java.awt.event.*;
import javax.swing.*;


// This is the Frame with the JButton and JTextField.
// As shown here, it will only display a message
// to the command line when the JButton is clicked.
// You will put both classes on the same page.

public class MyCloseableJFrame extends JFrame
{
                      JButton top        = new JButton("Say Hello");
                      JButton bottom     = new JButton("Say Goodbye");
                      JTextArea textArea = new JTextArea(15,30);
                      MyJButtonListener myJButtonListener;

                      MyCloseableJFrame ()
                      {

                      setupGUI();
                      }
                      public static void main(String[] args)
                      {
                      MyCloseableJFrame myCJF = new MyCloseableJFrame
  ();
                      }
                      private void setupGUI()
                      {
```

```
               Container c = getContentPane();
               c.add(BorderLayout.NORTH, top);
               add(BorderLayout.SOUTH, bottom);
               add(BorderLayout.CENTER, textArea);
               myJButtonListener = new MyJButtonListener();
               top.addActionListener(myJButtonListener);
               bottom.addActionListener(myJButtonListener);
               setSize(300,300);
               setVisible(true);
               }
    }
    class MyJButtonListener implements ActionListener
    {
               public void actionPerformed(ActionEvent e)
               {
               // Notice how you can distinguish one button
     from
               // another by examining the JButton label.
               if(e.getActionCommand().equals("Say Hello"))
               {
               System.out.println("Hello");
               }
               else
               if(e.getActionCommand().equals("Say Goodbye"))
               {
               System.out.println("Goodbye");
               }
               }
    }
```

Although this program responds to your JButtons, it does not yet meet your objective, which is to display the messages back in the JTextArea. How can you obtain a reference to the JTextArea from the listener object? The key is that you pass a reference to the listener object via its constructor.

In the original code, the ActionListener was added as follows:

```
myJButtonListener = new MyJButtonListener();
top.addActionListener(myJButtonListener);
bottom.addActionListener(myJButtonListener);
```

Now that you want to pass a reference to the MyJButtonListener, you can modify the preceding code as follows:

```
myJButtonListener = new MyJButtonListener(textArea);
top.addActionListener(myJButtonListener);
bottom.addActionListener(myJButtonListener);
```

Of course, by passing a reference of type JTextArea to MyJButtonListener, you need an appropriate constructor to receive textArea. The new and complete MyJButtonListener is:

```
import java.awt.*;
import java.awt.Window.*;
import java.awt.event.*;
import javax.swing.*;

class MyJButtonListener implements ActionListener
{
                      JTextArea textArea;
                      MyJButtonListener(JTextArea tmpTextArea)
                      {
                      textArea = tmpTextArea;
                      }
                      public void actionPerformed(ActionEvent e)
                      {
                      // Notice how you can distinguish one button
  from
                      // another by examining the JButton label.
                      if(e.getActionCommand().equals("Say Hello"))
                      {
                      textArea.append("Hello \n");
                      }
                      else
                      if(e.getActionCommand().equals("Say Goodbye"))
                      {
                      textArea.append("Goodbye \n");
                      }
                      }
}
```

Although this process seems to handle additional work, the code can grow in size and become unmanageable when you need callbacks to several components in your GUI. In this case, it may be better to simply pass a reference to the GUI itself (e.g., the `MyJCloseableFrame` class) and access the various `Components` in that manner. For example:

```
    myJButtonListener = new MyJButtonListener(this);
```

Now the `MyJButtonListener` class has a reference to the entire `MyCloseableJFrame` class.

You will learn that the SDK 1.2 model is very flexible and offers multiple options for handling each situation. In the next chapter, you will examine one of these options.

SUN CERTIFICATION

This chapter covered how to use the Java 1.1 event delegation model. You learned about listeners and how to register them with the components that will generate the event. One of the most difficult aspects of the Sun test is that the resources you normally rely on while writing code, such as the Java 2 API documentation that comes with your environment, are not available. Therefore, you should memorize the events that each component might generate and the methods of each of the applicable listeners. This task involves a great deal of work, but it is necessary to pass the exam.

As you used the various listener interfaces, you had to create many stubs. Often, there are more stubs than coded methods for the interfaces. In the `java.awt.event` package, a series of adapter classes is available. These adapters must be subclassed (they are a series of empty methods), but if used, the stubs are not necessary. One drawback to using a subclass of an adapter (or creating any listener method of your own) is that another class must be sent over the Internet if you are using an applet. The design is up to the author, but knowing how to use the adapter classes and listener interfaces are skills that will be tested on the exam.

SUMMARY

This chapter covered events and Java's models for event handling. You studied the event delegation model and through its processes. You also learned that if an event listener is used by several different classes, then it should exist as a separate class. In the next chapter, you will learn how to include the event listener directly into the class where it is used, when it is used in only one class.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. Does a JMenuItem implement ItemListener?

2. What are four important things to consider when dealing with events?

CHAPTER 15

Inner Classes

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Define an inner class.
- Explain the advantages inner classes have over package-level classes in relation to event handling.
- Design and implement inner classes for event handling.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is an Inner class?

   ....

   ....

INTRODUCTION

Inner classes are primarily the result of blending the concepts of block structured programming with those of classes. Inner classes facilitate an efficient way to organize classes within classes and provide mechanisms to access classes as member variables. In this section, we will delve into the specifics of inner classes, look at different flavors of inner classes, and finally analyze a few examples.

WHAT IS AN INNER CLASS?

As their name suggests, inner classes are classes defined within other classes. This feature was not available with the JDK 1.0; it was introduced with the SDK 1.2. Inner classes are one of the more significant changes to the SDK model.

Inner classes offer some conceptual advantages over separate classes. For example, you might be designing a `Body` class (such as a human body). A body contains organs such as the heart and brain. Inner classes allow you to visualize this idea easily. For example:

```
class Body
{
    class Heart
    {
        // Code
        // Inner classes allow you to conceptualize your constructs
        class Brain    // in meaningful (and highly encapsulated ways)
        {
            // Code
        }
}
```

There are four different types of inner classes:

- static inner classes
- member inner classes
- local inner classes
- anonymous inner classes

A static inner class has similar qualities to a static method. It has access to all static methods and variables of the current class and that of the parent classes. Like most static references, it does not require an instance of the enclosing class to exist for the static inner class to be constructed. A member inner class is a generalization of a static inner class and is similar to a member of a class. Like any other member, it has access to all methods and variables inside of the class. Local inner classes are classes defined within a local code block and are visible only within that block. Local inner classes only have access to their own variables and parameters and any `final` variables of that method. Finally, anonymous inner classes are those inner classes that are defined without any names. Anonymous inner classes are usually created for small tasks such as event handlers.

Given below is a simple example of a local inner classes:

```
public class A
{
    int compute(final int x)
    {
        int y;
        class B
        {
            public int compute()
            {
                return x * 5;
            }
        }
    }
}
```

The above example illustrates a class, `B`, that is a local inner class. Note that the variable `x` is accessible to all methods in class `B` since it is declared `final` ; although, variable `y` is not accessible inside of class `B`, because it is not declared to be `final`. Also, class `B` and its own members would not be directly accessible to other methods of class `A`.

The standard access protection mechanisms applicable to variables and methods, apply to member inner classes, i.e., non-local inner classes can be defined `public, private,` or `protected` just as member variables. Similarly, member inner classes can be defined as `final` or `abstract`. Thus, any class that inherits from an inner class would need to be further defined. We show an example later of how a class can extend an inner class. However, a local inner class cannot be declared `public`, `private`, or `protected` as it is not a member variable.

On a similar note, inner classes cannot declare static variables. This is because of the conflicting ideologies of inner classes and the `static` keyword. Static variables and methods are designed to be used by all instances of a particular class and always refers to a top-level class and never an enclosed class.

Using inner classes has many advantages, including good organizational structure, easy conceptual abstraction, and localization of relevant code. However, disadvantages to inner classes also exist, such as the increased number of classes, the increased number of interactions between classes, and the inability to use them outside the scope of their enclosing class. Using inner classes correctly requires careful design, but when used appropriately, they can prove very powerful.

INNER CLASSES FOR EVENT HANDLING

Event handling is an example of a process that can be streamlined by inner classes. Refer back to your example where clicking a JButton caused a message to display back in the JTextArea. To accomplish this, it was necessary to pass a reference to the JTextArea through the constructor. However, this method of attack can make your code become overly complicated as your needs grow. Inner classes can solve this problem.

Because inner classes can be defined within a normal class, they can have the same scope as any other method or variable defined at that scope. Therefore, in the example where your Button was sending a message to the JTextArea, if the MyJButtonListener class had been defined at the member level, it would have automatically had a reference to the JTextArea.

The modified code is as follows:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Window.*;
import javax.swing.*;

public class MyCloseableJFrame extends JFrame
{
    JButton top        = new JButton("Say Hello");
    JButton bottom     = new JButton("Say Goodbye");
    JTextArea textArea = new JTextArea(15,30);
    MyJButtonListener myJButtonListener;

    MyCloseableJFrame()
    {
        setupGUI();
    }
    public static void main(String[] args)
    {
        MyCloseableJFrame myCF = new MyCloseableJFrame();
    }

    private void setupGUI()
    {
        Container c = getContentPane();
        c.add(BorderLayout.NORTH, top);
        c.add(BorderLayout.SOUTH, bottom);
        c.add(BorderLayout.CENTER, textArea);
        myJButtonListener = new MyJButtonListener();
        top.addActionListener(myJButtonListener);
        bottom.addActionListener(myJButtonListener);
        setSize(300,300);
        show();
    }
    // Inner class defined
    class MyJButtonListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            if(e.getActionCommand().equals("Say Hello"))
            {
                textArea.append("Hello \n");
            }
            else if(e.getActionCommand().equals("Say Goodbye"))
            {
                textArea.append("Goodbye \n");
            }
        }
    } // End inner class
} // End MyCloseableJFrame
```

It is important to note the immediate access that the inner class has to any of the instance variables of the outer class (the MyCloseableJFrame class).

The preceding example demonstrated the use of a member level inner class. The use of member level inner classes is very similar to the use of a seperate class. The difference between the two are the means of referencing the inner class, and the scope of the inner class. The following is an example of an anonymous inner class. This class is unusual in two ways: first, it is defined entirely within the opening and closing parentheses of a method and second, it has no name associated with it.

```
class AnonymousInnerClassStuff extends JFrame
{
    JButton btn = new JButton("A JButton");
    AnonymousInnerClassStuff()
    {
        setupGUI();
    }
    private void setupGUI()
    {
        btn.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                // Code to respond to JButton being clicked...
            }
        } ); // End () and anonymous inner class.
    }
}
```

Anonymous inner classes provide an easy way for a programmer to implement small classes without having to generate meaningless names, but their use can make code very difficult to read. They should generally only be used in situations where the behavior of the class is well-defined and the class is fairly small, no more than two or three methods. Since they have no name, they may have no explicit constructor.

How do inner classes represent themselves? This question may actually be answered in two ways, how the class can be referenced in the code, and how the class appears on the disk. Inside the code, an inner class may be referenced in a form similar to OuterClass.InnerClass. Obviously, anonymous inner classes may not be referenced in this fashion, since they have no name. The name of an inner class on the disk is similar. If you were to find the compiled source for your `MyCloseableJFrame.class`, you would now find another file named `MyCloseableJFrame$MyButtonListener.class`. This file is the inner class.

A logical question that arises based on the above description is that: Can inner classed be extended? Consider the following code fragment:

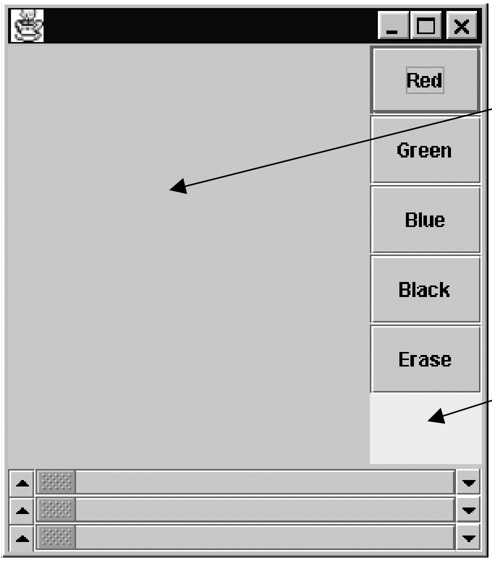
```
class A
{
    class B
    {
        int x;
    }
    B myB_1 = new B();
}

class C extends A.B
{
    C()
    {
        new test(). new A(). super()
    }
}
```

The preceding example shows how a member inner class, `B`, can be instantiated within the class `A`. The code fragment also shows how another class, `C`, can extend the inner class, `B`, and that class C requires an instance of its enclosing class and the class that it extends. If class `B` were abstract, an object of class `B` could not be instantiated, but an object of class `C` could be with the idea that class `C` would have further defined class `B`.

GRADUATING TASK #4: EVENT-ENABLING THE SCRIBBLE APPLICATION

In this task, you will event-enable the `Scribble` application to fully implement all its features. Although, you have learned several forms of event handling, you might find that using member inner classes is the best approach. Figure 15-1 shows the application and the `JPanels` with which you will be working.



This large *JPanel* is your "canvas," the surface upon which you will draw.

This small *JPanel* is your "palette." It always displays the current drawing color.

Figure 15-1: Scribble application

1. Event-enable the red, green, blue, and black `JButtons` so that when they are clicked, the `palette` will change to the appropriate color.
2. Event-enable the three `JScrollBars` so that as they are dragged, a new `Color` object is constantly generated. The `palette` should always reflect the color represented by the state of the three `JScrollBars`.

3. Event-enable your mouse so that when you press and hold the button, you will draw to the surface of the large `JPanel` (the `canvas`) in the current color of your `palette`.

HINT: *You must implement two interfaces to trap the "mouse-pressed" and "mouse-dragged" events.*

4. Event-enable the `Erase JButton` so that when it is clicked, the `palette` changes to the current background color of the drawing canvas. You can use your mouse to "erase" any previous drawings with this color.
5. (*Optional*) Modify Step 1 so that when the red, green, blue, or black `JButton` is clicked, the `JScrollBars` automatically adjust to represent the proper RGB value.

SUN CERTIFICATION

Expect to see inner classes on the Sun certification examination. Because you will often create an inner class that is used only once, you need not assign it a name. This is referred to as an anonymous inner class. This example was used earlier in this chapter to show an inner class:

```
import java.awt.*;
import java.awt.event.*;
import java.awt.Window.*;
import javax.swing.*;

public class MyJFrame extends JFrame
{
                      JButton top        = new JButton("Say Hello");
                      JButton bottom     = new JButton("Say Goodbye");
                      JTextArea textArea = new JTextArea(15,30);
```

```
// We will create an inner class of type ActionListener
// and because we will only use it in this class, we will
// make the class anonymous, even though
// MyJButtonListener is an instance of the class.
                    ActionListener myJButtonListener = new
ActionListener()
                    {
                    public void actionPerformed(ActionEvent e)
                       {
                       if(e.getActionCommand().equals("Say Hello"))
                       {
                          textArea.append("Hello \n");
                       }
                          else if(e.getActionCommand().equals("Say
Goodbye"))
                       {
                             textArea.append("Goodbye \n");
                       }
                       }
                    };

                    MyJFrame()
                    {
                    Container c = getContentPane();
                    c.setLayout(new BorderLayout());
                    c.add(BorderLayout.NORTH, top);
                    c.add(BorderLayout.SOUTH, bottom);
                    c.add(BorderLayout.CENTER, textArea);
                    setSize(300,300);
                    top.addActionListener(myJButtonListener);
                    bottom.addActionListener(myJButtonListener);
                    setVisible(true);
                    }
```

```
                      public static void main(String[] args)
                      {
                      MyJFrame myframe = new MyJFrame();
                      }

} // End MyJFrame
```

SUMMARY

An inner class is a class defined within another class. Inner classes can prove to be a very convenient tool for organizing code. However, managing scope within inner classes can be confusing. The following rules should aid in determining what is within scope of the inner class and what access modifiers are permitted for the inner class.

- In a member level inner class, the class has access to all variables declared within the enclosing class. The inner class may be declared public, protected, or private.
- In a local inner class the only variables within scope are the final variables of the enclosing method. A local inner class may not be declared with an access modifier, and can be treated as a private member of that local block.

Another confusing issue is the use of the static keyword with inner classes. The presence or absence of the static keyword dictates whether a reference to the enclosing class is necessary to access the inner class. In general you should remember:

- Local inner classes may not be declared static.
- An inner class that is declared static need not have a reference to its enclosing class. Any non-static inner class must have a reference to its enclosing class, though this reference may be an implicit this.new.
- A static inner class only has access to static members of the enclosing class.

Anonymous inner classes provide a handy way to implement trivial adapter classes and other similar constructs. The following three points summarize the concept of the anonymous inner class:

- An anonymous inner class is declared and instantiated at the same moment.
- An anonymous inner class is unnamed and therefore cannot have an explicit constructor. Since it is no named, it may be referenced outside of the area in which it is constructed.
- An anonymous inner class may extend another class or implement an innerface by calling the superclass in its declaration.

With the previous points in mind, anonymous inner classes should begin to make sense. They have a variety of potential uses, but in practice they are primarily used for simple interfaces and should not be used when the class is long or complicated.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What do you use inner classes for?

CHAPTER 16

Applets

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Compare and contrast applets and applications.
- Implement the life cycle of an applet through its inherited methods.
- Embed an applet into a HTML document.
- Pass parameters from a HTML document to its contained applet.
- Identify applet security restrictions.
- Convert an applet into an application.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is an Applet?

2. Can you convert an applet into an application?

INTRODUCTION

At the beginning of this course, you learned that Java programmers write either applications, servlets, or applets. Thus far, the focus has been on applications. However, now that Components, Containers, and event handling have been covered, applets can now be programmed. Programming applets poses only minor conceptual differences from programming applications.

Examine the AWT class hierarchy shown in Figure 16-1. Observe that Applet is the parent class for all applets and it extends Panel. The Applet class adds significant functionality to the Panel class. The difference is that Applets are designed to run inside an applet viewer, typically a Web browser.

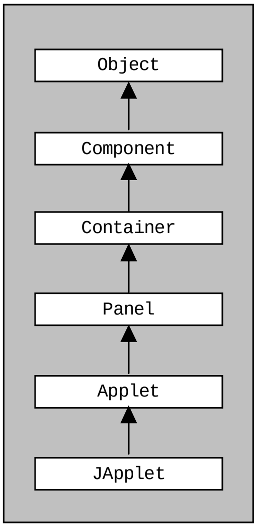


Figure 16-1: Applet class hierarchy

APPLETS AND WEB BROWSERS

The relationship between applets and Web browsers is unique. Foreign code, the applet, is trying to execute inside a browser that is executing on an operating system. This is a three level system. The levels are complicated by the fact that the Java Virtual Machine, the browser, and the operating system could all be distributed by different manufacturers. For these reasons, strict rules are needed to ensure proper communication between the levels.

- Rule No. 1: An applet's init() method must be called before it will execute.
- Rule No. 2: An applets life cycle is controlled by the following methods:

```
public void init()
public void start()
public void stop()
public void destroy()
```

- Rule No. 3: Applets can receive information from HTML pages.
- Rule No. 4: The browser controls all security for the applet.

TECH TIP:

Consider compatibility when you use applets. Not all browsers support the latest versions of Java. Sun JavaSoft offers a plug-in to make your Web browser Java 2-compliant. Go to www.javasoft.com/products/plugin/ to download the plug-in.

JApplets

JApplet is an extended class of java.applet.Applet. It adds support for interposing input and painting behavior in front of the applet's children. Additionally, it adds support for special children that are managed by a LayeredPane and for the Swing MenuBars. JApplet implements the Accessibile and RootPaneContainer interfaces. JApplet usage is similar to that of Applet, but there are some key differences.

- Components do not go directly in the frame. Instead they go in the content pane.
- The default layout manager is not FlowLayout as it is in Applet; it is BorderLayout like Frame and JFrame.

- Instead of getting the native look it defaults to the Java Metal look.
- Drawing is done in paintComponent() instead of paint().
- Double buffering is used by default.

Applet life cycle

The init() method is the first method executed in an Applet. Both the main() and init() methods are responsible for initialization of their code. The init() method is called only once.

The start() method is the second method called in an applet. The start() method might be called repeatedly throughout the life of the applet. It is called whenever the applet gains focus. This commonly occurs when the browser is minimized and then restored or when the user moves to another Web page and then comes back. This functionality is required when processes are running in the applet that consume many resources. These resources need to be released so that other programs can execute.

The stop() method works in conjunction with the start() method. The stop() method is responsible for releasing resources consumed by the applet. A good example is an applet that is running animation. Animation consumes large amounts of memory and processor cycles. If the user navigates to another Web page, the applet should be respectful of the user's resources and stop() the animation. These resources will be regained and the animation will resume when the applet regains focus, the browser will call start().

TECH TIP:

> *Not every applet will need the start() and stop() methods to be declared. The methods are inherited from Applet and are called automatically by the browser. It is not recommended that the methods be overridden except in process-intensive applets.*

The destroy() method is called when the applet is removed from memory. Since this is a process of Garbage Collection, there is no way of knowing exactly when the method will be called. However, it should be used to release any system resources that may not have been returned to the system by stop(). Such resources include database connections, threads, and open streams.

The paint() method should be considered one of the five essential methods of applets. The paint() method is inherited from Container. Typically, any Graphics drawn to the applet's graphics context will be performed by overriding the paint() method.

The paint() method is the only method involved with the life of the applet that the programmer has control over. Various actions that occur behind the scenes, under the control of the browser and the AWT, actually control the applet.

The <APPLET></APPLET> tags

Often applets are started from within a HTML page. The <APPLET> tag is used to inform a browser that an applet is embedded. The <APPLET> tag has three required parameters: CODE, WIDTH, and HEIGHT. A Web page example follows:

```
<HTML>
<BODY>
    <APPLET
    CODE="MyJApplet.class"
    WIDTH=300 HEIGHT=200>
    </APPLET>
</BODY>
</HTML>
```

This HTML code will execute an applet code named MyJApplet.class. The browser will assume that this code resides in the current working directory. The size of the applet is 300 x 200 pixels.

TECH NOTE:

HTML 4.0 favors the use the <OBJECT> tags instead of <APPLET>. Keep in mind, however, that the Appletviewer utility included with the Java 2 SDK does not support the <OBJECT> tag.

Along with the required parameters, several optional attributes can be used within the <APPLET> tags, including CODEBASE, ALIGN, and ALT.

CODEBASE specifies a relative directory path to locate the executable code. This is often used when images and class files are located in different directories.

ALIGN allows the applet to be positioned inside the browser window. The values that can be assigned to ALIGN are LEFT, RIGHT, BOTTOM, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, ABSBOTTOM, VSPACE, and HSPACE. Refer to a HTML manual for the specifics of each value.

ALT allows a message to be displayed to a user who has disabled Java in his or her browser.

The following code illustrates all of these attributes.

```
<HTML>
<BODY>
<APPLET
    CODEBASE = "Applet/Classes"
    CODE ="MyApplet.class"
    WIDTH="300"
    HEIGHT="200"
    ALIGN="LEFT"
    ALT="Your Java feature has been disabled">
</APPLET>
</BODY>
</HTML>
```

Passing parameters to applets

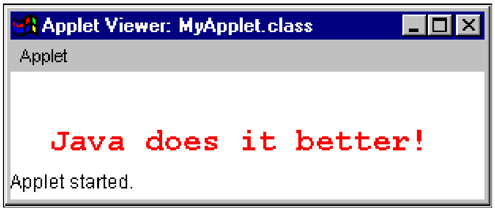
HTML pages can send information to an embedded applet by using parameter tags. Examine the following example:

```
<HTML>
<BODY>
<APPLET
CODE="MyJApplet.class"
WIDTH=300 HEIGHT=200>
<param name = "fontSize" value = "20">
<param name = "fontColor" value = "red">
</APPLET>
</BODY>
</HTML>
```

Although the nature of the information being passed can be anything, the Applet method used to retrieve the value, getParameter(), only returns a String. Therefore, the parameter value may have to be converted into the needed data type. The following applet will receive the parameters listed in the HTML from above. Notice the conversion from Strings to other data types.

```
public class MyJApplet extends JApplet
{
    int fontSize;
    Color fontColor;
    String tmpFontColor;
    public void init()
    {
        fontSize = new Integer(getParameter("fontSize")).intValue();
        tmpFontColor = getParameter("fontColor");
        if(tmpFontColor.equals("red"))
            fontColor = Color.red;
        else if(tmpFontColor.equals("green"))
            fontColor = Color.green;
        else
            fontColor = Color.blue;
        repaint();
    }
    public void paint(Graphics g)
    {
        g.setColor(fontColor);
        g.setFont(new Font("Courier",Font.BOLD, fontSize));
        g.drawString("Java does it better!", 25,100);
    }
}
```

The output should resemble Figure 16-2.



```
Java does it better!
```

Applet started.

Figure 16-2: Output of MyApplet

TECH TIP:

HTML is not case sensitive. Java is case sensitive. The value of a parameter will be accepted exactly as it is typed in the HTML page.

Applets should not be trusted

Applets have strict security limitations placed on them; these limitations present advantages and drawbacks.

Applet security is an advantage because an untrusted piece of code can be prevented from running on a computer where it does not have security permissions. The freedom to perform malicious acts such as deleting, copying, or corrupting data can be denied. The security permissions for applets are set in the browser of the users computer. On the other hand, applet security is a drawback because many common tasks cannot be performed by an applet.

Applets are said to live in a sandbox, that imposes the following security restrictions:

1. Applets cannot run any executable code on the local machine.
2. Applets can neither read nor write disk information from the local machine.
3. Applets can establish a network connection only with the host from which they were downloaded.
4. All windows that pop up from an applet have a warning message that alerts users not to enter sensitive information.

It is important to realize that these security restrictions are not part of the Java language, but are imposed on the applet via browser's SecurityManager. Each browser has a unique manner for setting security levels.

The Appletviewer supplied by the Sun SDK offers the same level of access for an applet as for an application. The Sun HotJava browser is also somewhat liberal, when compared to a Netscape or Microsoft browser.

Since the introduction of the JDK 1.1, applets can be signed. A signed applet contains a piece of data similar to a digital signature. Theoretically, if a signed applet is downloaded from a trusted source, the user can give it more access rights than an unsigned applet. This allows an applet to perform many tasks that were previously forbidden.

Table 16-1 describes many of the methods of a JApplet.

Table 16-1: Methods of the Applet class

Method	Description
AppletContext getAppletContext()	Returns an Applet Context. This object provides several methods useful for interacting with the browser.
String getAppletInfo()	Some applets override this method to display useful information to the user of the applet.
AudioClip getAudioClip()	Returns an AudioClip object at a specified URL.
URL getCodeBase()	Returns the URL that launched the applet.
URL getDocumentBase()	Returns the URL of the HTML page that launched the applet.
Image getImage()	Returns an Image at a specified URL.
String[] getParameterInfo()	Some applets display their parameter information as stored in an array.
Boolean isActive()	Returns true or false depending on whether the applet is started or stopped.
Void play()	Plays the AudioClip returned by getAudioClip().
Void resize()	Resizes the applet.
Void showStatus()	Displays a message in the status window of a browser (browser permitting).

CONVERTING AN APPLICATION INTO AN APPLET

An application can be converted into an applet and an applet into an application. The following rules describe the process of converting an application.

1. The init() method of an applet is similar to the main() method in an application. Use it to perform any form of initialization or instantiation.
2. The Web browser will instantiate the applet.
3. Applications are derived from Containers, applets must be derived from Applet or JApplet.

4. The size and visibility of the JFrame in an application is set within the application via setSize() and setVisible() methods. The size of an applet should be set within the HTML document, while the Web browser makes it visible.
5. Applets must be declared public.

The following code is a simple GUI application, that will be turned into an applet.

```
import java.awt.*;
import java.awt.Window.*;
import java.awt.event.*;
import javax.swing.*;

class MyApplication extends JFrame
{
    JButton north = new JButton("North");
    JButton south = new JButton("South");
    JButton east =  new JButton("East");
    JButton west =  new JButton("West");
    JTextArea textArea = new JTextArea();
    MyApplication()
    {
        setUpGUI();
    }
    public static void main(String[] args)
    {
        MyApplication m = new MyApplication();
    }
    void setUpGUI()
    {
        Container c = getContentPane();
        c.setLayout(new BorderLayout());
        c.add(BorderLayout.NORTH, north);
        c.add(BorderLayout.SOUTH, south);
        c.add(BorderLayout.EAST, east);
        c.add(BorderLayout.WEST, west);
       c.add(BorderLayout.CENTER"Center", new JScrollPane(textArea));
        setSize(200,200);
        setVisible(true);
    }
}
```

The output should resemble Figure 16-3.

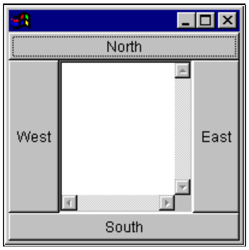


Figure 16-3: Output of MyApplication

The following code is the same program written in Applet form:

```
/* HTML code necessary to start the applet

<HTML>
<BODY>
<APPLET
    CODE ="MyApplet.class"
    WIDTH="200" HEIGHT="200">
</APPLET>
</BODY>
</HTML>

*/

public class MyApplet extends JApplet
{
    JButton north = new JButton("North");
    JButton south = new JButton("South");
    JButton east =  new JButton("East");
    JButton west =  new JButton("West");
    JTextArea textArea = new JTextArea();
    public void init()
    {
        setUpGUI();
    }
    void setUpGUI()
    {
         Container c = getContentPane();
         c.setLayout(new BorderLayout());
         c.add(BorderLayout.NORTH, north);
         c.add(BorderLayout.SOUTH, south);
         c.add(BorderLayout.EAST, east);
         c.add(BorderLayout.WEST, west);
    c.add(BorderLayout.CENTER"Center", new   JScrollPane(textArea));
    }
}
```

The output should resemble Figure 16-4.

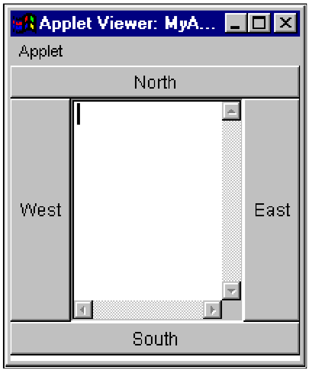


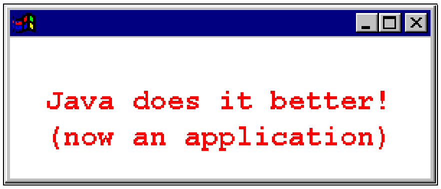
Figure 16-4: Output of MyApplet

CONVERTING AN APPLET INTO AN APPLICATION

Applet extends Panel, therefore an applet can be added to a JFame. This is accomplished by adding a main() method and a JFrame to the MyApplet class. Inside the main() method, create an instance of the MyApplet and add it to the JFrame. Then, call the init() method on the applet. Because the applet is no longer receiving parameters from an HTML page, the applet's variables must be set in another manner.

```
public class MyApplet extends JApplet
{
    int fontSize = 20;
    String tmpFontColor = "red";
    Color fontColor;
    public static void main(String args[])
    {
        JFrame f = new JFrame();
        MyApplet myApplet = new MyApplet();
        f.getContentPane().add(myApplet);
        myApplet.init();
        myApplet.start();
        f.setSize(300,125);
        f.setVisible(true);
    }
    public void init()
    {
        if(tmpFontColor.equals("red"))
            fontColor = Color.red;
        else if(tmpFontColor.equals("green"))
            fontColor = Color.green;
        else
            fontColor = Color.blue;
        repaint();
    }
    public void paint(Graphics g)
    {
        g.setColor(fontColor);
        g.setFont(new Font("Courier",Font.BOLD,
                            fontSize));
        g.drawString("Java does it better!", 25,50);
        g.drawString("(now an application)",25,75);
    }
}
```

The output should resemble Figure 16-5.



Java does it better!
(now an application)

Figure 16-5: Output of MyApplet after conversion

Exercise 16-1: Converting the Scribble application into an applet

The differences between applets and applications are subtle. Other than the security restrictions placed on an applet by its environment, application and applet development are virtually identical. In this exercise, you will convert your Scribble application into an applet. If you do not have a browser that supports the Swing components, you can either download the plug-in from Sun JavaSoft at http://www.javasoft.com/products/plugin/ or use the Appletviewer that is included with the SDK 1.2.

1. Convert your Scribble application into an applet. Keep the following points in mind:
   - You must extend JApplet instead of JFrame.
   - Although applets support a default constructor, you will probably want to substitute the role of the main() method in your application with the init() method of your applet.
   - Applets should not be able to close the browser in which they are contained.
2. *(Optional)* Instead of having your Scribble applet exist within the browser, retain it as an application. Launch it from another applet in the browser as a JFrame. It is possible to have free-floating windows from within a browser.

SUN CERTIFICATION

Java 1.1 added the ability to use JAR files with applets. By doing this, an applet gained the ability to load all of its associated files at one time. Sun's exam will possibly ask quesitons concerning the use of JAR files with applets.

Applet is a subclass of Panel which is a subclass of Container which is a subclass of Component which is a subclass of Object. Applet inherits every method of all of these super classes. Sun tests on these methods and their proper use. Applets and applications can be converted. Questions will be asked about the conversion process and what methods are allowed.

SUMMARY

This chapter discussed the differences between Java applets and applications. You learned about the applet life cycle and some of its inherited methods. You also considered the security restrictions placed on applets, as well as some alternatives to those limitations. Finally, you converted an application into an applet and an applet into an application. Although the development of applets is an important part of the Java language, this course focuses on applications. In following chapters, the focus will be on more advanced topics, such as exceptions, threads, streams, and networking.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. How do you pass an integer named number with a value of 4 to an applet?

   ....

   ....

2. What are the applet security restrictions?

   ....

   ....

CHAPTER 17

Exceptions

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Differentiate between Errors and Exceptions.
- Differentiate between RuntimeExceptions and explicit Exceptions.
- Handle an Exception using the try/catch statement.
- Propagate an Exception using the throws statement.
- Create and use a user-defined Exception.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is an Exception?

   ....

   ....

2. What is an Error? What is the difference between an Error and an Exception?

   ....

   ....

INTRODUCTION

Abnormal conditions may occur inside of code for a number of reasons, ranging from unexpected input, to undefined program behavior. The ability to handle and recover from these situations is called exception handling. Proper use of exception handling increases program robustness and can insure program correctness. Java has a cleanly designed exception handling mechanism built natively into the language. The use of these exception handling features is required in order for many methods to execute.

WHAT IS AN EXCEPTION?

An abnormal condition in the program execution may cause unexpected behavior. In Java, in many cases it is possible to detect and recover from this condition. In most cases where this occurs the program has thrown an exception. An `Exception` is an abnormal condition that has arisen. The Java design states that an `Exception` is a subclass of the `Throwable` class.

Java defines a number of exceptions that may arise from various common causes. These exceptions may range from examples such as a user having entered incorrect input, to the program attempting to access an invalid area of memory, to a program attempting to access a file which does not exist on disk.

A programmer may designate certain areas of the code as being subject to an exception being thrown, and guard against abnormal behavior by catching this exception. When an exception is thrown in this manner, it maintains some information on what type of situation it resulted from, and what type of exception it is. The programmer may use this information inside of a `try/catch/finally` clause in order to deal with the exception, or he may choose to ignore the exception by not dealing with it and continue program execution.

Multiple different types of exceptions may also be dealt with in separate ways, or the programmer may attempt to deal with all `Exception`, by catching all exceptions which fall underneath the super class `Exception`, which is all of them.

In this way, a programmer may protect against all but the most serious types of program failure. For these types of failures, it is normally impossible for a program to recover and generally, there should be no reason for a program to deal with a serious failure of this type, known as an `Error`.

Errors

`Error` is the second subclass of `Throwable`. `Errors` can occur at runtime and usually are of a serious enough nature that they should not be handled by the `Exception` handling mechanism. `Errors` will terminate a program and are generally functions over which the programmer has little control, such as `OutOfMemoryError` or `StackOverflowError` or the dreaded `UnknownError`.

Exceptions

The superclass of all exceptions is `Exception`. A number of subclasses of `Exception` exist, but they can be effectively divided into two types, all exceptions which are subclasses of `RuntimeException` and all other exceptions which are not subclasses of `RuntimeException`.

RuntimeExceptions (unchecked)

`RuntimeException` is an abstract class defining a set of possible conditions that might occur anywhere in a program. Some examples of `RuntimeExceptions` are:

- ArithmeticException (often the result of dividing by 0)
- ClassCastException (trying to perform an illegal cast operation)
- IndexArrayOutOfBoundsException (e.g., accessing element 15 of a 10-element array)
- NullPointerException (trying to access an object when the variable contains null)

The Java compiler does not enforce programmatic handling of `RunTimeExceptions` (you can ignore them), which is why they are referred to as unchecked exceptions. This relaxation of the common exception handling rule is because a runtime exception can feasibly occur at any point in the code, but the occurrence can normally be avoided through proper and careful programming. Although, they may be easily avoided most of the time, runtime exceptions may still be treated as normal exception

Other exceptions (checked)

The remaining exceptions can be found in multiple Java packages. Some of them are defined in `java.lang` (such as an `InterruptedException` for a sleeping `Thread`). Others are defined in `java.net` or `java.io` (such as a network connection not found or a file not available). You can even create your own `exceptions`, then use the same handling mechanisms discussed in this chapter with your custom exceptions. You can accomplish this by creating a class that is derived from `Exception` and using a default constructor. Another term for these exceptions is checked `Exceptions`. This is where you must state that they might occur and you must make provisions for them via a `try/catch/finally` block(you cannot ignore them).

Figure 17-1 illustrates how the various exceptions are related to one another.

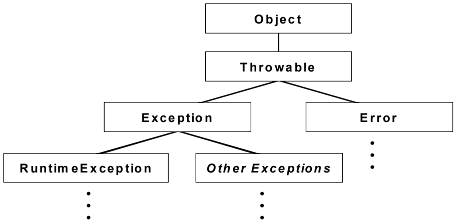


Figure 17-1: Exception class hierarchy

WHEN BAD THINGS HAPPEN TO GOOD PROGRAMS

Exceptions will occur in your programming. Better programming practices can help you avoid `runtime exceptions`, and you cannot do much to prevent `errors`. Therefore, we will focus on explicit `Exceptions` (referred to as *Other Exceptions* in Figure 17-1).

There are four possible actions you can take if an `Exception` occurs:

1. Ignore the `Exception`.
2. Handle the `Exception` with a `try/catch` statement.
3. Throw the `Exception` to the calling method.
4. Handle the `Exception` and rethrow it to the calling method.

Ignoring the Exception

In the case of a `RuntimeException`, you can simply ignore the occurrence. If no action is taken when an `Exception` occurs, the method in which it occurs immediately stops. It then reports back to the method that called it, stating that an `Exception` occurred. If this method ignores the reporting method, the reporting method continues to propagate up the calling methods until the program finally terminates. This propagation should be avoided by careful programming.

For explicit `Exceptions`, this scenario cannot occur. If a method might generate an `Exception`, it is explicitly stated by a `throws` declaration. For example, the method specification for `clone()` in the `Object` class is as follows:

```
    protected native Object clone() throws CloneNotSupportedException;
```

Once this declaration exists, you cannot ignore it, so you must take other action.

Handling the Exception

This action is most often taken with `exceptions`. The code that might generate an `exception` (usually a call to a method with a `throws` statement) is written into a `try/catch` statement as follows:

```
try
{
     myObject.clone();
}
catch (CloneNotSupportedException e)
{
    // action to take if the Exception occurs
    System.err.println("This clone wasn't supported!");
}
```

The `try` block implies that the program is going to attempt to execute the code contained within the block, but recognizes that an exception may be thrown from a method inside of that block. The catch block is required and will handle any exception of the type declared in the catch clause. In this case, it is a `CloneNotSupportedException`. A finally clause may also exist.

The goal is to catch an `Exception` and correct it so it does not interfere with the smooth flow of your program. A potential `Exception` might occur in the `try` statement. If an `Exception` is thrown, control is immediately transferred to the `catch` statement to handle the `Exception` in an appropriate manner.

An extension to the `try/catch` block is the `finally` clause. The finally clause is guaranteed to always be executed after the `try` and associated `catch` blocks are finished executing. Following is an example of the use of a `finally` clause.

```
try
{
    //some code that may throw an exception
}
catch(Exception e)
{
    // "fix" any problems
}
finally
{
    // clean up - for example, close any IO streams.
}
```

It is possible to catch different types of `Exceptions`. For example:

```
try
{
    //some stuff that throws an Exception
}
catch(IOException e1)
{
    //fix problems
}
catch(InterruptedException e2)
{
    // fix problems
}
catch(Exception e3)
{
    // fix problems
}
```

In the previous code, you tested first for IOException. If that was not the exception which occurred, then an InterruptedException is tested for. If that is not the correct type of the exception, then the Exception clause will catch it because all exceptions ultimately inherit from Exception. Notice the order of the catch statements. Note that if an IOException were caught by this example, the first catch block would handle the exception and execution would resume after the last catch block. In this example, if there is no Exception, the return statement will return control to the calling method. However, the finally clause is always invoked, even if no exception was raised.

Throwing the Exception to the calling method

Sometimes, you want to defer the handling of an exception. For example, create a program that handles all exceptions in methodA(), and notifies the programmer that they occurred.

```
void methodA()
{
    try
    {
        methodB();
    }
    catch (Exception e)
    {
        System.out.println("something went wrong");
    }
}

void methodB() throws Exception
{
    methodC();
}
```

```
void methodC() throws Exception
{
    throw new Exception()
}
```

In this program, `methodA()` calls `methodB()`, which calls `methodC()`. Then `methodC()` declares that it `throws an Exception`, so the `Exception` is passed up to `methodB()`. Now `methodB()` appears to have thrown the `Exception`, and passes it up to `methodA()`. The `methodA()` contains the `Exception` handling code (`try/catch`), so the processing occurs here. This process shows how explicit `Exceptions` can be passed up the calling stack.

Handling and rethrowing the Exception

It is possible to `try/catch` an `Exception` and then rethrow it. The syntax is as follows:

```
public someMethod() throws SomeException
{
    try
    {
        // code that may throw SomeException
    }
    catch(SomeException e)
    {
        // some Exception handling code
        throw e;
    }
}
```

This way, some level of `exception` handling may take place, and then the exception passed back up the calling stack for other methods to perform further `exception` handling.

CREATING AND THROWING YOUR OWN EXCEPTIONS

Creating the Exception

Creating exceptions in Java is straightforward. Typically, you will want to create a checked `exception`. You can create a checked `exception` by subclassing the `Exception` class.

```
public class MyException extends Exception
{
    // You now have your own exception.
    // There is a little more to do still...
}
```

Each `Exception` generates a unique message when it is thrown. The `Exception` you build should generate a message in case it is not caught. The constructors defined in the `Exception` class are designed to make a message available when the `Exception` occurs. However, constructors are not inherited, so you must use an explicit call to `super()` constructor from within `MyException` to access this feature.

```
public class MyException extends Exception
{
    public MyException(String message)
    {
        // Let the Exception class do all the work.
        super(message); // a call to super must be the first thing
                         // in a constructor
    }
    public MyException()
    {
        // It is possible to set a default message.
        super("A MyException was thrown.");
     }
}
```

Throwing the Exception

Throwing an `Exception` in Java is straightforward. It also provides much insight into how certain methods used in this course were defined. For example, you have used the `Thread.sleep()` method several times, yet we have never explained why it is necessary to place this method in a `try/catch` block. The following code, which throws a `MyException`, would be nearly identical to the `Thread.sleep()` method, which throws an `InterruptedException`.

```
public class ThrowMyException
{
    // In this method, if help is set to true (and it
    // is), a MyException is thrown.
    // Notice that the method itself is designed to throw
    // the MyException with the "throws" keyword.
    public void throwMyExceptionMethod() throws MyException
    {
        boolean help = true;
        if(help)
        {
            // If the conditions are right, throw an instance
            // of the MyException class.
            throw new MyException("Help! Help!");
        }
    }
}
```

Now, the system is set. The MyException class is well-defined, and the code within the method that throws the MyException is also well-defined. Now, test the system:

```
public class TestMyException
{
    public static void main(String[] args)
    {
        ThrowMyException throwME = new ThrowMyException();
        // It is a checked exception, so the method must be
        // placed inside of a try/catch (for example).
        try
        {
            throwME.throwMyExceptionMethod();
            {
                catch(MyException e)
                {
                    System.out.println(e);
                }
            }
         }
     }
}
```

Upon running TestMyException, you will see the following on your command line:

```
test2.MyException: Help! Help!
```

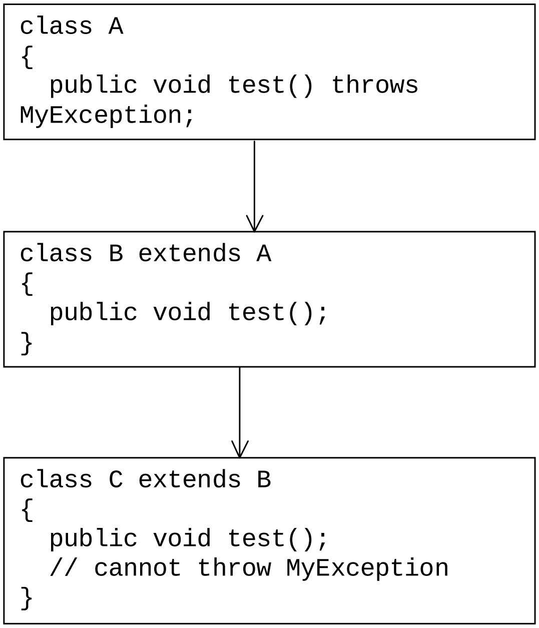
EXCEPTION HANDLING TIPS

1. When deciding on whether a method should throw an exception, you should consider whether this exception must be handled by the external method. If all appropriate actions could be taken from within the internal method, the one which you intend to throw the exception, then it is best to handle exception there.
2. While it is often easier to declare a catch clause to catch all exceptions of type `Exception`, it is better practice to declare each individual type of exception that the try block may throw. This practice allows for easier code maintenance and modification.
3. Remember that after all methods inside of a `catch` block have been executed, flow of control will return to the `finally` block if it exists, or to the end of the `try/catch` statements if `finally` does not exist.
4. Subclasses of `RuntimeException` need not be caught, but they can be caught if the programmer desired. Thus an attempt may be made to check an `ArrayOutOFBoundsException` or a `NullPointerException`.
5. Remember that all code that appears in a `try` block after a method that may generate an exception cannot be guaranteed to execute.

SUN CERTIFICATION

On the Sun certification exam, exceptions are divided into checked and unchecked `Exceptions`. Unchecked `Exceptions` are the same as `RuntimeExceptions`, which were discussed in this chapter. Unchecked `Exceptions` (like `Errors`) do not have to be thrown or caught. Checked `Exceptions` can either be included in a `try/catch` block or declared to be thrown by the method.

It is interesting to see which exceptions can be thrown when a method is overridden in a subclass. An overridden method cannot throw any checked `Exceptions` that are not thrown by the method overridden. Figure 17-2 demonstrates this.



```
class A
{
  public void test() throws
MyException;
```

```
class B extends A
{
  public void test();
}
```

```
class C extends B
{
  public void test();
  // cannot throw MyException
}
```

Figure 17-2: Overridden method

SUMMARY

This chapter defined `Exceptions` and `Errors`, and discussed how to deal with Java's `exception` handling capabilities. You will learn more about `Exceptions` in the upcoming chapters. `Thread` methods, IO methods, and networking methods are all prone to `Exceptions` being generated, so they must often use the `try/catch` or `exception` propagating mechanisms discussed in this chapter.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What will be printed in the code below?

```
double d = 5.0;
try{ System.out.print(d/0); }
catch(ArithmeticException ae){System.out.println(“ERROR”); }
finally{ System.out.println(“ TEST”);
```

2. What does the throw(JRE)~~s~~(JRE) keyword do?

3. What are the four things you can do with an Exception?

CHAPTER 18

Creating Threads and Thread Methods

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Define threads.
- Create and instantiate threads using two different techniques.
- Control single-thread flow using many thread methods.
- Define the four thread states and their relationships to thread methods.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is a thread, and what is the significance of proper thread use?

   ....

   ....

2. In Java, what are the two possible ways one would go about creating a new thread of execution?

   ....

   ....

3. What are the conditions that might prevent a thread from running?

   ....

   ....

4. What is the significance of the of the synchronized keyword?

   ....

   ....

INTRODUCTION

Thus far, all of your programs have been written to executed sequentially: all of the statements execute one after the other, in a specified order. Multi-processing and parallelism allow multiple actions to be executed simultaneously, either physically simultaneous on multiple CPUs, or logically simultaneously on a single CPU. This chapter explores the Java support for simultaneous processing using threads

WHAT ARE THREADS?

Before defining a thread, there must first be a distinction made among the terms multitasking, multiprocessing, and multithreading.

Multitasking is a general term referring to the ability to perform more than one function simultaneously.

Multiprocessing is the ability to run more than process at the same time. If you are running a word-processing program and a graphics editor and also listening to your favorite CD on your computer, you are running three processes at once. Most modern operating systems are efficient at multiprocessing.

Multithreading is the ability to excite more than one action within the same process. Suppose that you are running a Web browser (a process), and you are using it to send an e-mail, download an audio file, and upload a text file to an FTP server. You are running three threads within one process. The primary difference between multiprocessing and multithreading is that each process maintains a separate data set, whereas multiple threads can work on the same data set.

As a result of the integral multithreading support designed into the language, Java applications and applets are implicitly multi-threaded. The `Object` class contains methods `wait()` and `notify()` which are used in thread control. Multithreading is normally more efficient than non-threaded programs or multiprocessing. Multithreading may prevent parts of a process from waiting while a slower part executes. It is also less taxing for an operating system to maintain multiple threads than to maintain multiple processes. This enables programmers to write very efficient Java code.

HOW OPERATING SYSTEMS HANDLE MULTITASKING

One of the difficulties faced by Java language designers was the various approaches used by different operating systems to address multi-tasking. Fundamentally, multitasking involves two techniques: pre-emptive multitasking and cooperative multitasking.

Pre-emptive multitasking allows the operating system to interrupt the execution of one process or thread, and transfer control to another process or thread. Each process notifies the OS of its priority. The OS uses this information to continuously provide each active process with its slice, or allotment, of CPU time. In such a system, the tasks need not cooperate with each other because a task can receive CPU time based on its priority. Thirty-two bit Microsoft Windows operating systems use pre-emptive multitasking.

Cooperative multitasking requires that a task willingly surrender control in order for task can gain CPU time. The problem becomes apparent: What if a task is unwilling to relinquish control? The possibility for dominating the CPU is great.

Java uses pre-emptive multitasking. Although Java may be running on a system that uses cooperative multitasking, this discrepancy does not pose a problem as long as you design your threads to behave in a considerate manner. The JVM handles the thread scheduling inside of the Java process.

TYPES OF THREADS IN JAVA

Java uses two types of threads: daemon threads and user threads. Most daemon threads are created by the JVM, while most user threads are created by the programmer. However, user threads may be set as daemon threads.

The difference between the two threads is based on the services they perform. Daemon threads are usually designed to run in the background for the purpose of servicing user threads. The garbage collector thread is a good example of a daemon thread. It works in the background performing its garbage collection service for other threads. When all other user threads have finished, the garbage collector thread no longer exists.

The programmer will rarely work directly with daemon threads. The JVM provides other servicing threads in addition to the garbage collector. There is a daemon thread for processing mouse events, and for maintaining your graphics context. If you need some type of scheduling service running in the background, you could create a daemon thread for this purpose. Daemon threads expire when the last user thread expires. At that point, the program will terminate.

Programmers create and utilize user threads more frequently than daemon threads. You have worked with one particular user thread since your first Java program: the main thread.

The main thread is a user thread made available by the JVM. This thread is launched in the `public static void main(String[] args)` method. From this main thread, you will launch all other threads. See Figure 18-1.

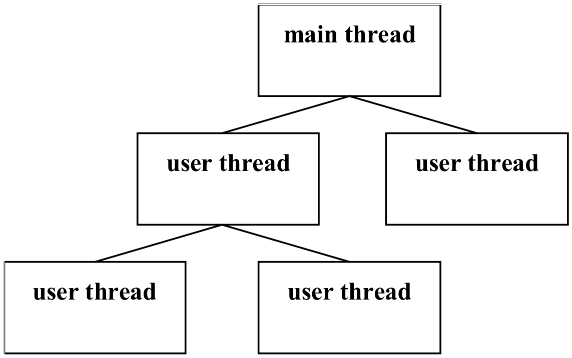


Figure 18-1: The main thread

CREATING THREADS

In Java, a thread may be created using two techniques. You may either subclass the Thread class, or create a class which implements the Runnable interface. Both techniques are used extensively in Java, and each has advantages and disadvantages. The next section will examine these techniques in detail and discuss their strengths and weaknesses.

Subclassing the Thread class

Java has a class called Thread. By extending the Thread class, you inherit all the functionality of this class.

```
class MyThread extends Thread
{
}
```

This code is a valid Thread that does nothing. You can remedy this by adding a method named run().

The JVM expects to find all code intended for execution as part of a Thread in the run() method. By consulting the Runnable interface the JVM knows that the run() method is available.

The Thread class implements the Runnable interface. The only method defined in this interface is run(). When you extend Thread, you also inherit this run() method. You must override the run() method to implement the threaded code.

To start your thread, you cannot simply call the run() method of the Thread instance. You must call the start() method of the Thread instance; the start() method will invoke the run() method. As stated previously, threading varies extensively among operating systems. When the programmer calls start() and the JVM calls run(), you should be reminded that much must be accomplished behind the scenes before your threads can run.

The following code demonstrates the power of threads. You will run two infinite loops in a "before threads" and "after threads" demonstration.

```
class ThreadStuff
{
    public static void main(String[] args)
    {
        MyThread myThread = new MyThread();
        myThread.start();

        while(true)
        {
            System.out.println("Main Thread");
        }
    }
}

class MyThread
{
    // Before we implement threading
    public void start()
    {
        run();
    }
    public void run()
    {
        while(true)
        {
            System.out.println("MyThread Thread");
        }
    }
}
```

The output of this code is as follows:

```
MyThread Thread
MyThread Thread
MyThread Thread


.
.
```

Because this program is not threaded, you cannot display the message in both `while` loops. Once you call `start()` in `MyThread`, it enters into an infinite loop, and the main thread is blocked out, so it will never execute.

Programmers constantly encounter instances when two threads must begin running at the same time. The solution to this problem is simple. By modifying the `MyThread` class, you can remedy this example problem and write your first explicitly multi-threaded application.

```
class MyThread extends Thread
{
    // After we implement threading
    // the MyThread class inherits the start() method from the
    // Thread class.
    public void run()
    {
        while(true)
        {
            System.out.println("MyThread Thread");
        }
    }
}
```

Here, the MyThread class inherits from the Java Thread class, and therefore inherits all of the behavior of the Thread class. The run() method has been overridden to provide the functionality that we desire, so that when the Start() method is called, the JVM creates another thread in the program and switches between the execution of that thread and the main thread. One possible output is the following, but this may vary each time it is run:

```
MyThread Thread
Main Thread
MyThread Thread
Main Thread
MyThread Thread
Main Thread
.
.
.
```

Implementing the Runnable interface

The previous example of multithreading by extending Thread was straightforward and functional. This technique, however, has two serious drawbacks. The first is that Java does not support multiple inheritance. Once you extend Thread, you have locked your class into being a subclass of Thread, and you forfeit a degree of flexibility in your class design.

The second drawback is that, in terms of design, the Thread class may not be an appropriate superclass for your class. You wanted to gain the functionality of the Thread class, yet by inheriting from Thread you are forcing yourself to assume a type that does not support your class structure.

The use of the `run()` method to write code has been discussed previously. The `Thread` class implements the `Runnable` interface, and by contract must implement this method. You can implement the `Runnable` interface for your class. See the following rewrite:

```
class MyThread implements Runnable
{
    // Rewrite using Runnable instead of Thread
    public void run()
    {
        while(true)
        {
            System.out.println("MyThread Thread");
        }
    }
}
```

Two words have been changed; this threaded class has now been rewritten to use the second method of writing threaded classes. Now you must start your `Thread`.

Consider your `Main` class again; you instantiated the `Thread` and then started it by calling its `start()` method as follows:

```
MyThread thread = new MyThread();
thread.start();
```

You cannot solve this problem with your modified `thread` because there is no `start()` method. Remember that you are no longer inheriting from `Thread`, therefore you do not have a `start()` method.

You can solve this problem by using the Thread class constructor that takes an Object of type Runnable. The delegation design pattern is very common in object-oriented design. In essence, a front-end object delegates the responsibility of the work to some other implementation object. You can now start your Runnable MyThread:

```
class ThreadStuff
{
    public static void main(String[] args)
    {
        Thread myThread = new Thread(new MyThread());
        myThread.start();

        while(true)
        {
            System.out.println("Main Thread");
        }
    }
}
```

Which technique?

The choice of technique deserves careful consideration.

Object-oriented purists would argue that one should never subclass another class without the intention of further defining it. This point counters the subclassing technique, because you do not add to the base class Thread.

The Thread class is such an inherent part of the Java language that it is almost a reusable component, and is one of the easier elements of Java to understand. These points favor the subclassing technique.

The interface technique is more difficult to understand. However, once understood, it proves much more flexible, because it allows the class to inherit from another class. This feature, combined with its adherence to the rules of subclassing, and its relationship to the delegation design principle, makes it an elegant and recommended technique.

A thread can be compared to a talented violinist: Playing solo is one skill, but playing in an orchestra is completely different. So far, your `MyThread` has been playing solo; you must now learn ways of orchestrating your threads.

A note should be made concerning three methods of the `Thread` class: `stop()`, `suspend()` and `resume()`. These three methods have been deprecated with the release of Java SDK 1.2. The `stop()` and `suspend()` methods were prone to leaving a thread in an inconsistent state; the `resume()` method is the counterpart to the `suspend()` method. In the next lesson, you will learn some alternative techniques to replace these methods.

You can orchestrate your threads with the many thread-controlling methods available in Java. Table 18-1 lists the methods you will use in this chapter to help control a single thread. In the next chapter, you will learn ways of controlling many threads simultaneously.

Table 18-1: Methods used to control a single thread

Method	Description
currentThread()	Returns an instance of the currently running thread.
getName	Returns the name assigned to this thread.
notify()	Notifies the single thread that initially called `wait()` on the object.
notifyAll()	Notifies all threads that called `wait`() on the same object.
* resume()	Resumes the execution of this thread (*deprecated).
run()	Executes the body of the thread.
setName()	Sets the name of the thread.
setPriority()	Sets the priority of the thread.
sleep()	Causes the currently running thread to sleep for a specified amount of time (in milliseconds).
* stop()	Kills the thread (*deprecated).
Start()	Calls the thread's `run()` method to begin executing the thread.
* suspend()	Suspends execution of this thread until resumed (*deprecated).
yield()	Causes the currently running thread to yield to other threads.
wait()	Dictates that the calling thread give up the object monitor and sleep until it is notified by a later thread.

Thread states

Before you use any of the preceding thread-controlling methods, it helps to know the various thread states and how these methods affect a thread's state.

The Java specification defines four thread states: new, runnable, not runnable, and dead.

New thread

A thread is in the new state after it is instantiated, but before it is started. The following statement will place a thread in the new state.

```
MyThread myThread = new MyThread();
```

Runnable

A thread is runnable once its `start()` method has been called. It may seem odd to define the state as runnable instead of running; however, this term is appropriate: a thread spends much of its life on a thread queue waiting to be run. The following statement places a thread in the runnable state.

```
MyThread.start();
```

Not runnable

Certain methods will remove a runnable thread from the active queue. The thread is then not runnable. At this point, it cannot be placed back on the active queue until specifically requested. In the next section, you will learn techniques for placing a thread in the not runnable state. In essence, you will simulate the now-deprecated `suspend()` method of the `Thread` class.

Dead

A thread is dead once it has been explicitly stopped. You will simulate the now-deprecated `stop()` method in the next section. A thread is also dead once its `run()` method exits normally. In Java, dead is a permanent state; the thread cannot be resurrected.

Figure 18-2 illustrates the thread states.

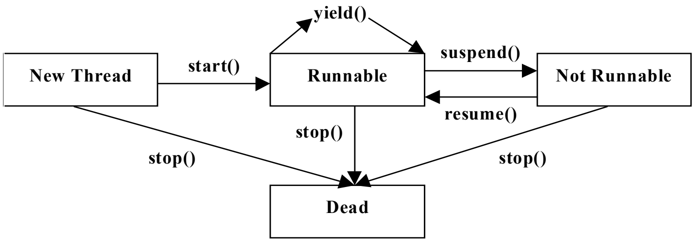


Figure 18-2: Thread states

The currentThread(), getName() and sleep() methods

To gain a better understanding of the methods of Thread, you can take a look at the most commonly used thread, the main thread.

currentThread() and getName()

The currentThread() method allows you to obtain an instance of the currently running thread, while getName() provides the String name associated with the thread.

```
class ThreadStuff
{
    public static void main(String[] args)
    {
        Thread myThread = Thread.currentThread();
        // Obtains an instance of the currently running thread
        System.out.println(myThread.getName());
        while(true);  // Pause the command screen
    }
}
```

The output of ThreadStuff is as follows:

```
main
```

In the preceding code, the only thread running is the user thread main, which is supplied by the JVM. When you ask for the current thread, you are guaranteed an instance of the main thread.

sleep()

The static method sleep() is a utilitarian method of Thread. You can specify the number of milliseconds you want the current thread to sleep as an argument. However, the call to sleep() may generate an InterruptedException, so you must invoke the method in a try/catch statement, otherwise you will get a compiler error.

In the following code, you will display the name of your main thread, and then display an additional message two seconds later.

```
class Main
{
    public static void main(String[] args)
    {
        Thread myThread = Thread.currentThread();
        System.out.println(myThread.getName());
        try
        {
            Thread.sleep(2000);
        }
        catch(InterruptedException e)
        {
            System.out.println(e);
        }

        System.out.println("2 seconds since our last message.");
        while(true);
    |}
}
```

You will use the sleep() method throughout this course.

The setName() and setPriority() methods

setName()

The method setName() will associate a String name with a Thread. This function is optional, but it may help you distinguish one thread from another when necessary.

```
MyThread myThread = new MyThread();
myThread.setName("My Thread");
```

setPriority()

The setPriority() method is an important method of Thread.

In the beginning of this chapter, we stated that Java followed a model of pre-emptive multitasking. This model requires that one thread be able to pre-empt another thread. This operation is accomplished by assigning a priority from one to 10 to any Java thread.

You need not assign a priority to a thread, since all threads automatically receive a priority of 5 upon creation, including the main thread. However, if you create many threads, and all have a priority of 5, then you need to ensure that you make your Threads behave correctly. Methods for accomplishing this will be explained later in the text.

Set a thread's priority as follows:

```
MyThread myThread = new MyThread(); // Has priority of 5
myThread.setPriority(8);   // Has priority of 8
```

Interestingly, this change in priority affects the behavior of your threaded program. To see this, refer back to your well-behaved threaded example in which the main thread and the myThread took turns displaying messages in an infinite while loop. Adjust only the priority of the myThread.

```
class ThreadStuff
{
    public static void main(String[] args)
    {
        MyThread myThread = new MyThread();
        myThread.setPriority(8);
        myThread.start();

        while(true)
        {
            System.out.println("Main Thread");
        }
    }
}
class MyThread extends Thread
{
    public void run()
    {
        while(true)
        {
            System.out.println("MyThread Thread");
        }
    }
}
```

The thread with highest priority does not necessarily claim all the CPU time. The nature of thread behavior will varies greatly from system to system. When the author tested the thread on Windows 95, a round-robin effect was produced: both threads were able to display their messages, though an almost two-to-one ratio favored the thread with the higher priority. Your operating system may behave differently. Suffice to say, a programmer cannot rely on thread priority alone to control the behavior of threads.

The yield() method

One way to force your threads to behave better is to use the yield() method. The thread that is asked to yield() must relinquish its current running cycle to be placed on the thread queue. A high-priority thread will be running again very quickly; however in the interim, it will give some CPU time to other threads. It is a common thread technique is to always yield() at some point within the body of a thread. For example:

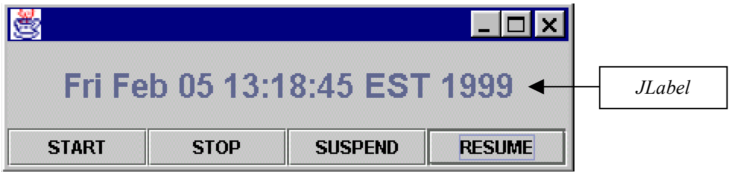
```
class MyThread extends Thread
{
    public void run()
    {
        while(true)
        {
            System.out.println("MyThread Thread");
            yield();  // Relinquish some CPU time graciously
        }
    }
}
```

The yield() method provides an excellent way to create polite threads. In reality, whenever a thread has completed a cycle of its responsibilities (for example, a mathematical calculation or part of an animation), it should release the CPU to allow processing time for other threads. Using the yield() method will facilitate this action.

GRADUATING TASK #5: CREATING A THREADED DIGITAL CLOCK

You have seen two ways to create threads, and used numerous methods of the `Thread` class. In this exercise, you will expand on the `Timer` class to make it a threaded class. You will use this thread-enabled `Timer` class to build a digital clock.

1. Create a class called `DigitalClock` that extends `JFrame` and implements `TimerInterface` (from an earlier exercise). This class will serve as the GUI for your digital clock and should resemble Figure 18-3.



Fri Feb 05 13:18:45 EST 1999

Figure 18-3: DigitalClock interface

2. Using the interface technique for creating threads, modify your `Timer` class so it is a threaded class.
3. Add a method to the `Timer` class called `public void startTimer()`. Invoking this method will start the thread (the body of the `run()` method).

HINT: *You will probably want to create an instance variable of type* Thread *in your* Timer *class called* timerThread*. This* timerThread *variable will refer to the* `run()` *method of your* Timer *class.*

4. Event-enable your interface's Start button so that when it is clicked, the `Timer` thread will start, and the time will be continuously updated on the `JLabel`. Note the following considerations:
   - You will use this `Timer` instance in several methods of your `DigitalClock` class. Therefore, declare it as an instance variable.
   - The remaining three buttons (Stop, Suspend and Resume) will be enabled in the next chapter.

5. (*Optional*) Adjust the font of the `JLabel` so that the display is large and readable. Perform any other cosmetic enhancements you deem necessary to the GUI.
6. (*Optional*) Convert your `DigitalClock` application into an applet.

SUN CERTIFICATION

The two methods of creating a `Thread` (subclassing `Thread` or implementing `Runnable`) will be tested on the exam. You should also understand the `Thread` life cycle, and know that a dead `Thread` cannot be restarted. Understand that `thread priority` should not be relied upon for deterministic programs, but only to provide a suggestion to the operating system (which it may ignore completely in some implementations). The use of the `sleep()` method is also tested on the exam.

Suspending and resuming threads, and the setting of names and daemons are not tested on the exam, but they are useful if you want to create considerate programs.

SUMMARY

This chapter covered the two ways to create a process: extending the `Thread` class and implementing `Runnable`. You also examined a few of the methods of `Thread`. In the next chapter, you will learn how to control shared resources and coordinate the action of several `Threads`.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. Explain the significance of the start(), run(), sleep(), wait(), notify(), and notifyAll() methods?

   ....

   ....

2. What is a possible output to the following code? Is this output guaranteed to always be the same?

```
public class MyThread2 extends Thread
{                        public void run()
                         {
                         System.out.println("Run Thread "+
  this.getName() +" going to sleep");
                         try { this.sleep(1000);
                         }
                         catch(InterruptedException x)
                         {
                         System.out.println("Run Thread sleep
  interrupted");
                         }
                         System.out.println("Run Thread "+
  this.getName() +" waking
                         up");
}
```

```
public static void main(String[] args)
{
                      MyThread2 t0 = new MyThread2();
                      MyThread2 t1 = new MyThread2();
                      t0.start();
                      try { Thread.sleep(1000); }
                      catch(InterruptedException x)
                      {
                      System.out.println("Thread sleep interrupted");
                      }
                      t1.start();
                      try
                      {
                      System.out.println("Main Thread going to
  sleep");
                      Thread.sleep(1000);
                      System.out.println("Main Thread waking up");
                      }
                      catch(InterruptedException x)
                      {
                      System.out.println("Main Thread sleep
  interrupted");
                      }
}
```

CHAPTER 19

Synchronization

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Define synchronization in relation to object monitors.
- Control Thread racing using Thread synchronization.
- Convert non-atomic processes to atomic processes to avoid Thread racing.
- Use sophisticated methods for controlling Threads.
- Stop, suspend and resume Threads.
- Explain Thread deadlock.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. Explain how Java uses monitors in thread synchronization.

2. True or false: Every call to a wait() method must have a corresponding call to a notify() or notifyAll() method.

3. Explain the difference between the notify() and notifyAll() methods.

INTRODUCTION

When executing a Java program, the only process being executed for that program by the operating system is the virtual machine. All activities of that program execute as threads of that one process. For this reason, Java programmers should have a firm understanding of the way Java handles multiple threads and synchronization.

WHAT IS THREAD SYNCHRONIZATION?

You have already learned how to control independent threads. In realistic systems, where more than one thread is competing for the same resource, you need more sophisticated ways to manage the threads. If you do not manage your threads in a predictable manner, you will likely produce corrupted data.

Java uses a method of controlling threads called synchronization. To appreciate the power of thread synchronization, we will introduce and duplicate the problem of thread racing. Later, through synchronization, you will solve these type of race conditions.

THREAD RACING

An excellent example of thread racing is based on banking.

Assume that a husband and a wife use an ATM system that does not incorporate synchronization.

The husband goes to the ATM to withdraw $250 from a joint checking account. The wife does the same thing in another part of town. The checking account has only $300 in it.

The husband checks to see that the account has at least $300 in it. The ATM shows that the money is there. While the husband is preparing to withdraw the money, the wife checks the same account on the ATM across town. The ATM shows the account has $300 in it, so she also decides to withdraw $250 from the account. At the exact same time they both type in the withdrawl. Since the ATM system has not synchronized the two ATM's, they both check to see that at least $250 is in the account and since it is, they perform the withdrawl. The result is that the couple has now overdrawn their account.

How can you prevent this problem? The account should have been made unavailable until the entire process of the first transaction was complete. Only after that transaction was completely finished should the second transaction have had access to the account. It would have been apparent then that the funds were insufficient to grant the second request.

The following is a pseudo-class that could have caused the flawed request. In this code, there is no way of blocking access to the withdraw method.

```
class Account
{
    public double withdraw(double amount)
    {
        if(amount <= amountAvailable)
        {
            amountAvailable -= amount;
        }
        else
        {
            amount = 0; // return nothing – an error;
        }
        return amount;
    }
}
```

If multiple threads are running, you cannot guarantee which thread is executing what part of that code at any time. In the above code, thread 1 could have executed the `if` statement, and then the scheduler may have placed thread 2 onto the processor. It may have executed the `if` statement and then continued and subtracted the amount. When thread 1 comes back onto the processor, it will continue from where it was stopped. Thus, it will execute the subtraction and the account will be overdrawn. In the previous scenario, the mishap could have been avoided by use of an object monitor that allowed single access.

SYNCHRONIZED AND THE OBJECT MONITOR

An Object that contains a monitor is able to control its methods so that only one thread has access to that object at a time.

You can associate a monitor with an Object by simply declaring a method as synchronized. For example, if you consider your Account object, you can place a monitor on that object by declaring its method synchronized. See the following:

```
class Account
{
    public synchronized double withdraw(double amount)
    {
        if(amount <= amountAvailable)
        {
            amountAvailable -= amount;
        }
        else
        {
            amount = 0; // return nothing - an error;
        }
        return amount;
    }
}
```

In this example, once a thread calls the withdraw() method, it obtains a lock from the objects's monitor. At this point, all other threads are blocked from executing this method or changing anything in this object; any that attempt to do so are placed on a thread queue. When the first thread exits the synchronized method, it releases the monitor automatically. One of the threads on the wait queue may now execute the method and obtain the monitor.

If more than one method is declared as synchronized, then only one thread may execute any of the synchronized methods at any one time. Only one monitor is present for each object, therefore every synchronized method is locked when the monitor is obtained by a thread. Even if only one synchronized method was entered by a thread, all synchronized methods will be locked until the first thread exits its method and releases the monitor.

THREAD RACE CONDITION

You will now create a race condition, then attempt to solve it. You will use the RaceThread class.

```
class RaceThread
{
    MainThread m;

    RaceThread (MainThread tmpM)
    {
        m = tmpM;
    }

    public void run()
    {
        for (int i=0; i < 10; i++)
        {
            int c = m.getCount();
            m.setCount(c+1);
        }
    }
}
```

Competing for resources

Creating a race condition is not difficult. Using your `RaceThread` from the preceding example, you need to make only minor modifications. You will add a second `RaceThread` so that two threads will compete to set the `count` value. You will also add a `pause()` method to help magnify the race condition.

```
class MainThread extends CloseableFrame
{
    // All the GUI components... and in addition ...

    private int count;
    RaceThread raceThread1;
    RaceThread raceThread2;

    MainThread()
    {
        raceThread1 = new RaceThread(this);
        raceThread2 = new RaceThread(this);
    }

    public static void main(String[] args)
    {
        MainThread mainThread = new MainThread();
        mainThread.raceThread1.start();
        mainThread.raceThread2.start();
    }
    public void setCount(int tmpCount)
    {
        pause(50);
        count = tmpCount;
        textArea.append(count + "\n");
    }
    public int getCount()
    {
        pause(50);
        return count;
    }
    public void pause(int milliseconds)
    {
        try
        {
            Thread.sleep(milliseconds);
        }
        catch(InterruptedException e)
        {
        System.err.println("MainThread: pause() " + e);
        }
    }
}
```

Threads are not guaranteed to execute in any particular order, nor are they guaranteed any amount of processing time. Therefore, the results of the program will vary from system to system, but the output will probably show a double print of the `count`, as follows:

```
1
1
2
2
3
3
.
.
```

This is not the result you want. Each thread should be updating a unique count.

Synchronizing the methods

The natural solution to your race condition is to synchronize your methods. We stated earlier that when the `synchronized` keyword is added to a method, a thread is able to get the monitor of that object, and prevent any other thread one of its `synchronized` methods.

```
public synchronized void setCount(int tmpCount)
{
    pause(50);
    count = tmpCount;
    textArea.append(count + "\n
}
public synchronized int getCount
{
    pause(50);
    return count;
}
```

After running the program again, the result is the same:

```
1
1
2
2
3
3
.
.
```

Why? The synchronization should stop the race condition. This problem is not the fault of the synchronization feature of the Java language; this problem is a design issue refered to as an atomic process.

Atomic processes

Atomic processes are processes that do not have intermediate states. In the example, the task of updating the count variable in MainThread is not an atomic process. Consider the following:

1. RaceThread1 performs a synchronized getCount(). It now exits that method for a brief moment to increment the count in its run() method.
2. RaceThread1 now performs a synchronized setCount() method to set the updated count variable.

The fact that there was a break in the complete process between the getCount() and setCount() method calls means that the process was not atomic, and that RaceThread2 was able to slip in between these method calls and corrupt the data.

Conceptually, the solution is apparent: make it one process. The following is some code that would make this an atomic process.

```
// Pseudo code for the solution
public synchronized void updateCount()
{
    int c = getCount();
    setCount(c + 1);
}
private int getCount()
{
    // return the count
}
private void setCount()
{
    // set the count
}
```

Note two points:

1. Making getCount() and setCount() private is a design choice. If you are going to increment count via the updateCount() method, do not make the methods more accessible than necessary.
2. If getCount() and setCount() are only called internally from a synchronized method, you do not need to make them synchronized as well; doing so will slow the system.

SOPHISTICATED THREAD SYNCHRONIZATION

Java provides a more sophisticated way to synchronize threads: the methods wait(), notify(), and notifyAll().

Normally, a thread is suspended and later resumed either by some external factor (such as pushing a button) or by continuously polling for the condition upon which to change the thread's state. Java has a poll-free mechanism to conditionally control threads through the method invocations of wait() and notify()/notifyAll().

The wait() and notify() methods are not methods of Thread; they are methods of Object, which proves that Java was originally designed as a multithreaded language. Since every class is a subclass of Object, every class inherits all of Object's methods. Although wait() and notify() do constitute a form of thread synchronization, they do not solve the thread race problem. In fact, the wait() and notify() methods must be called within a synchronized block themselves. For this reason, the wait() and notify() techniques are often regarded as a form of interthread communication.

An excellent example for the use of wait() and notify() is the consumer/producer model.

Consumer/producer scenario

A print queue represents a consumer/producer scenario. Although the following situation is somewhat outdated, the skills learned here can be directly applied to other consumer/producer scenarios, such as graphics.

A *primitive computer* (a producer thread) is going to print a document by placing it on a print queue one character at a time. A *primitive printer* (a consumer thread) will take the characters off the print queue and print them one at a time. The problem is that the printer can print information only when the information is available. Therefore, two possibilities must be considered. The first possibility is that no information is available to print. The second possibility is that information is available to print. In this primitive setup, we assume that the queue can only perform one operation at a time. It can allow read access or write access, but not both.

No information is available for printing

If there is no information to print, then the printer must call `wait()` on itself until the computer has placed some information in the print queue.

The `wait()` method can be called only from within a `synchronized` block. Therefore, the printer will have had control of the object's just before calling `wait()`.

Once the printer calls `wait()` on itself, it is placed in a wait queue and relinquishes the object's monitor, allowing another `Thread` to gain access to the `synchronized` block. In this case, the computer `Thread`.

The printer `Thread` will remain in this wait queue until an external `Thread` calls `notify()` or `notifyAll()`. At this point, the printer `Thread` may now have access to the print queue.

TECH NOTE:

When notify() is called, there is no way of knowing which waiting thread will execute. It only guarantees that one waiting thread will be notified. If notifyAll() is called, every waiting thread will be notified.

Information is available for printing

If information is available for printing, the printer can take that information from the print queue and print it.

Because information was available for printing, the printer must assume that the computer is waiting for the opportunity to place more information on the print queue.

The printer can notify the computer that new information is available on the print queue by calling `notify()` or `notifyAll()`. This action releases the computer from the wait queue.

These steps could have been stated from the computer's perspective with no loss of clarity. Overall, it is important to understand that the computer and printer take turns working with the print queue. When one has access to the print queue, the other is waiting in a wait queue until it is notified. This process continues back and forth.

You will now develop a more general consumer/producer system with the creation of four classes.

```
// class One – the queue
class Q
{
// This class represents the queue. A producer thread
// will place information in the queue using the putData()
// method or wait. A consumer thread will get information
// from the queue using getData() or wait.

    boolean info_in_q = false;
    int data;
```

```
synchronized public void putData(int tmpData)
{
    if(info_in_q)
    {
        try
        {
            wait();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
    data = tmpData;
    System.out.println("Put: " + data);
    info_in_q = true;
    notifyAll();
}

synchronized public int getData()
{
    if(!info_in_q)
    {
        try
        {
            wait();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
```

```
        info_in_q = false;
        notify();
        return data;
    }
}
// class Two - the producer
class DataProducer extends Thread
{
    // The DataProducer continuously places an
    // incremented integer into the Q.

    Q q;
    int data;

    DataProducer(Q tmpQ)
    {
        q = tmpQ;
        start();
    }

    public void run()
    {
        while(true)
        q.putData(data++);
    }
}

// class Three - the consumer
class DataConsumer extends Thread
{
    // The DataConsumer retreives the incremented
    // integer that the DataProducer placed in the Q.
    Q q;

    DataConsumer(Q tmpQ)
    {
        q = tmpQ;
        start();
    }

    public void run()
    {
        while(true)
           System.out.println("Got: " + q.getData());
    }
}
//class Four - the dummy class
class Main
{
    // This class gets the ball rolling.

    public static void main(String[] args)
    {
        Q q = new Q();
        DataProducer producer = new DataProducer(q);
        DataConsumer consumer = new DataConsumer(q);
    }
}
```

The output of the preceding class system resembles the following:

```
.
.
Put: 21
Got: 21
Put: 22
Got: 22
Put: 23
Got: 23
Put: 24
Got: 24
Put: 25
Got: 25
Put: 26
.
.
```

TECH NOTE:

It is important to note that there are some methods that are no longer used. The methods stop(), suspend(), and resume() are all deprecated and should never be used. These methods are considered to be inherently unsafe.

DEADLOCKS

Another common timing problem with threads is deadlock. This occurs when two or more threads from different objects are waiting for the same object monitor. Consider the situation illustrated in Figure 19-1.

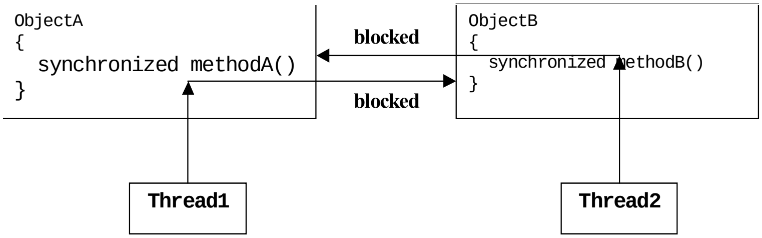


Figure 19-1: Deadlock

`Thread1` enters `methodA()` and obtains `ObjectA`'s monitor. It then proceeds to `methodB()` for further processing. However, at the same time, `Thread2` obtains `ObjectB`'s monitor, then proceeds to `methodA()`. The two threads are now deadlocked, neither willing to give up its own object monitor.

There are two different ways to deal with a deadlock. The programmer may either attempt to detect a deadlock and recover from it once it has occurred, or he may write code so that a deadlock cannot occur. The second is normally the easier and preferred method. Unfortunatly, Java provides no built-in mechanism to avoid deadlocks, and the programmer is responsible.

GRADUATING TASK #6: ENHANCING THE DIGITAL CLOCK WITH ADVANCED THREAD TECHNIQUES

In the previous exercise, you created a `DigitalClock` class that had four buttons: `Start, Stop, Suspend` and `Resume`. Only the `Start` button was implemented. In this exercise, you will use more advanced threading techniques to implement functionality in the remaining three buttons.

1. Add a method to the `Timer` class called `public void stopTimer()`. Invoking this method will stop the thread.
2. Add a method to the `Timer` class called `public void suspendTimer()`. Invoking this method will suspend the thread.
3. Add a method to the `Timer` class called `public void resumeTimer()`. Invoking this method will resume the thread.
4. Event-enable the `Stop`, `Suspend` and `Resume` buttons on your interface so that when those buttons are clicked, the appropriate method of the `Timer` class will be called.
5. *(Optional)* At this point, no code has been implemented in the `startTimer()`, `stopTimer()`, `suspendTimer()` or `resumeTimer()` methods to create a robust system.

   o Modify your `startTimer()` method so that repeated attempts to call this method will *not* repeatedly attempt to start your `timerThread`.

   HINT: *Test for* `null` *on your* `timerThread` *instance variable, defined in the hint following Step 3 of the previous graduating task.*

   o Modify the `stopTimer()` method so that it will stop the `timerThread` only if it is not `null` and if `doRun` is `true`.

   NOTE: *Be sure to set your instance of the* timerThread *to* null, *to aid the* startTimer() *method.*

   o Modify the `suspendTimer()` method so that it will suspend the `timerThread` only if it is not `null` and `doSuspend` is `false`.

   o Modify the `resumeTimer()` method so that it will resume only if the `timerThread` is not `null` and `doSuspend` is `true`.

SUN CERTIFICATION

Monitors and synchronized methods are included on the Sun certification examination. Although not discussed in this section, any block of code can be synchronized by using the keyword synchronized, followed by the object whose monitor the code needs to acquire. For example, the following syntax will synchronize the currentAccount object for a block of code:

```
synchronized (currentAccount)
{
    // this code would be synchronized
}
```

Although the resource will be an object reference, if the data to be locked is a class (static) variable, then it is appropriate to synchronize the class name.

SUMMARY

This chapter discussed several methods used to protect data that is sensitive to change caused by multiple threads. Remember that the more object monitors you place on a program, the greater the likelihood of deadlock occurring. Experience will teach you what must be monitored.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What exception is thrown if a thread uses a wait(), notify(), or notifyAll() without having acquired a lock for the object?

   ....

   ....

2. True or False: Java prevents certain kinds of deadlock (locks that are never released) by throwing an exception.

   A. True

   B. False

CHAPTER 20

Streams and Serialization

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Define a stream.
- Differentiate between byte streams and character streams.
- Recognize the abstraction of byte streams through the `InputStream` and `OutputStream` classes.
- Recognize the abstract character streams through the `Reader` and `Writer` classes.
- Create and use `File` objects.
- Use `System.in` and `System.out` to perform stream operations.
- Nest streams using wrapper classes to enhance basic stream behavior.
- Perform `File` I/O.
- Define Object serialization.
- Use serialization to save an Object to a file, then deserialize that Object.
- Explain the transient keyword and issues relative to security.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What is a Stream?

2. Can Java create and check properties of files and directories?

3. What is Serialization?

INTRODUCTION

Communication between Java and computer hardware is accomplished through streams, which have been used throughout this book. The `System.out.println()` method is an example of stream usage. Java automatically initializes an output stream connection to the command line. Without this function, programs would have no way of displaying output to the user.

WHAT IS A STREAM?

A **stream** is a path of information from a source to a destination. Though somewhat vague, it is a widely applicable concept.

Some examples of elements that form a path of communication between a source and a destination are:

- A keyboard (source) and a monitor (destination).
- A page of text on a monitor (source) and a file on a hard drive (destination).
- A file on a hard drive (source) and a monitor (destination).
- A Web server (source) and a browser (destination).
- A keyboard (source) and a string (destination).
- A text file (source) and a printer (destination).

It may seem daunting that a programmer should have to develop various programming mechanisms to handle every stream possibility that exists. However, the process of streams is a well-abstracted concept. Java takes advantage of this abstraction and provides a rich set of classes to handle a wide variety of stream situations.

Because of the numerous ways that streams can be used, Java provides dozens of stream classes. C provides only one type of stream, and Visual Basic only provides three; in that respect, Java is much more complicated. However, by providing solid guidelines on how to handle specific stream situations, Java code will be more robust and error-free.

Although there are dozens of streams, a programmer does not need to learn the more than 50 different stream classes. Streams are well-organized, and can be classified as one of four types: `InputStream`, `OutputStream`, `Reader`, and `Writer`.

InputStream, OutputStream, Reader and Writer

With the introduction of the JDK 1.0, all streams in Java were subclasses of either an `InputStream` or an `OutputStream`.

`InputStream` is an abstract class that provides the framework from which all other input streams are derived. At the core of the `InputStream` is the `read()` method. The basic `read()` method is abstract, but for the subclasses of `InputStream`, the method will simply read a byte of information from a stream and return the byte as an `int`.

`OutputStream` is an abstract classt that provides the framework from which all other output streams are derived. At its heart is the `write()` method, which takes an `int` and writes a single byte of information to an output stream. All other stream input and output is simply a variation of `InputStream` or `OutputStream`. You will see many variations of the `read()` and `write()` methods, but remember that they are only variations; the heart of input and output with the JDK 1.0 are the abstract classes `InputStream` and `OutputStream`.

If all streams can be represented as variations of the `InputStream` and `OutputStream`, then what purpose do the `Reader` and `Writer` classes serve?

In Section I, we said that Java takes an international approach to its characters. It adapted a 16-bit representation of characters referred to as Unicode so that it could support many languages. The problem with the `InputStream` and `OutputStream` of the JDK 1.0 is that they were not designed to handle Unicode. For those classes that attempted to handle Unicode, it was an incomplete solution.

The `InputStream` and `OutputStream` classes are designed to handle byte streams, while the `Reader` and `Writer` classes are designed to handle character streams. In fact, the `abstract Reader` and `Writer` classes serve the exact same role in character streams that the `InputStream` and `OutputStream` classes serve in byte streams. All other `Reader` and `Writer` streams are merely subclasses of these two.

TECH TIP:

Unicode characters are represented using a word, 16 bits; however, ASCII characters are represented using a byte, 8 bits. For this reason, it is very difficult for a stream that reads and writes bytes to deal with Unicode characters. However, a character stream that reads and writes words can easily handle bytes.

FILES

While `File` objects are used heavily with streams in reading and writing to a disk, a `File` is not a stream.

`File` objects encapsulate what you would expect of a `File` class. Once you have an instance of a `File` object, you can find the length of the file, determine whether the file is readable or writable, rename the file, etc. With a `File` object, you can also create directories or obtain directory listings.

Representing a file and its directory structure on many different operating systems poses a problem. The easiest solution uses the `FileDialog` class; this class will automatically respect the file conventions on the local system. The user will be able to choose files and directories in a manner that is consistant with the system they are using.

We will examine the `File` class to see the many functions you can perform with it, and how you can address some of the cross-platform issues. Table 20-1 provides a listing of the methods you will use.

Table 20-1: Methods of the `File` class

Method	Description
exists()	Returns a boolean describing whether or not a File exists
canRead()	Returns a boolean describing whether or not a File can be read
canWrite()	Returns a boolean describing whether or not a File can be written to
getPath()	Returns a String with the path of the File
getFile()	Returns a String with the name of the File
getAbsolutePath()	Returns a String with the absolute path of the File
mkdir()	Creates a directory from a File object

Instantiating a File object

You can instantiate a `File` object in the manner you would expect:

```
File myFile = new File("SomeFile.txt");
```

In this case, you are creating a `File` object in the current directory of your program. If you wanted to specify a `File` in a different directory, you must consider the operating system being used.

On a Windows system, you can create a `File` object in another directory. See the following:

```
File myFile = new File("/MyDocs/Java/SomeFile.txt");
File myFile = new File("\\MyDocs\\Java\\SomeFile.txt");
File myFile = new File("C:\\MyDocs\\Java\\SomeFile.txt");
```

TECH NOTE:

Notice that you had to double the backslashes in the second two examples. Windows interprets a single backslash as an escape character.

Windows is very flexible. Other systems are not, and you may have to use syntax such as the following:

```
String myTmpFile = "/MyDocs/Java/SomeFile.txt";
myTmpFile = myTmpFile.replace ('/', File.separatorChar);
File myFile = new File(myTmpFile);
```

Working with a File object

Methods of File

Once you have your `File` object, you can use it for many functions. The following code demonstrates a few examples. Assume that you have a file called `SomeFile.txt` in your local directory that contains text.

```
import java.io.*;

class FileTests
{
    public static void main(String[] args)
    {
        File myFile = new File("SomeFile.txt");
        System.out.println("Exists:" + myFile.exists());
        System.out.println("Can Read:" + myFile.canRead());
        System.out.println("Can Write:" + myFile.canWrite());
        System.out.println("Path:" + myFile.getPath());
        System.out.println("Name:" + myFile.getName());
        System.out.println("AbsolutePath:"+myFile.getAbsolutePath());
        System.out.println("Length:" + myFile.length());
    }
}
```

Directories

The File object is also the means by which you create directories and make directory listings. If you want to make a directory called MyDirectory in your current directory, you can create a File object with the name of the directory you want, and call the mkdir() command.

```
File myDirectory = new File("MyDirectory");
myDirectory.mkdir();
```

If you want a listing of all files in a directory, you can again use the File object. In this example, you will make a listing of your local directory.

```
File myLocalDir = new File(".");
String[] dirListing = myLocalDir.list();
for(int i = 0;i<dirListing.length;i++)
{
    System.out.println(dirListing[i]);
}
```

File Dialog (revisited)

In the AWT section of this course, you learned how to instantiate a FileDialog and display it. Now you will learn how to work with FileDialog's ability to deliver a file name.

To cause a FileDialog to pop up, instruct it to display itself as follows:

```
FileDialog fileDialog=new FileDialog(this,"My Dialog",
  FileDialog.LOAD);
fileDialog.setVisible(true);
// ... on to the next line in the program...
```

At this point, assuming the `FileDialog` is modal, the pop-up window will appear and wait for the user to respond. Once the user has selected the file, the program flow returns to the next line in the program. This is where you get the file name.

```
// ... the next line in the program after the user has
// selected a file.
// Get the absolute path of the file name in case the
// user browses to a directory other than the
// current one.
String myFile = fileDialog.getDirectory() + fileDialog.getFile();
```

Now the program has the absolute file name.

STREAM CLASSES OF JAVA.IO.*

Past sections have provided a class hierarchy to show the arrangement of classes discussed in the section. It is difficult to do this with the stream classes, because they vary greatly. Table 20-2 lists the most common classes.

Table 20-2: Useful Stream classes

Class	Description	Useful Methods
Classes for Byte **Streams – JDK 1.0**		
InputStream	The super class for all other byte input streams	**read()** – An abstract method that reads a byte and returns it as an int
OutputStream	The super class for all other byte output streams	**write()** – An abstract method that writes an int representation of a byte
PrintStream	An effective stream for writing lines of text to an output stream	**println()** – A useful method for outputting text formats
DataInputStream	An input stream filter class that allows formatting for incoming byte information	**readLine()** – Creates a line of text (but was deprecated)
BufferedInputStream	An input stream filter class for adding buffering to input streams	**read()** – Reads a specified byte length into its buffer

Table 20-2: Useful Stream classes

Class	Description	Useful Methods
FileInputStream	Creates an input stream from a File	read() – Reads a byte of information from a File
LineNumberInputStream	An input stream filter class that counts the number of lines read	read() – Reads the byte stream tracking the lines read getLineNumber() – Returns the number of lines read
Classes for Character Streams – JDK 1.1 and higher		
Reader	The super class for all other character input streams	read() – An abstract method that reads a Unicode character from an input stream
Writer	The super class for all other character output streams	write() – An abstract method that writes a Unicode character to an output stream
PrintWriter	An effective stream for writing lines of text to an output stream	println() – A useful method for writing to a character output stream in text format
BufferedReader	An input stream filter class for adding buffering to input streams	read() – Reads a specified character length into its buffer
InputStreamReader	A converter class that converts byte streams to character streams for input	read() – Reads a byte from a byte stream and converts it into a character stream
FileReader	Creates an input stream from a file	read() – Reads a character from a file
OutputStreamWriter	A converter class that converts byte streams to character streams for character output	write() – Writes a character to an output character stream
FileWriter	Creates an output stream to a file	write() – Writes a character to a file
LineNumberReader	An input stream filter class that counts the number of lines read	read() – Reads the character stream tracking the lines read getLineNumber() – Returns the number of lines read

Table 20-2: Useful Stream classes

Class	Description	Useful Methods
All Stream Classes		
flush()	Used with Writer and OutputStream classes to force any buffered bytes out	Must be used to see any output; ensures that output streams are flushed
close()	Closes input or output streams and frees system resources, which are limited	Automatically flushes output streams

System.in and System.out

To gain experience with streams, you should practice with them. Fortunately, Java provides the equivalent of standard input (STDIN) and standard output (STDOUT) in other programming languages: `System.in` and `System.out`.

`System.in` is an `InputStream` whose source is the keyboard. Keys pressed on the keyboard are made available to the Standard.in stream. `System.out` is an `OutputStream` (in fact, it is the `PrintStream`) whose output is the display monitor. You have been able to use commands such as `System.out.println("text")` because `System.out` is a `PrintStream` and it has a method called `println()` that takes a `String` as a parameter.

You will now use the `System.in` and `System.out` streams to practice with stream concepts.

Reading bytes from System.in

The most fundamental use of a byte input stream is reading bytes via the read() method. You will notice with I/O operations that you are often forced to deal with exceptions. This is reasonable if you consider all that can go wrong with I/O, including downed networks, faulty hard drives, and deleted files. The following application reads information from the system input stream. Run the application, then enter alphanumeric characters. You must press ENTER after each character whose stream you want to read. This application will read byte information you enter, then return the byte as an int. The result is that you will see ASCII value for each character you enter.

```
import java.io.*;

class StreamExample01
{
    public static void main(String[] args)
    {
        // Access the System.in InputStream
        InputStream is = System.in;
        /
        {
            int i;
            // Loop until the letter 'q' is hit
            while((i = is.read()) != 113)
            {
            System.out.println(i);
            }
        is.close();
        }
            catch(IOException e)
        {
            System.out.println("main(): " + e);
        }
    }
}
```

Following is sample output for StreamExample01:

```
d

<ENTER>

100
13
10
r

<ENTER>



114
13
10
```

Oddly enough, this output is correct. Recall that the read() method of this application reads a byte from the input stream, and returns the byte as an int. When you typed d then the RETURN key to flush the stream, you saw the ASCII integer representation of the letter d (100). Then the 13 and 10 are displayed, these numbers represent \r and \n, which are escape characters for the return and end-of-line terminators used by Windows. Finally, you exited the while loop by entering integer 113, which is the ASCII value for the letter q.

If you want to see the ASCII character output instead of the integers, modify the code and cast the int into a char as follows:

```
System.out.println((char) i);
```

After casting the `int i` as a `char`, you receive the following output:

```
d
d


r
r
```

Now you see the appropriate characters with the new line and end of line characters.

As Java becomes more international, there will be an increased use of Unicode and a deprecation of some byte stream classes, constructors, and methods. It is important for the Java programmer to consider the character stream classes as his or her staple approach to I/O.

Converting a byte stream into a character stream

Most of the `Reader` and `Writer` classes, character stream classes, cannot work with byte streams and must use character streams. Java provides the necessary framework to convert a byte stream into a character stream via `InputStreamReader` and `OutputStreamWriter`. `InputStreamReader` converts a byte stream into a character stream for input, while `OutputStreamWriter` allows you to specify a byte stream to which you can direct your character stream.

This conversion is natural as shown by the following example. Although you see no apparent difference between this example and the previous example, remember that program output is now a character stream that transmits Unicode, instead of a byte stream that transmits bytes.

```
import java.io.*;

class StreamExample02
{
    public static void main(String[] args)
    {
        InputStream is = System.in;
        InputStreamReader isr = new InputStreamReader(is);
        try
        {
            int i;
            while((char)(i = isr.read()) != 'q')
            {
                System.out.println((char)i);
            }
            isr.close();
        }
        catch(IOException e)
        {
             System.out.println("main(): " + e);
        }
    }
}
```

Wrapper streams

Rarely would you want to work with single characters at a time; typically, you want to read in lines of text. Java provides a clever approach to this demand by allowing you to wrap streams within streams to gain the results you seek. The streams you use to wrap other streams are not actually streams; they are filter classes that use a raw stream to produce formatted results.

In the two previous examples, you were able to read only one character at a time. If you want to read a whole line at a time, use a wrapper class. In this case, you need to use the `BufferedReader` class.

The `BufferedReader` class offers two elements. The first is a method called `readLine()`, which will read entire lines of text from a character stream. The second element offered is buffering. Buffering delivers an efficiency that non-buffered streams cannot. A buffered stream will read or write information into a buffer. Once the buffer is filled, subsequent reads or writes will come from the buffer, instead of making repeated system reads or writes. System reads and writes eat into resources. It is far less resource-consuming to perform these reads and writes from a buffer. Streams should almost always be buffered.

Before you implement the necessary code, study Figure 20-1 to help you understand this wrapper approach.

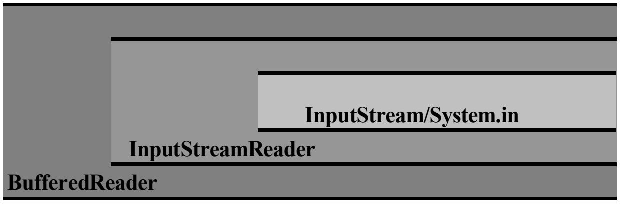


Figure 20-1: Wrapper streams

```
import java.io.*;

class StreamExample03
{
    public static void main(String[] args)
    {
        // Notice the nesting of streams
        InputStream is = System.in;
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);

        try
        {
            String s = null;
            // For a String you test for null
            while((s = br.readLine()) != null)
            {
                System.out.println(s);
            }
            br.close();
        }
        catch(IOException e)
        {
            System.out.println("main(): " + e);
        }
    }
}
```

Sample output for Example03 is as follows:

```
Once upon a time...
Once upon a time...
... there was a fairy princess...
... there was a fairy princess...
```

TECH NOTE:

It is necessary to close only the outermost stream in a series of nested streams. Thus, you closed only the bufferered reader.

File I/O

Reading and writing to a File is one function you can perform with streams. In this example, you will read from one File then write to a second. In essence, you are copying the first File. You will need several more classes.

The FileReader and FileWriter classes allow you to read and write a raw character stream to and from a File. However, you will also need to format the input and output for line input and output. You have already been introduced to the BufferedReader for the purpose of reading a line of text. The PrintWriter will allow you to write a line of text to the output stream and thus, to the File. This stream is not buffered, but you can add buffering if you choose.

For this example, assume there exists a text file called input.txt. The output file will be called output.txt.

Notice that the FileReader and FileWriter classes must be instantiated inside of a try/catch block. You choose to instantiate the BufferedReader and PrintWriter classes in the try/catch because they are dependent on the FileReader and FileWriter classes.

```
import java.io.*;

class StreamExample04
{
    public static void main(String[] args)
    {
        File input = new File("input.txt");
        File output = new File("output.txt");
        FileReader fr = null;
        BufferedReader br = null;
        FileWriter fw = null;
        PrintWriter pw = null;
        try
        {
            // fr and fw must be declared in a try/catch block.
            fr = new FileReader(input);
            fw = new FileWriter(output);
            br = new BufferedReader(fr);
            pw = new PrintWriter(fw);
            String s;
            while((s = br.readLine()) != null)
            {
                pw.println(s);
                // Don't forget to flush the output stream.
                pw.flush();
            }
            // Don't forget to close your streams.
            br.close();
            pw.close();
        }
        catch(IOException e1)
        {
            System.out.println("main(): " + e1);
        }
    }
}
```

Here, you read text from a file and saved that text to another file. In the next chapter, you will see that Java also delivers the means to read and write complete object streams.

SERIALIZATION

In a typical object-oriented system, objects are instantiated and their internal states are modified. After an object has reached a certain state, it may beacceptable that all the information is lost when the system is shut down. However, if it is desirable to save the state of an object, then you need a solution.

Several solutions can make an object persistent. One approach is to use force. If you have an object of a certain type, you must be able to save its type, its internal information, and information regarding references to other objects. You would save this information to a file via a `FileOutputStream`. This process of saving objects is serialization. With the release of JDK 1.1, Java provided a means of serialization that is effective and much easier to implement than the brute-force method.

The applications for serialization are endless. The ability of objects to use objects across different virtual machines is a primary use. RMI, which is Java's version of a distributed object protocol, uses serialization. JavaBeans also use serialization regularly. Sophisticated, graphical beans must be saved upon creation to be used at a later time. Some graphical IDEs will save the interface created by the user via serialization. With the flexibility that objects and the object-oriented paradigm provide in system design, the ability to make these objects and systems persistent is a powerful and useful mechanism.

The process of object serialization

Marking an object for serialization

Although serialization is part of the Java language, Java will assume that you do not want an object to be serialized unless you mark it as such. To mark an object as serializable, you must implement the `Serializable` interface.

```
import java.io.*;

class SampleObject implements Serializable
{
    // You will implement some sample code with this
    // sample object.
    String name;
    int age;

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
    public void setAge(int age)
    {
        this.age = age;
    }
```

```
        public int getAge()
    {
        return age;
    }
}
```

The Serializable interface itself is rather mundane:

```
public interface Serializable
{
}
```

This code points out an important programming practice. You will often want to tag an object as a certain type without specifying any functionality.

Writing the object to a file

In your example, you are going to save your SampleObject to a file. You can send your object anywhere that you would normally direct an output stream.

To write an object to a stream, use the ObjectOutputStream class. This class is really just a wrapper class, as you have used before, to wrap a FileOutputStream . You then will use the writeObject() method of the ObjectOutputStream class to serialize the object.

```
import java.io.*;

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

        try
        {
            // Get an instance of the SampleObjects and
            // set their state.
            SampleObject originalObj1 = new SampleObject();
            SampleObject originalObj2 = new SampleObject();
            originalObj1.setName("Mary Smith");
            originalObj1.setAge(32);
            originalObj2.setName("John Doe");
            originalObj2.setAge(42);

            // To write the object, you will need a
            // FileOutputStream and an ObjectOutputStream.
            FileOutputStream fos = null;
            ObjectOutputStream oos = null;

            fos = new FileOutputStream("SerializedObj.obj");
            oos = new ObjectOutputStream(fos);
            oos.writeObject(originalObj1);
            oos.writeObject(originalObj2);
            oos.close();
        }
        catch(Exception e)
        {
            System.out.println("Main: main(): " + e);
        }
    }
}
```

Saving the serialized object with the file name `SerializedObj.obj` was arbitrary, including the file extension. Also, note that you are catching the exception using `Exception`. This step is not recommended; however, there are many exceptions to catch in these examples. To help keep the focus on topic, we have elimnated as much code as possible.

Reading the serialized object from a file

Once you have a serialized object saved to disk, you can read it in much the same manner that you wrote it. Instead of a `FileOutputStream`, you now need a `FileInputStream`; instead of an `ObjectOutputStream`, you have an `ObjectInputStream`.

```
import java.io.*;

class Main
{
    public static void main(String[] args)
    {
        try
        {
             // Get an instance of the SampleObjects
             // and set their state.
             SampleObject originalObj1 = new SampleObject();
             SampleObject originalObj2 = new SampleObject();

             originalObj1.setName("Mary Smith");
             originalObj1.setAge(32);
             originalObj2.setName("John Doe");
             originalObj2.setAge(42);

             // To write the object, you will need a
             // FileOutputStream and an ObjectOutputStream.
             FileOutputStream fos = null;
             ObjectOutputStream oos = null;
             fos = new FileOutputStream("SerializedObj.obj");
             oos = new ObjectOutputStream(fos);
             oos.writeObject(originalObj1);
             oos.writeObject(originalObj2);
             oos.close();
             // To read the objects, you will need a
             // FileInputStream and an ObjectInputStream.
             FileInputStream fis = null;
             ObjectInputStream ois = null;
             fis = new FileInputStream("SerializedObj.obj");
             ois = new ObjectInputStream(fis);
             SampleObject newObj1 = (SampleObject) ois.readObject();
             SampleObject newObj2 = (SampleObject) ois.readObject();
             ois.close();

             // And the results:
             System.out.println("SampleObject1 name: " +
             newObj1.getName());
             System.out.println("SampleObject2 name: " +
             newObj2.getName())
        }
        catch(Exception e)
        {
           System.out.println("Main: main(): " + e);
        }
    }
}
```

Two points should be noted about the preceding example:

- You must read the objects from the file created by an object stream in the same order that you wrote them.
- Because the `readObject()` method returns an `Object`, you must perform an explicit cast the moment you read the `Object` from the object stream. That is,

```
  SampleObject newObj1 = (SampleObject) ois.readObject();
  SampleObject newObj2 = (SampleObject) ois.readObject();
```

These steps are necessary if you want to use the methods available with this typed object. However, keep in mind that you should always keep track of the type of object you are reconstituting. As your serialized systems become bigger (and less manageable), you should consider other ways of determining object type.

The objects `originalObj1` and `newObj1` do not occupy the same memory space. One object will not affect the other. Of course, the same is true for `originalObj2` and `newObj2`.

Transient variables and security

It is possible that within an object marked as serializable, you do not want some variables to be put into a persistent state. For example, you might have a variable in your object that always depends on the current system date. In such a case, there is no purpose in serializing this `date` variable because it will have no meaning when the object is deserialized. You can prevent an instance variable from being serialized by using the `transient` keyword. Consider the following example:

```
import java.util.*;

class TrivialObject implements Serializable
{
    transient Date currentDate;
    transient private int accountID;
}
```

In this code, you made the `currentDate` variable `transient` because it relies on the current date and need not remember any past dates. But, is the `accountID` variable also persistent?

One of the disadvantages of serialization is that private members are also serialized. In essence, this information can be exposed or corrupted during the serialization process. For security, you should never underestimate the desire or ability of others to gain access to this information. Therefore, a programmer may choose not to serialize the `accountID` variable. A `TrivialObject` instance would have to obtain the `accountID` in some other way.

TECH TIP:

> `Static` *variables are (by default) not serializable. The* `transient` *keyword has no significance with a* `static` *variable.*

GRADUATING TASK #7: BUILDING A SIMPLE WORD PROCESSOR

The ability to handle I/O is a staple element in any programming language. In this exercise, you will perform I/O with `Strings` by building a simple word processor to read a text file's contents to a `JTextArea`. Additionally, you will be able to save the contents of the `JTextArea` to a file under a different name. The contents of the `JTextArea` will be cleared as well.

1. Create a class called `WordProcessor` that extends `JFrame`. This class will serve as the GUI for your word processor and should resemble Figure 20-2.

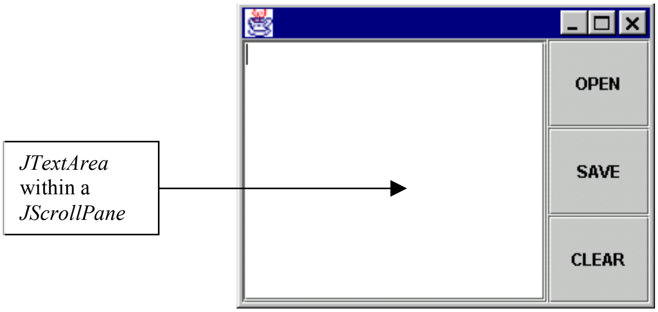


Figure 20-2: Word Processor GUI

NOTE: You should use the `FileReader`*,* ***Buffered***`Reader`*,* `FileWriter` *and* ***Print***`Writer` *classes for Steps 2 and 3.*

2. Event-enable the Open button. Upon clicking this button, a `JFileChooser` should appear in "open" mode, starting with your root directory (c:\ on a Windows system). Selecting (or typing) a text-based file to open will output its contents into the `JTextArea`.
3. Event-enable the Save button. When this button is clicked, a `JFileChooser` should appear in "save" mode, starting with your root directory (c:\ on a Windows system). Selecting (or typing) a file name will save the contents of the `JTextArea` to the hard drive under that chosen file name.
4. Event-enable the Clear button. When this button is clicked, the contents of the `JTextArea` will be cleared.
5. (*Optional*) In addition to displaying the contents of the selected file, also show the name, length and date last modified of the file in the `JTextArea` (consult your API).

SUN CERTIFICATION

The Sun certification exam covers two basic types of streams: filter streams and file streams. The two basic file streams are `FileInputStream` and `FileOutputStream`, which are derived from the `abstract` classes `InputStream` and `OutputStream`, respectively. The constructors for `FileInputStream` and `FileOutputStream` will return a stream object, which is then used to create the filter streams. In preparing for the exam, you should understand how to use the constructors to create a stream.

The filter streams are subclasses of `FilterInputStream` and `FilterOutputStream`. Filter streams are used to process bytes in different ways. For example, the class `DataInputStream` allows you to read `Strings` and primitive, unlike the bytes read by `FileInputStream`. You should know the subclasses of `FilterInputStream` and `FilterOutputStream` for the exam. They are as follows:

```
FilterInputStream
LineNumberInputStream
BufferedInputStream
DataInputStream
PushbackInputStream
FilterOutputStream
PrintStream
BufferedOutputStream
DataOutputStream
```

At the time of this writing, the `Reader` and `Writer` streams were not included in the examination, so use the `DataInputStream` and `PrintStream`.

Questions about the `File` object, which was discussed earlier in the lesson, also appear on the exam.

The `RandomAccessFile` object is tested on the exam. The constructor for the object gives you functionality to assign a mode, either `r` (read) or `rw` (read/write) for a `File`. `RandomAccessFile` has a method called `seek()`, which allows you to find a certain point in the file. The `RandomAccessFile` object implements `DataInput` and `DataOutput`, so it contains methods for reading and writing various primitives and `Strings`. However, it is not an `InputStream` or `OutputStream`, therefore it cannot be used with the filter streams.

> *NOTE: It is very common for questions about stream and filter constructors to appear on the exam. You should know the hierarchy of all of the streams discussed in this chapter. Also, you should know the possible modes for files and how to use them.*

Serialization composes only a small portion of the Sun Certified Java Programmer exam. You might be asked a simple question about the purpose of serialization. However, you will use serialization extensively in your programming assignment for the Sun Certified Java Developer exam.

SUMMARY

Although many different streams exist, they can be easily organized into a few broad categories. The differences depend on what types of data each stream manipulates, and the direction of the stream (input or output). In this chapter, you dealt with primitive data types and `Strings`. You also learned how to save the persistent state of an object using serialization. Although you saved object states to a file, you will learn in the next chapter that the same streams can be sent across a network, making the implementation of your chat server possible.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. Can a BufferedReader have a constructor with a InputStreamReader in it?

2. Can private variables be made transient?

3. What command do you use if you are writing to a file with an OutputStream class and you want to prevent 'garbage' from being written?

4. Will this compile, and if so, where will the input come from?

```
InputStream is = System.in;
InputStreamReader isr= new InpuStreamReader(is);
BufferedReader br= new BufferedReader (isr);
```

Appendix A—Answers to Pre-Test and Post-Test Questions

CHAPTER 1

Pre-Test Answers

1. .java
2. .class

Post-Test Answers

1. To compile a file this is what needs to be typed.

   ```
   javac filename.java
   ```

2. To execute a program this command needs to be entered

   ```
   java filename
   ```

CHAPTER 2

Pre-Test Answers

1. There are 8 types: double, float, long, short, int, char, boolean and byte.
2. int test = 5;
3. The value of a short can be assigned to a long, but the value of a long cannot be assigned to a short with out an explicit cast.
4. The "=" symbol is used for assignement, and the "==" symbol is used for comparison.

Post-Test Answers

1. boolean and byte both occupy 8 bits. short and char both occupy 16 bits. int and float both occupy 32 bits, and double and long both occupy 64 bits.
2. ||
3. A local variable is defined inside of a method and is only within scope in that method. Class variables are defined underneath a class outside of all methods and are accessible everywhere within the class. Local variables must be initilized before use.
4. a: 2

   b: 1
5. A static variable maintains a single copy in memory that is in use by all instances, while an instance variable establishes seperate copies for each object instance.

CHAPTER 3

Pre-Test Answers

1. The if statement is used to control the flow of a program by checking certain boolean expressions.
2. The try, catch, finally construct is the mechanism which provoides exception handling in Java. This way the program may handle errors which are generated during runtime, providing robustness code.

Post-Test Answers

1. The variable in the switch statement must be of the int, short, byte or char types.
2. The break statement is optional inside of a switch construct, but it should be used to insure program correctness.
3. The loop will iterate as least once.

CHAPTER 4

Pre-Test Answers

1. An inner class is a class that is located inside of another class. This construction is used mostly for anonymous calls to Event Handlers and for specialized functions.
2. Command line arguments are found in the args[] String array that is declared as a parameter in the main method.

Post-Test Answers

1. The index of the first element of the array is zero.
2. A method is overloaded in Java by creating two methods that have the same name and different parameters.
3. Local. When the method ends, the all local variable becomes eligible for garbage collection.

CHAPTER 5

Pre-Test Answers

1. An array is linear collection of variables of a single data type that can be accessed via an integer index.
2. To declare an array of ten elements

   ```
   int arr [] = new int[10];
   ```

3. Yes you can. It is the same as converting a int to a long

Post-Test Answers

1. You access the word is by using

   ```
   public static void main (String args[]){   String word = args [3];
   ```

2. B, C
3. Use a for loop and assign the counter to the array.

CHAPTER 6

Pre-Test Answers

1. A class is a logical encapsulation of related data and code.
2. An object is an instance of a class.
3. A method is a Java subroutine consisting of a header that indicates the return type, name, and parameters of the method, and a method body that contains the code that the method executes.

Post-Test Answers

1. A class is a logical encapsulation of data. An object as an instance of a class.
2. `Employee Dave = new Employee()(JRE);(JRE)`
3. `double someDouble = Dave.getPay();`

CHAPTER 7

Pre-Test Answers

1. Encapsulation is when the code is hidden from the user and the data and implementation are grouped together.
2. Create a class Employee and a class Nurse. The class Nurse should be a subclass of Employee and the Nurse class should have a method called getPay() as well, this method will override the getPay() method of class Employee for all Nurse objects.

Post-Test Answers

1. To create a new instance from the class Employee, the following code would be used.

   ```
   Employee jim = new Employee();
   ```

2. To call a parental class constructor, use the keyword super() with any arguments as parameters.

   ```
   super(string, integer);
   ```

3. LASTFIRST would print because the calculate (int number) is called first and then calculate (doublenum) would be called

CHAPTER 8

Pre-Test Answers

1. A constructor initializes any graphics, variables and uses anything from the super class.
2. Use the command form: Class.method();
3. No, the String class is immutable.

Post-Test Answers

1. Hello
2. Hello World
3. Sub Super

CHAPTER 9

Pre-Test Answers

1. Public class className implements interfaceName
2. Polymorphism is using one method (JRE) call(JRE) to invoke several different methods.
3. An incompletely defined class used to collect properties or behaviors of other classes with similiar attributes. Abstract classes are never instantiated.

Post-Test Answers

1. This code can be executed. The class Student is never directly instantiated. The classes Undergraduate, Graduate and Alumni may be considered objects of class Student because they extend the abstract class. This grants them access to all of the methods of the class Student. The program, then, finds the amount owed by all students.

CHAPTER 10

Pre-Test Answers

1. Listener classes accept various actions and implements the result.
2. The Math class provides various methods to perform on variables.
3. It returns an long that is closest to the float or double provided.
4. The elements in a Set are unique.

Post-Test Answers

1. The protected access modifier allows package access only and derived classes. The synchronized access modifier denotes that it only available to one thread at a time.
2. The code returns 9.
3. A List.
4. A Map implements a key value pair and contains no duplicate keys.

CHAPTER 11

Pre-Test Answers

1. The AWT is the Abstract Windowing Toolkit package that allows Java programmers to create user interfaces that use native operating system components.
2. Swing is a set of light weight components that are built on the top of the AWT. It also provides a pluggable look and feel to allow for the application to change how it looks. All Swing components implement Java Bean technology.
3. A JApplet component would be used if you wanted use Swing components in a Web browser.
4. JavaBeans technology is a specification of a software component model. JavaBeans are designed according to a strict specification so they can be easily integrated into sophisticated applications as separately developed components. JavaBeans are self contained code that can be implemented with the idea of "Software Reuse".

Post-Test Answers

1. The first line of code creates an ImageIcon using the image save_button.gif, the second line creates a JButton with the image on it.
2. A JScrollPane adds scrolling functionality to other components.
3. A modal component is one which blocks its parent component until it is properly responded to. A modal component will retain focus until an action has been taken on the window. A JFileChooser is an example of a modal Swing component.

CHAPTER 12

Pre-Test Answers

1. A Layout Manager dictates how graphical components will be placed.
2. They are located in java.awt package.

Post-Test Answers

1. FlowLayout is the default Layout Manager.
2. There are 5 fields. North, South, East, West and Center.
3. GridBagLayout will not allow components to change size if the weights are set to zero.

CHAPTER 13

Pre-Test Answers

1. The AWT package in Java provides a set of classes to create graphics, the Graphics class, the Color class, and the Font class.
2. To draw text and shapes on the screen you would use the Graphics class and its appropriate methods.
3. Java is capable of displaying 16.7 million colors.
4. There are only a few fonts available to all JVMs in order to provide complete platform independence in Java.

Post-Test Answers

1. The coordinate system in Java has its origin at the upper-left corner of the system.
2. The code would create a JFrame and draw 75 blue rectangles increasing the size of the rectangles with each pass through the for loop.
3. The code would use a SansSerif font, it would be italic and 20 point.
4. The code would be used as a typical main method in a Java program. It provides for the event of the user closing the window in which the program is running.

CHAPTER 14

Pre-Test Answers

1. Any defined occurrence from the user or the program.
2. Button clicking, keystrokes, mouse movements, etc.

Post-Test Answers

1. Yes, from the Abstract Button class.
2. How is the event generated?

   How does the generator of the Event know where to send the Event object?

   How does an Event listener receive the Event object?

   What happens once a listener has received an Event?

CHAPTER 15

Pre-Test Answers

1. An inner class is a class that is defined inside another class.

Post-Test Answers

1. Inner classes can be used for event handlers, and can be called anonymously. It is for small operations that are only to be used once.

CHAPTER 16

Pre-Test Answers

1. An applet is Java code that is designed to run in a browser.
2. Yes, a main() method may be added to applet to make it into an application.

Post-Test Answers

1. You can pass parameters to can applet by using the code below

   --In the HTML page--

   ```
   <APPLET CODE="MYAPPLET.CLASS"<param name="number" value="4">
     </APPLET>
   ```

   --In the Applet Code --

   ```
   int appletnumber;
   appletnumber = new Integer(getParameter("number") ).intValue;
   ```

2. The security restrictions are:
   - Applets cannot run any executable code on the local machine.
   - Applets can only establish a network connection with the host they were downloaded from.
   - Applets can neither read nor write disk information to the local machine.
   - All pop up windows have a warning at the top of the window.

CHAPTER 17

Pre-Test Answers

1. An Exception is an abnormal condition which occurs during execution of the program and which a reasonable program may wish to handle. Exceptions that may occur in the usual operation of the Java Virtual Machine are descended from RuntimeException.
2. An Error is a serious condition which should stop a program while an Exception is a condition from which the program may recover. An Error may not be caught, as a program should cease execution if one occur.

Post-Test Answers

1. The code will print

   ```
   ERROR
   TEST
   ```

2. The throw keyword passes the Exception to the caller method to be dealt with there.
3. You can:
   - Ignore the Exception
   - Handle the Exception in a try/catch block
   - Rethrow the Exception to the calling method
   - Throw the Exception and handle it in the try/catch block of the caller method.

CHAPTER 18

Pre-Test Answers

1. A thread is a designated portion of a program that may execute concurrently with other threads and programs. A thread may be thought of as a lightweight process that executes within the memory space of its parent process. Java's built in capabilities provide the programmer with a rapid way to develop multithreaded applications.
2. A new thread of execution may be created either by extending the Thread class or implementing the Runnable interface.
3. A thread may not run if it is blocked waiting on a synchronized resource currently in use by another object. A thread may also have been preempted by a thread with higher priority or may have been the target of a wait(), sleep(), or similar method.
4. If a method has the synchronized keyword in its header, that method is monitored and can only be executed by one thread at a time.

Post-Test Answers

1. The start() method launches the thread's execution, in turn calling the thread's run() method. The run() method does the main work of the thread and must be overridden in a subclass of Thread, or in the Runnable object. The sleep() method puts a running thread into a sleeping state and it does not become ready for execution again until the sleep time expires. The wait() method tells the thread to sleep until it is notified by another thread via the notify() or notifyAll() methods.
2. Run Thread Thread-0 going to sleep

   Run Thread Thread-0 waking up

   Main Thread going to sleep

   Run Thread Thread-1 going to sleep

   Main Thread waking up

   Run Thread Thread-1 waking up

   No, the timing of concurrent threads is not specified, and program correctness should not rely on a specific timing or ordering of execution.

CHPATER 19

Pre-Test Answers

1. To preform synchronization, Java uses monitors. If an object contains methods that are synchronized, that object is a monitor. The monitor permits one thread at a time to execute synchronized methods on an object. The object is said to be locked when a synchronized method is invoked. Thus, all other threads attempting to call the synchronized object must wait until the synchronized object is finished executing.
2. True
3. The notify() method allows a waiting thread to become ready for execution, while the notifyAll() method permits all threads waiting for the object to become ready for execution.

Post-Test Answers

1. IllegalMonitorStateException
2. A. True

CHAPTER 20

Pre-Test Answers

1. A Stream is a one way data channel. Information flows through Streams allowing data to pass from one source to another.
2. Yes. The File class allows for the creation of directories and the ability to check for properties of files and directories. Java may create files with the File class and the OutputStream classes.
3. Serialization is Java's mechanism for preserving the state of an object by writing that object through a byte stream to disk so that byte stream may be read back in at a later date and that object be restored. Serialization is implemented using the Serializable interface.

Post-Test Answers

1. Yes it may. Many streams may be chained together to provide multiple options for filtering I/O.
2. Yes, private variables have to be made transient due to security concerns.
3. Use the flush() method.
4. Yes, it will compile, and the input will come from the keyboard.

Glossary

<APPLET></APPLET>

HTML (Hyper Text Markup Language - the language used to code web pages) tags used to embed an applet.

abstract

A class or method that is incompletely defined, used to collect shared properties, behaviors, or attributes in a single superclass.

Abstract Windowing Toolkit (AWT)

A package of Java classes that support GUI (Graphical User Interface) programming - part of the JFC (Java Foundational Classes).

abstraction

The consideration of an object in terms of its functionality and not its implementing details (see also abstract).

accessor

A method designed to allow outside classes to access information about an object while maintaining the principles of encapsulation and information hiding.

add()

A method of `Container` used to add components to a GUI container.

addItemListener()

One of a series of methods that register a listener object with its source for notification of some specified change of state; this method is always of the form `addSomeListener()`.

ALIGN

Optional attribute which can be imbedded within APPLET tags in HTML code; specifies the position of the applet.

allocate

Java sets aside (allocates) enough space in memory for a data structure as soon as it is fully defined; connected with the idea of instantiation.

ALT

Optional information to embed an applet; allows you to display a comment to a Java-disabled browser.

anonymous inner class

An inner class defined within an expression; not a top level class which is part of a package.

applet

A small Java program that must be run from a Web browser or the Appletviewer; more restricted in permitted activities than applications for security reasons.

Application Programming Interface (API)

The pre-written code that provides the built-in functionality of Java. The term is also used to refer to the organized descriptions of classes and their methods within this pre-written code.

arithmetic operators

Operators that perform manipulation on numbers.

ASCII

A standard means to represent characters as integers ranging from 0 to 127.

atomic process

Refers to an operation that is never interrupted or left in an incomplete state under any circumstance.

bitwise operators

Operators that perform logical manipulation comparing and changing each bit.

***boolean* expression**

An expression that returns true or false; the result of a relational or logical operation.

BorderLayout

A layout manager that adds components in specified regions.

BoxLayout

A layout manager that adds components horizontally or vertically.

break

A statement that causes the current conditional or iteration statement to terminate.

bytecode

A platform-neutral file consisting of bytes that the JVM can execute.

callback

When an object refers back to a class to which it has gained reference.

casting

Converting values from one data type to another.

class

One of the most fundamental parts of Java (and other object-oriented languages) data structure. Classes contain methods, variables, and even other classes. Top-level classes belong to a package.

class member

A variable or method that exists only once for the class regardless of how many objects are created; designated by the keyword `static`.

CODE

Essential information needed to embed an applet; specifies the name of the applet class. Example:
<APPLET CODE=AppletSubclass.class WIDTH=anInt HEIGHT=anInt>

</APPLET>

code block

A sequence of statements contained in curly braces.

CODEBASE

Optional information to embed an applet; specifies a relative directory in which to save the applet class. Example: CODEBASE="someDirectory/"

Color

A class of the AWT that represents colors.

concatenation

The act of adding `Strings` together.

constructor

A special method having the same name as its class. It returns an instance of its class.

Container

An abstract, high-level component that offers `JFrame` and `JPanel` much of their functionality.

cooperative multitasking

A system of multitasking in which one task must willingly yield to another task.

coupling

The level of interdependency between objects.

daemon thread

A thread designed to service user threads. Daemon threads terminate after the last user thread terminates.

dead

The state of a thread once it has been stopped or the `run()` method has exited.

deadlock

A condition that occurs when two or more threads reach an impasse due to competition for object monitors.

default constructor

A constructor without parameters; Java assumes a default constructor until you define your own.

destroy()

The last method called in the life cycle of an applet; should be used to release system resources.

Domain Name System (DNS)

A system that converts domain names into corresponding IP addresses.

dot notation

A reference used to access a member of an object.

drawRect()

A method of the `Graphics` class.

encapsulation

The wrapping of variables and methods together; variables can only be accessed through the methods provided.

entry condition/ exit condition

Condition based on whether the boolean expression is evaluated at the beginning or the end of the loop.

Error

An inoperable condition that can occur in a program and cause it to terminate.

event

An occurrence or happening such as a pre-defined change of state in a component.

event delegation model

The event handling model used in JDK 1.1 and SDK 1.2.

event object

The informational object Java creates in response to an event.

Exception

An abnormal condition that occurs in a program; also the name of a class in `java.lang`. An elaborate system of exception handling functionality exists within the JFC.

extends

The Java keyword that designates the class that a new class is to be derived from.

File

A Java class abstracting the nature of a system file.

FlowLayout

A layout manager that adds components centered on each line, top to bottom, in a flowing manner similar to the layout of HTML pages.

Font

A class of the AWT that represents fonts.

getParameter()

A method of `Applet` used to obtain the name/value pairs sent by the HTML page.

Graphics

An `abstract` class used to get a graphics context of a `Component` or `Image`.

graphics context

The workable representation of the `Graphics` class for a `Component` or `Image`.

GridLayout

A layout manager that adds components in a grid-like format.

handleEvent()

The method most widely used in the JDK 1.0 to handle event processing.

ID

A way of distinguishing the type of event generated.

identifier

Another term for a variable name.

***if* statement**

The basic conditional statement; chooses different control flow paths based on the value of some boolean statement.

ImageIcon

A class that facilitates the display of images; implements the *Icon* interface

immutable

Unable to be changed; a characteristic of `Strings`.

implicit vs. explicit casting

Explicit casting requires you state the type name being converted to; in implicit casting, the type is implied.

import

A statement that makes a package available for use within a class.

index

A numeric reference to a specific element of an array; the only way to access elements of an array.

inheritance

Properties or attributes of a class which are not explicitly stated but passed down from its superclass(es).

init()

The first method called in the life cycle of an applet; it resembles a constructor.

inner class

A class defined within the body of another class.

InputStream

The abstract class from which all other input byte streams are derived.

InputStreamReader

A utility stream that converts a byte stream into a character stream for input.

instance and class variables

Variables defined outside of any method; they are available throughout the class.

instance member

A variable or method unique for each object (instance of the class).

instantiation

The use of a class definition to create an object.

interface

Term used to define a collection of method definitions and constant values which can be implemented by classes that define this interface with the "implements" keyword.

IP (Internet Protocol) address

A numeric configuration used to identify a unique computer on the Internet.

java

The command given to execute bytecode (`*.class`) file.

Java 2 Software Development Kit (SDK)

A set of Java tools available from Sun for free; includes the Java compiler, interpreter and debugger, along with documentation.

Java Virtual Machine (JVM)

The artificial computer that runs Java programs.

javac

The command that compiles a Java source file (`*.java`) into a bytecode file (`*.class`).

JButton

A simple clickable widget.

JComponent

A high-level Swing class from which many other components receive their core functionality.

JDK 1.0

The original release of the Java Software Development Kit, whose event model has now been deprecated.

JFileChooser

A Swing component used to retrieve the names of files.

JFrame

The fundamental Window in Java; it cannot be contained in another container.

JLabel

A Swing widget used to place text in relation to an icon.

JPanel

A low–level `Container` used to hold other `Containers`; can also be used as a drawing surface.

JScrollBar

A Swing widget used to visually select a single value from a range.

JScrollPane

A Swing text-entry widget that adds scrolling functionality to other components.

JTextArea

A multiline text-entry widget in Swing

JTextField

A single-line text-entry widget in Swing.

label

An identifier that points to a specific line of the code to where a loop can break or continue.

layout managers

`Objects` used to control the positioning of `Components` within a `Container`.

***length* property**

A characteristic of an array; more specifically, how many elements it contains.

life cycle

The life of an applet as defined by the `init()`, `start()`, `stop()` and `destroy()` methods.

listeners

Classes that listen for, and usually process, the events generated by an event source. Such classes will implement an appropriate listener interface.

literal

A representation of an actual value for a data type. For example; 12 is a literal integer.

local variable

Variables defined inside of a method; they are not available when the method is not running.

logical operators

Operators &, |, ^, and ~ used on primitive data types to perform bitwise operations.

lower-level container

A `Container` that must be contained in another Container as in a `JPanel`.

main thread

The user thread created by the JVM from which all other user threads are generated.

member

A general term for both methods and variables.

member inner class

An inner class defined at the same scope as a method or instance variable.

method

A "callable" block of code defined within a class.

method signature

The method name and parameter types; the return type is not included in the signature.

modal

A property designating that a component blocks its parent `Component` until properly responded to.

monitor

In essence, the lock and key for an object

multiprocessing

The ability to run multiple processes (applications) in parallel.

multitasking

The ability to perform more than one function simultaneously.

multithreading

The ability to perform multiple lines of instruction within a single process.

mutator

A method that modifies a variable.

MyCloseableFrame$ MyButtonListener.class

The $ is the notation that the Java Compiler uses to identify an inner class.

nesting

The use of `Containers` within `Containers` to achieve sophisticated layouts or the use of conditionals within other conditionals.

networking

Computer-to-computer communication across a distance.

new

A keyword that signals the JVM to allocate space for an object.

new

The state of a thread once it has been instantiated.

no-arguments constructor

A constructor for a class having no parameters; the default constructor is an example.

not runnable

The state of a thread when it is removed from the runnable queue but is not dead.

object

A generic term for non-primitive data types; objects represent data and the operations on data.

object

An instance of a class, with unique data and operations.

OutputStream

The abstract class from which all other output byte streams are derived.

OutputStreamWriter

A utility stream that converts a character stream into a byte stream for output.

overloading

Two or more methods with the same name and different parameter types; they usually implement similar types of operations.

overridden method

A method in a derived class with the same method signature as that of a method in the superclass; it replaces the superclass method for the subclass.

package

A collection of classes designated by the `package` keyword; Java stores the classes in a directory structure based on the package.

paint()

An inherited method from `Container` that allows painting to the applet surface.

paintComponent

(Graphics g)

A method inherited from `JComponent` that is passed a reference to the graphics context of that component.

parameter

Information sent to a called method which includes data type and local name.

parameter tags

Optional name/value tags that send information to an applet from within the HTML page.

pass by reference

A pointer to the data is passed to a method; changes made to the data by a method are also made to the original data. Although Java always uses pass by value for primitive data types, pass by reference is used for non-primitive data types because the value for a non-primitive type is a reference.

pass by value

A copy of the value of the variable is available to the method; changes to the copy do not affect the original value; used by Java for primitive data types.

persistent

The ability to maintain a prior state.

polymorphism

Using one method name to invoke many different methods.

port

An integer from 0 to 65535 representing a particular path for information flow, a specific protocol is typically associated with a particular port; usually associated with networking.

pre-emptive multitasking

A system of multitasking in which one task can pre-empt another task if it has a higher priority.

primitive data types

The basic data building blocks of the language; all the primitives consist of a single value.

private

Access modifier that is applied to the member only; restricts access to the class in which it is defined.

protected

Access modifier that allows the class to be seen by classes in the same package or subclasses in other packages.

public

Access modifier that allows the class or member to be seen by every part of the program.

Reader

The abstract class from which all other input character streams are derived.

reference

Java stores the location of a non-primitive type in the variable; the actual memory for the type exists at the pointed-to-memory location.

relational operators

Operators that compare values of variables; all evaluate to the boolean data type (`false` or `true`).

repaint()

A method of `Component` that is used to indirectly call `paintComponent()` for a graphics update.

run()

The method that contains the body of a thread.

runnable

The state of a thread once it has been started.

***Runnable* interface**

An interface that prototypes the `run()` method, targeting it as a thread.

RuntimeException

A set of possible conditions that can occur in a program and cause an illegal operation.

sandbox

The strict security restrictions placed on applets.

scope

The range over which a variable is available for use.

SDK 1.2

The current release of SDK with a much improved event model.

SecurityManager

A Java class used by browsers to set security restrictions on an applet.

***Serializable* interface**

An interface that marks an object to be serialized.

serialization

The process of maintaining an object's state by converting it into a byte stream.

ServerSocket

A class in Java that represents the socket model for servers.

servlet

A small Java application that runs on a server.

setColor()

An overloaded method of the `Graphics` class that establishes the current color of the graphics context.

short-circuit operators

Operators that do not evaluate all the relational operations unless necessary.

signed applet

A digitally signed applet; if accepted by the user, security limitations are reduced.

Socket

A class in Java that represents the socket model for clients.

source

The `Component` that generates the event object.

stand-alone application

A Java program that need not be run in a browser.

start()

The second method called in the life cycle of an applet.

start()

The method that starts a thread by calling the `run()` method.

static

A keyword used to designate a class variable (and also a class method). Class methods and variables exist only one time, as opposed to instance methods and variables that exist for each object of the class.

stop()

A method that works in conjunction with `start()`; should be used to suspend resource-draining code.

stream

A path of information from a source to a destination.

StringBuffer

A peer class to the `String` class; it is not immutable.

strongly typed

Requiring that the programmer specify the kind of information that each variable can hold; used to describe a language.

super

The keyword used to refer to the superclass object; it allows you to call a method defined in the superclass.

super

A keyword referring to the superclass of the current class.

superclass/subclass

The superclass is extended by the subclass. The subclass is derived from the superclass.

Swing

A GUI toolkit used by Java.

***switch* statement**

A more complex conditional statement; chooses from multiple options.

synchronization

Controlling the flow of multiple, simultaneous threads.

synchronized

The keyword used to access an object's monitor.

System.in

Java's version of standard input (usually the keyboard).

System.out

Java's version of standard output (usually the monitor).

tag an object

Implementing a dummy interface on an object for the sole purpose of giving that object a type.

TCP/IP

The premier set of networking protocols used for the Internet.

this

A keyword that refers to the current object.

thread

A division of execution within a program. Each thread can be started and stopped and can wait for other threads to suspend or run concurrently with them.

thread racing

Competition between two or more threads for the same resource.

thread states

The various ways of describing the current state of a thread.

top-level container

A `Container` that cannot be contained in another `Container` as in a `JFrame`.

transient

A member marked as transient will not be serialized.

type

Identifies the kind of information that a variable can store.

user thread

A thread created by other user threads with the purpose of being controlled by the programmer.

widgets

A term used to refer to visual `Components` of a GUI.

WIDTH/HEIGHT

Essential information needed to embed an applet; determines the size of the applet.

WindowListener

One of many interfaces that allow an object to receive certain events.

wrapper

A class that wraps around other stream classes to add functionality.

Writer

The abstract class from which all other output character streams are derived.

Index