ZDU Student Manual

# JAVA PROGRAMMING:
## *PART 1*

# Java Programming: Part 1

ISBN: **0-73725-349-5**

Part number: **ZDU56705**

# ACKNOWLEDGMENTS

### Content Development

The content of this self-study guide is based on the training course "Java Programming," developed by Instruction Set, Inc. for its curriculum of instructor-led technical training. This guide was designed and developed by an Instruction Set team of instructional designers, course developers, and editors.

### Administration

Vice President and General Manager of ZD University: Ed Passarella

Marketing Director: Risa Edelstein

Director, ZD University: George Kane

Senior Editor, Curriculum: Jennifer Golden

Project Director, Instruction Set: Laurie Poklop

Project Manager, Instruction Set: Sandy Tranfaglia

### DISCLAIMER

While Ziff-Davis Education takes great care to ensure the accuracy and quality of these materials, all material is provided without any warranty whatsoever, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose.

Trademark Notices: ZD University and Ziff-Davis Education are trademarks and service marks of Ziff-Davis Inc. Java Programming: Part 1 is a Copyright of Instruction Set, Inc. All other product names and services used throughout this book are trademarks or registered trademarks of their respective companies. The product names and services are used throughout this book in editorial fashion only and for the benefit of such companies. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with the book.

Copyright © 1998 Instruction Set, Inc. All rights reserved. This publication, or any part thereof, may not be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, storage in an information retrieval system, or otherwise, without the prior written permission of Instruction Set, Inc., 16 Tech Circle, Natick, MA 01760, (508) 651-9085, (800) 874-6738. Instruction Set's World Wide Web site is located at http://www.InstructionSet.com. ZD University's World Wide Web site is located at http://www.zdu.com.

Photocopying any part of this book without the prior written consent of Instruction Set, Inc. is a violation of federal law. If you believe that Instruction Set materials are being photocopied without permission, please call 1-800-874-6738

CONTENTS

# Java Programming: Part 1

## LESSON 1: FUNDAMENTAL ASPECTS OF PROGRAMMING


Objectives . . . . . 2
What Is Programming? . . . . . 2
A Brief Overview of Software Development . . . . . 3
Conceptual Differences Between Software and Hardware . . . . . 4
Conceptual Differences Between the Developer and the User . . . . 4
Programming Languages . . . . . 4
Syntax vs. Semantics . . . . . 5
Types of Programming . . . . . 6
Console Versus Windows-Based Applications . . . . . 6
Procedural Programming . . . . . 7
Object-Oriented Programming . . . . . 7
Event-Driven Programming . . . . . 9
C/C++/Java Language Family . . . . . 9
The Common Language Core . . . . . 10
Comments . . . . . 10
Identifiers . . . . . 11
Data Types . . . . . 11
Expressions . . . . . 13
Input and Output . . . . . 13
Program Structure . . . . . 14
Functions . . . . . 15
Control Structures . . . . . 16
Scoping . . . . . 19
Lesson Summary . . . . . 21
Exercise . . . . . 22

## LESSON 2: THE JAVA ENVIRONMENT

Objectives . . . . . 28
Introduction . . . . . 28
Uses for Java . . . . . 29
The World Wide Web (WWW) . . . . . 30
HTML . . . . . 30
Competing Technologies . . . . . 31
Java Characteristics . . . . . 33
Java Portability . . . . . 33
Advantages and Disadvantages . . . . . 34
Applications vs. Applets . . . . . 34
Application/Applet Development . . . . . 35
Writing Source Code . . . . . 35
Compiling an Application . . . . . 37
Running the Application . . . . . 38
Tools and Packages . . . . . 41
The `javadoc` and `jdb` Tools . . . . . 41
Using `packages` . . . . . 41
Searching for Classes . . . . . 42
The `import` Statement . . . . . 43
Packages, Classes, Files, Directories . . . . . 44
Lesson Summary . . . . . 45
Review Questions . . . . . 46
Exercise . . . . . 47

## LESSON 3: JAVA BASICS

Objectives . . . . . 50
Language Basics . . . . . 50
Comments . . . . . 50
Data Types . . . . . 51
Declaring Variables . . . . . 53
Literals . . . . . 54
Expressions . . . . . 56
Java Operators . . . . . 56
Operations . . . . . 57
Precedence and Associativity . . . . . 61
Statements . . . . . 64
Control Structures . . . . . 65

Scope . . . . . 70
Lesson Summary . . . . . 73
Review Questions . . . . . 74
Exercise . . . . . 76

## LESSON 4: CLASSES IN JAVA

Objectives . . . . . 78
Java Is Object-Oriented . . . . . 78
Classes, Objects and Variables . . . . . 79
Instantiating a Class . . . . . 79
Class-Type Variables . . . . . 80
Operations on Class-Type Variables . . . . . 81
The `null` Value . . . . . 82
Member Access . . . . . 83
Invoking a Method . . . . . 83
Class Definitions . . . . . 84
Declaring Instance Variables . . . . . 86
Declaring Instance Methods . . . . . 87
The `this` Variable . . . . . 88
Class Definitions and Source Files . . . . . 89
Lesson Summary . . . . . 90
Review Questions . . . . . 91
Exercise . . . . . 92

## LESSON 5: CLASSES IN JAVA—II

Objectives . . . . . 94
Method Overloading . . . . . 94
Constructors . . . . . 95
Encapsulation . . . . . 96
Access Specifiers . . . . . 96
Comparing Objects . . . . . 97
Class Variables . . . . . 97
Class Initialization . . . . . 98
`final` Variables . . . . . 99
Class Methods . . . . . 100
Finalization . . . . . 101
Lesson Summary . . . . . 102

Review Questions . . . . . 103
Exercise . . . . . 104

## LESSON 6: ARRAYS AND STRINGS

Objectives . . . . . 106
Java Arrays . . . . . 106
Array Constants . . . . . 107
Using Arrays . . . . . 108
Copying Array Elements . . . . . 108
String Objects . . . . . 109
String Methods . . . . . 110
String Comparison . . . . . 110
String Searching . . . . . 112
Other String Methods . . . . . 113
String Concatenation . . . . . 113
Converting Objects to Strings . . . . . 114
Converting Strings to Numbers . . . . . 114
Lesson Summary . . . . . 116
Review Questions . . . . . 117
Exercise . . . . . 118

## LESSON 7: INHERITANCE

Objectives . . . . . 120
Introduction to Inheritance . . . . . 120
Example of Inheritance . . . . . 121
Derivation Syntax . . . . . 122
Effects of Inheritance . . . . . 122
Protected Access . . . . . 123
Overriding Methods . . . . . 124
Dynamic Method Dispatching . . . . . 125
Polymorphism . . . . . 126
The `super` Keyword . . . . . 127
`Final` Methods and `Final` Classes . . . . . 128
Constructor Chaining . . . . . 128
Inheritance and Finalization . . . . . 131
Abstract Classes . . . . . 131
Interfaces . . . . . 132

The `implements` Declaration . . . . . 133
Casting Between Class Types . . . . . 133
Lesson Summary . . . . . 135
Review Questions . . . . . 136
Exercise . . . . . 137

## LESSON 8: WRITING JAVA APPLETS

Objectives . . . . . 140
What Is an Applet? . . . . . 140
A "Hello, World" Applet . . . . . 140
The **`Applet`** Class . . . . . 141
Invoking an Applet . . . . . 142
Getting Applet Parameters . . . . . 143
Specifying Applet Parameters . . . . . 146
The Delegation Event Model—Action Events . . . . . 146
The `ActionEvent` Class . . . . . 148
Adjustment Events . . . . . 149
The **`paint()`** Method . . . . . 151
The **`Graphics`** Class . . . . . 151
Java Fonts . . . . . 152
Selecting a Font . . . . . 153
Drawing Lines and Shapes . . . . . 154
Drawing with Color . . . . . 155
The **`Color`** Class . . . . . 156
Foreground and Background Colors . . . . . 157
Lesson Summary . . . . . 159
Review Questions . . . . . 160
Exercise . . . . . 161

## APPENDIX A: HYPERTEXT MARKUP LANGUAGE (HTML)

HTML History and SGML . . . . . 164
Structure . . . . . 165
Head Elements . . . . . 165
Formatting: Blocks and Separators . . . . . 166
Formatting: Physical . . . . . 166
List . . . . . 167
Netscape List Extensions . . . . . 167

Links . . . . . 168
Images . . . . . 168
Forms . . . . . 169
Tables (HTML 3) . . . . . 170
Miscellaneous Netscape Extensions. . . . . 170
Java Applets . . . . . 171
Java Script . . . . . 171

## APPENDIX B: JAVA SAMPLE

Usage and Copyright Notification . . . . . 174
The XYZApp.java Source. . . . . 175

## APPENDIX C: JAVA CLASS HIERARCHY

Java Class Hierarchy . . . . . 186

## ANSWERS: TO REVIEW QUESTIONS

Lesson 2 . . . . . 196
Lesson 3 . . . . . 196
Lesson 4 . . . . . 197
Lesson 5 . . . . . 197
Lesson 6 . . . . . 197
Lesson 7 . . . . . 198
Lesson 8 . . . . . 198

# LESSON 1

# Fundamental Aspects of Programming

## OVERVIEW

*Learning to use a programming language effectively requires a good background in the principles of software development. The novice programmer must visualize the software development process as a whole, from requirements analysis through testing. The various types of programming and their implications must be understood clearly. To facilitate this, the several basic components of programming and programming languages are explicated in this lesson by applying them to an incrementally developed software example.*

## LESSON TOPICS

- What Is Programming?
- Types of Programming
- The Common Language Core
- Program Structure

## OBJECTIVES

By the end of this lesson, you should be able to:

- Explain the nature of software development.
- Differentiate between the different types of programming approaches.
- Understand the similarities and differences between C, C++ and Java.
- Edit, compile, and run simple console based applications.

## WHAT IS PROGRAMMING?

Before tackling Java programming, it is important to understand what programming is and how it is used. People program to create software, which can take many forms such as spreadsheet applications, Web applets, operating systems or corporate transaction databases.

Software consists of a series of instructions that tells hardware (or another piece of software) to perform one or more actions. The programming language is the set of syntax used to create those instructions and the rules associated with their usage.

Like other natural languages, programming languages allow for some freedom and creativity in how a piece of software achieves its purpose. However, unlike in spoken languages, programming must rigorously adhere to the rules and constructs of the programming language to achieve precise results. Machines cannot guess at meaning like humans can. Therefore, a good programmer must be very good at taking a problem or an idea in real world human terms, and abstracting it into the constructs provided by the programming language.

*Programming* generally refers to the coding portion of software development. Good software development, however, requires several other steps in addition to coding.

## A Brief Overview of Software Development

To develop an application, the developer normally follows a series of steps such as the following:

- Requirement Analysis
- Design
- Coding
- Testing

*Requirement analysis* is the process of determining the requirements of the application. Typical application requirements might be, “to support a spreadsheet-like interface” or “to calculate the precision of a result to 5 significant decimal places.”

The requirement analysis phase is perhaps the most important step in the development process. Application requirements need to be clearly understood and articulated if the software is to be a success. Proper requirements analysis will help avoid unnecessary corrective effort later in the process.

The *design phase* may occur at a high or low level. This step is responsible for the overall structure of the application and how the parts of the application interact.

*Coding* is the process of converting a design into a series of instructions through a programming language. Source code is written and then put through a code compiler, which translates the code in machine level instructions.

Before putting code into use, it needs to be tested to locate and remove errors. *Testing* checks the application against the requirements, tests the application functions with sample data, and verifies whether the user will find the application easy-to-use, among other steps. When the application behaves abnormally, corrections are made to the software and the testing process resumes. When testing is complete the application is ready to use.

## Conceptual Differences Between Software and Hardware

Software is a series of instructions to a machine. The computer takes the instructions and executes them to produce the intended results. Software can be readily changed. It can be modified to make changes in the behavior of the application. These changes are changes in the application itself.

The hardware that makes up the computer does not readily change. At the heart of the computer is a central processing unit that executes machine level instructions. These instructions are very low level and are not easy for the average programmer to use. The actual set of instructions for any language is fixed. However, they are recombined in different sequences that result in different behavior for different programs.

## Conceptual Differences Between the Developer and the User

The end user is the individual that executes an application for the purpose of performing a task. The developer is the individual that creates the application.

There are different types of developers. Some developers specialize in the requirement analysis or design phase of a project. There are others who primarily code and test an application. For small to medium size projects, one developer or a small group of developers may perform all of these tasks.

In addition, there are different types of applications that require different types of development expertise. Database intensive applications will require the skills of a developer proficient in the architecture and mechanics of databases, usually a specific database such as Oracle. Other developers will specialize in real-time applications that require carefully timed responses to system inputs.

## Programming Languages

A variety of different programming languages have evolved over time. Each of these languages has its own unique syntax and semantics. Languages are often categorized as either high level or low level, though this simple classification is not always adequate.

Most developers work with high-level languages such as COBOL, C or C++, Java, and FORTRAN. High-level languages do not use instructions that the machine understands directly; instead, they utilize commands that are more intelligible for humans. Naturally, high-level languages are easier to use. Few high-level language instructions are required to perform the same task in a low-level language. While programmers using a high-level language lose direct control of instructions to the machine, this is not a concern for most developers.

Certain high-level languages are specialized. For example, there are languages designed to work directly with databases. Others are targeted towards developing graphics applications.

Low-level languages have a one-to-one correspondence between a statement in the language and a machine instruction. These languages are normally called assembly languages and are unique to a specific computer. They provide a great deal of control over what can be done by a program but are very difficult and tedious to use.

Some languages rely upon an interpreter instead of a compiler. The interpreter is an application that takes high level instructions for a given language and simulates the execution of the instructions. The interpreted application will produce the same results as if it were compiled. Interpreted code executes slower than a compiled version of the application but interpreters provide several advantages over compilers, including better portability and more powerful development features. Java is an example of a language that is frequently interpreted rather than compiled. Any language may be compiled or interpreted if a compiler or interpreter has been written for that language.

## Syntax vs. Semantics

There are syntactic rules for each language that determine whether a given command is in the proper form. Like rules that govern the form of an English sentence, syntax rules of a programming language determine whether the program is valid or invalid.

When a program violates the syntax rules of the language, the compiler will generate an error message and will usually stop the compilation process. If no errors are generated, the program is said to have compiled cleanly. The program will run, but that does not necessarily mean that the program will do what it is intended to do. The program must order instructions in such a

way that they produce the correct result. In other words, the program must be *semantically* correct.

The semantics of an instruction is the meaning of the instruction i.e., what the instruction does. Understanding this is fundamental to learning programming languages. Fortunately, this is normally a straight-forward process. High-level programming languages are designed with logical and intuitive semantic structures.

# TYPES OF PROGRAMMING

There are several types of programming. This section, by no means exhaustive, will discuss some of the key types:

- Console-based
- Windows-based
- Procedural
- Object-oriented
- Event-driven

## Console Versus Windows-Based Applications

Console based applications use an MS-DOS type user interface (often called *character driven*) for input and output. The application is limited to character-based displays of information. Frequently, the order of data entry is dictated by the application. There are rarely windows of any nature present in these.

Windows programming use extensive amount of graphics programming to provide a more diverse range of input and output features. This graphics programming may be hidden from the application developer but is supported by various operating system libraries.

Windows programming offers the developer freedom to use different Graphical User Interface (GUI) controls to display or obtain information from the user. Text Boxes can be used for textual information, lists can be

displayed in List Boxes, and mutually exclusive options can be provided through Radio Buttons.

Console-based programming is easier than windows programming and does not use as many system resources. Windows programming is more difficult to learn and is resource intensive. However, given that most people expect windows based applications these days, creating windows is often an application requirement. The demand for windows has spurred significant advances in computer processing power and windows development tools.

## Procedural Programming

Procedural programming is used by almost all programming languages. Procedural programming focuses the step-by-step sequence of instructions that are needed to perform a task. This approach has been needed because applications and computers are not capable of automatically deciding the appropriate sequence to use.

Non-procedural programming is characterized by the developer specifying what needs to be done rather than how the task is to be performed. Procedural programming involves instructing the system as to how to achieve a task. While non-procedural languages exist, these languages are normally limited to a small application domain and are limited in the range of tasks that can perform.

Procedural programming is tricky for large applications. Large procedural programs can become unwieldy and difficult to manage. There is often little opportunity for abstraction. By abstracting a problem, it can be simplified. The interfaces between parts of an application can become difficult to manage also. Programmers tend to spend a lot of time learning how to program to these interfaces.

Most commonly used languages are procedural, including C, C++, Java and FORTRAN. These languages have evolved over time but they all have procedural aspects to them.

## Object-Oriented Programming

The evolution of object oriented programming brought about a paradigm shift in how application problem sets are modeled for development. In a traditional procedural language like C, things represented in a banking application like *account number*, *customer*, and *balance* dissolve into a series of

functions and variables. In object-oriented programming, objects are defined and maintained that represent these real things that are part of a problem statement.

Just as things in real life have characteristics and actions associated with them, objects possess *properties*, *methods*, and *events*. Each of these is directly associated with the object. For example, a bank account object would have the *properties* of account number and balance. These properties are tied directly to an instance of Bank Account, and do not exist in the program, other than as a part of Bank Account.

*Methods* are actions performed against the object. For example, the bank account object may have a deposit method that is used to add money to an account. *Events* are actions that take place under specific conditions. The bank account might have a negative balance event that would occur whenever the balance became negative. The programmer would code this event to take appropriate action when this event occurs.

Object-oriented languages support the definition of a class that represents an object and the actual allocation of memory for the object. Classes are the generalization of a specific object: Customer is a class, and Kelly Smith is an object that is an instance of the class Customer. C++ and Java are examples of object-oriented languages (note that they are also procedural languages). The coding used in support of methods and events is procedural.

Object-oriented languages offer a number of benefits. Systems modeled in objects are easier to maintain and manipulate, because properties and methods can be altered for a class without affecting the rest of the system. Object-oriented programming also supports reuse of code: once developed, a class can be utilized as often as necessary. For instance, a programmer could use an existing `Button` class in an application without necessarily understanding all of the code that supports the class. In addition, classes may inherit properties from more general classes, so that the properties do not have to be redefined for each sub class.

An *object-based* language, as opposed to an object-oriented one, is a language that supports the creation of classes but does *not* support inheritance, a crucial programming feature which facilitates the creation of related types from existing types. Visual Basic 5.0 and its earlier versions are an example of an object-based language.

## Event-Driven Programming

Event driven programming is frequently used to support Windows programming. The operating system will monitor the system for events and then send these events to the appropriate application. The application's reaction is based on the nature of the event. Keyboard input and mouse movements are the most common events.

An event-driven application is structured around the process of receiving and reacting to events. Unlike in procedural programming, where the programmer dictates the order of information received, event-driven applications generally allow the user to choose the order in which information is entered.

For example, an application may have an entry field for Zip Code. In an event driven application, the Zip Code field will be one in a series of address fields in a dialog box. The user can use the mouse or tab key to jump around to different fields and fill them in any order. The application's structure must account for the fact that it can not anticipate when Zip Code will be filled, but rather must react when the event occurs.

Object-oriented programming is well suited for event driven applications. Objects can have events assigned to them, which the application can monitor. Traditional procedural languages such as C, do not support event driven programming very well. Java, in particular, is designed to address events explicitly and has a well defined event model to create and handle events.

## C/C++/Java Language Family

C was developed to meet general system programming needs. It was quickly adapted for general use and found widespread acceptance. C is a high-level procedural language that has many low-level features. These features help to make it versatile and efficient. However, many of these features give it a reputation for being cryptic and hard to maintain.

C++ extends the C language to provide object-oriented features. The language is backward compatible with C, and code from the two languages can be used with each other with little difficulty. C++ has found quick acceptance and is supported by a number of pre-built specialized classes.

Java can be considered the third generation of the C/C++ family. It is not backward compatible with C/C++ but was designed to be very similar to these languages. The creators of Java intentionally left out some of the fea-

tures of C/C++ that have been problematic for programmers. Java is strongly object-oriented. In fact, one cannot create Java code that is not object-oriented. Java's portability is a key advantage and is the reason why Java is often used for Web development.

# THE COMMON LANGUAGE CORE

There are several programming features common to most programming languages. These include the definition of *identifiers*, *expressions*, and *condition control* statements. These common core features are discussed and illustrated below with sample code where appropriate.

## Comments

Comments are sections of a program that are ignored by the compiler and are used to document the program. Documentation is important in that it provides information about the program that is useful in the maintenance of the program.

Software applications generally require periodic revision—the application requirements change, errors are corrected or new features are added. Good documentation aids the programmer in identifying the structure and purpose of each section of code. Often, the person modifying the code is not the same person that wrote the code, making the use of good comments critical.

The only comment that is common to C, C++ and Java is the *multi-line* comment. The comment begins with a `/*` and ends with a `*/`. Everything between these two sets of symbols is treated as a comment and is ignored by the compiler.

```
/* This is a comment
    spread across several lines */

/* This comment occurs on a single line */

i = i + 1;  /* This comment begins on the same line as
a statement */
```

An alternative style available in C++ and Java uses two forward slashes (//) to indicate that everything from that point until the end of the line is a comment.

```
i = i + 1;  // Alternate comment notation
```

## Identifiers

Identifiers are used to temporarily hold information. For example, to compute the pay for an individual, the numbers of hours worked is multiplied by the pay rate. Identifiers (the name of the variable listed on the left) are needed for these values.

```
hours = 40;
payrate = 12.50;
pay = hours * payrate;
```

Identifiers have certain rules for their creation. Generally, the identifier must begin with a letter and is followed by letters, digits, and perhaps another character, such as the underscore character.

The assignment operator, which is the equal sign (=), is used to modify the value assigned to an identifier. Simply using the identifier as part of some operation will retrieve its current value. The value stored in an identifier can change.

## Data Types

There are different types of identifiers based on different needs. If simple counting numbers are needed then there is the integer data type. If a number with decimal points is required then the float data type is used. The common data types are listed in Table A.

TABLE A. *Data Types*

| Data Type | Value | Examples |
|---|---|---|
| int | Positive and negative integers | 127<br>-6 |
| float | Single precision numbers | 6.42 |

TABLE A. *Data Types (continued)*

| Data Type | Value | Examples |
|---|---|---|
| double | Double precision numbers (numbers with a very large value or # of digits) | 137.5426246321<br>5.42 x 1045 |
| char | Single characters | h |
| char* or String | String data (any series of characters) | widget<br>42XXTALL |

Note that very large numbers or very precise numbers use a different data type than normal real numbers. The boundary between single and double precision numbers is defined differently for each language.

String data is handled differently between C/C++ and Java. C and C++ use a pointer to the address of a string of characters that is terminated by a zero. Java uses a built-in data type *String* to hold strings.

Before an identifier can be used, it must be declared. Specifying the type of data followed by the identifier's name does this:

```
int    i;
float  pay;
int    hours;
double radius;
```

Most languages provide the developer with the ability to define more complex data structures based on primitive data types. C and C++ do this using the `struct` statement.

```
struct Employee {
       char*name;
       int age;
       double payrate;
};
```

An actual identifier is declared as other primitive types are declared.

```
struct Employee boss;
```

## Expressions

*Expressions* are needed to perform computations. An expression consists of an operator and one or more operands. Operators specify the action to be performed, such as addition or multiplication. Operands are the data acted upon. Statements are separated by semicolons and consist of one or more expressions. In the following example, the assignment operator, equal sign, and the addition operator are used to add two numbers together and assign the sum to a third identifier.

```
a = b + c;
```

There is a predefined order in which expressions are evaluated. This is known as the *order of precedence*. For example, addition always occurs before assignment (+ happens before =). There also exists *rules of associativity* that specify the order in which two expressions of equal precedence are to be evaluated. These rules can be overridden through the use of parentheses. For addition, evaluation is done left to right. The following two statements are equivalent:

```
a = b + c + d;
```

```
a = b + (c + d);
```

## Input and Output

Getting information into a program and then writing it out are important aspects of programming. Data can be read from a number of sources: keyboard, mouse, files, databases, etc. Data can be outputted to the monitor, to various files, and to a database, among other places.

The syntax of Input/Output (I/O) is mostly language specific. C and C++ rely upon `include` files that contain specialized code to perform I/O. Java uses a series of classes to do similar I/O actions. Input statements are not addressed here, however, output examples are provided.

In C, the `printf` statement is used. It has two major parts: a format specification that is enclosed with double quotes and a list of data values to be printed. The following shows printing out pay information:

```
printf("Hours worked: %f Payrate: %f  Pay: %f\n",
    hours, payrate, pay);
```

The `%f` fields within the format string are "place holders" for the output values to be printed. The text printed will be that which is inside of the double quotes, with the `%f` field replaced by the values stored in the variables in the parameter list. The three `%f` fields are replaced with the values stored in `hours`, *payrate* and *pay.* The `\n` is used to cause a carriage return and a new line to be printed. Output goes to standard output, which is normally the monitor. For floating values, a `%f` file is used. For other data types, different place holders are used.

In C++, insertion operators (`<<`) are used to display the same information.

```
cout << "Hours worked: " << hours << " Payrate: "
    <<payrate << " Pay: " << pay;
```

`cout` represents standard output. Java uses a `println` method to display output. This same line would be coded as:

```
System.out.println("Hours worked: " + hours + "
    Payrate: " + payrate + " Pay: " + Pay);
```

## PROGRAM STRUCTURE

The structure of a program is an important aspect of the overall design of the application. A program is typically structured as functions that logically partition the program into manageable and comprehensible units. A function is a block of code that has an optional return value, a name, and a set of *arguments.* Arguments are identifiers that are passed to the function.

## Functions

The first function executed when an application begins is the `main` function.

```
main () {
       /* Body of the function */
}
```

Other functions can be declared and then invoked as needed. The `ComputePay` function accepts two arguments (`hoursworked` and `rate`) and then returns the resulting pay.

```
float ComputePay(float hoursworked, float rate)
{
    float result;
    result = hoursworked * rate;
    return result;
}
```

```
main () {
    float hours;
    float payrate;
    hours = 40;
    payrate = 12.50;
    printf("The pay is: %f\n", ComputePay(hours,
payrate));
}
```

The identifiers for hours and payrate are declared in the `main` function. They are then *initialized* by assigning them values. The `printf` statement calls the `ComputePay` function and is passed a copy of `hours` and `payrate` to the `hoursworked` and `rate` identifiers respectively. These are used to compute the pay and are stored in the identifier result. The result is returned using the `return` statement, whose value is then used as part of the `printf` statement in the `main` function.

The `printf` statement is actually part of a library. Libraries contain different functions that programmers find useful. To use a function from a library, the library must be included within the program. With C and C++, this is

accomplished using the `#include` directive. For the previous program to compile cleanly, the following `#include` directive must be added to the top of the file:

```
#include <stdio.h>
```

The creation of a program is an abstract process. It requires an understanding of the nature of the problem to be solved, an understanding of how the problem can be solved, knowledge of the language used to implement a solution, and the creativity to put it all together.

A general approach to tackling a programming problem is:

- Determine what needs to be done.
- Determine how to do it (i.e., write the code).
- Determine where to place the code.

In the pay computation example, only the hours worked and payrate are utilized. The problem becomes more complex if other factors have to be considered, such as overtime and insurance deductions.

Creating a successful program for this example requires an understanding of both the syntax and semantics of the solution. Semantically, the proper process for computing pay must be known and understood. Syntactically, these procedures must be translated into the proper coding functions, and the I/O display capabilities of the language must be utilized properly.

Each of the code sequences developed must be placed in the right location in the program. Functions are used to break a program up into more manageable units. There is frequently more than one way to partition a program. The right way depends on a number of factors: size of the program, efficiency, portability, etc.

## Control Structures

For programs containing more than just the simplest logic, control statements are needed to control the flow of execution of the program. For example, *tests* may be needed to differentiate between hourly and salaried employees. *Loops* may be needed to process more than one employee at a time.

The `block` statement is a commonly used control statement. It logically groups related statements together. The `block` statement is necessary when the control statement's syntax expects a single statement but multiple statements are what are required. The `block` statement is a single statement that meets the syntax requirements and also allows multiple statements to be included at the same time. A `block` statement is nothing more than one or more statements included between an open curly brace ({) and a close curly brace (}).

```
{
        /* A block statement */
}
```

The `if` statement tests an expression. Based on a true or false evaluation of the expression, branches to one of two places.

**WARNING**

*It is possible to have one `if` statement nested within another `if` statement. When this happens, you should be careful about knowing which `if` matches up with each `else`. The `else` matches with the nearest `if`. When unsure, use statement blocks with `{}` to specify how the `else` should be matched.*

```
if (expression)
        statement; /* Branch here if true */
    else
        statement; /* Branch here if false */
```

The following illustrates testing for overtime:

```
    if (hours > 40)
        pay = 40 * payrate + (hours – 40) * 1.5 *
payrate;
    else
        pay = hours * payrate;
```

The `switch` statement is like a multiple-branch `if` statement. The expression tested is an integer and branches are made based on the integer's value. The `case` keyword is followed by a constant. If the value of the integer expression matches the constant, then the branch is made to that location. The `break` statement causes control to flow to the end of the `switch` statement; otherwise, it will flow through to the next `case` statement. The

`default` clause is optional and will catch anything that is missed by the previous `case` clauses .

```
switch (integer) {

    case 1:
        statement;
        break;
    case 2:
        statement;
        break;
    …
        default:
    statement;
    }
```

The following branches based on an age:

**WARNING**

*It is a common error to forget the `break` statement. This will result in the execution continuing on to the next statement instead of falling out of the `switch` statement.*

```
    int age;
        …

    switch (age) {

    case 1:
        printf(“The age is 1\n”);
        break;
    case 2:
    case 3:
        printf(“The age is either 1 or 2\n”);
        break;
    …
    default:
    printf(“The age is not 1, 2 or 3\n”);
    }
```

Looping statements cause a sequence of statements to be executed multiple times. This construct is useful for processing multiple lines of input, initializing identifiers, summing values, etc. The two most common looping statements are the `for` loop and the `while` loop.

The `for` loop uses an identifier as a counter and a test condition. It has three parts: *initialization*, *test* and *termination*. Each of these parts is separated by a semicolon. In the following example, the numbers from 1 to 10

are printed along with their square. The `i++` is a shorthand notation that is allowed to represent the statement, `i = i +1;`.

```
    for (i=1; i<=10; i++) {
printf("Number: %d  Square: %d\n", i, i * i);
    }
```

The `%d` field is used with integers.

The `while` statement has a test expression and a body. The expression is tested and as long as it evaluates to *true*, the body is executed. The following is the equivalent of the previous `for` loop.

```
    i = 1;
    while (i <= 10) {
    printf("Number: %d  Square: %d\n", i, i * i);
        i++;
    }
```

## Scoping

Scoping refers to the bounds of use of a particular identifier. Identifiers may only be used within the function in which they are declared. Consider the example again, in which the identifiers named `pay` and `rate` have been declared in both the `main` function and the `ComputePay` function.

```
    float ComputePay(float hours, float rate) {
        float pay;
        pay = hours * rate;
        return pay;
    }

main () {
        float hours;
        float payrate;
        hours = 40;
        payrate = 12.50;
        printf("The pay is: %f\n", ComputePay(hours,
payrate));
}
```

While the identifiers are given identical names in `main` and `ComputePay`, they should be distinct sets of identifiers. When an identifier is declared, it is allocated memory in main memory.

In the `main` function, memory is allocated for the identifiers `hours` and `payrate`. In the `ComputePay` function, memory is allocated to the identifiers `hours`, `rate` and `pay`. All five identifiers have separate areas of memory allocated to them. The values of the `hours` and `payrate` identifiers are passed to the `ComputePay` `hours` and `rate` identifiers respectively.

The scope of the `hours` and `payrate` identifiers is within the `main` function. That is the only place that these identifiers can be used, since they were declared within the `main` function.

The scope of the `ComputePay` `hours`, `rate`, and `pay` identifiers is within the `ComputePay` function, since they were declared within that function. Attempting to use the `pay` identifier within the `main` function would generate a compile-time error.

# LESSON SUMMARY

In this lesson you have learned that

- Programming refers to the coding portion of software development.
- Software development involves a series of steps—*requirement analysis*, *design*, *coding*, and *testing.*
- C, C++, Java, etc. are considered high-level languages as these require fewer instructions to perform a given task than low-level languages.
- For a program to be successful, it has to be both syntactically and semantically correct.
- Console based programming is character driven and is easier to use than Windows based programming which is resource intensive.
- Procedural programming specifies step-by-step sequential instruction.
- Object-oriented programming supports *objects* which represent real world concepts that are part of a problem statement.
- Windows programming is event driven programming—the common events being keyboard input and mouse movement.
- *Comments* are those sections of a program that the compiler ignores, but useful as signposts for programmers.
- `int`, `float`, `double`, `char`, and `String` are some of the data types that programmers use.
- *Expressions*, used to perform computations, consist of an operator and one or more operands.
- A *function* is a block of code that has an optional return value, a name, and a set of arguments.
- *Control statements* are needed to control the flow of execution of a program. The `block` statement is a common control statement.
- *Looping* causes a sequence of statements to be executed multiple times.
- *Scope* refers to the bounds of use for a particular identifier.

## EXERCISE

1. Type in the pay computation example as shown below and verify that it works properly.

```
public class Exercise1 {

    static float ComputePay(float hours, float rate) {
        float pay;
        pay = hours * rate;
        return pay;
    }

    public static void main (String argv[]) {
        float hours;
        float payrate;
        hours = 40f;
        payrate = 12.50f;
        System.out.println("The pay is: " +
ComputePay(hours, payrate));
    }
}
```

2. This exercise walks through the construction of a program that reads in specifications for a geometric figure, tests for valid input, and then computes and displays the area of that figure. You will need to add additional code to complete the exercise. Type in the first part as shown below. This section reads in a double number and is provided because Java does not read in numbers in as easy a manner as other languages.

```
import java.io.*;

public class Exercise2 {

static double getDouble() throws IOException{
    double  d = 0.0;
    byte buffer[] = new byte[8];
    String response;

    System.in.read(buffer);
    response = new String(buffer,0,buffer.length-2);
    try {
        d = Double.valueOf(response).doubleValue();
        return d;
    }
    catch (NumberFormatException e) {return d;};

}
```

The `main` method follows:

```
    public static void main (String argv[]) throws
IOException {

    /* Declare identifiers */

    byte buffer[] = new byte[8];

    /* Read in geometric shape type */

    System.out.print("Enter Geometric Type (r, t or c):
");
    System.in.read(buffer);

    /* Determine geometric type, get size information
and compute ares */
    /* Display area and perimeter */

    }
}
```

The identifie, `buffer[0]`, is used to hold the type of geometric figure selected by the user. The input statement, `System.in.read(buffer)`,

is used to read in data from standard input. Declaration will need to be added after the comments as explained below.

The three types of shapes to be processed are: *rectangle, triangle* and *circle*. Add declarations for the identifiers in Table B:

TABLE B. *Geometric Shape Identifiers*

| Identifier | Data Type |
|---|---|
| length | float |
| width | float |
| radius | float |
| area | float |

Next, add a `switch` statement that selects among the three options:

- r—Rectangle
- t—Triangle
- c—Circle

The basic structure of the `switch` statement will be similar to the following:

```
switch (buffer[0]) {
case (byte)'r': /* Rectangle selected */
    break;
case (byte)'t':     /* Triangle selected */
    break;
case (byte)'c':     /* Circle selected */
}
```

Within each `case` statement, add code to read in the appropriate values (height and width for a rectangle) and then compute the area. Use the following format to read in a `float` data type:

```
System.out.print("Enter Height:");
height = getDouble();
```

Next, add statements to verify that the values entered by the user are positive, non-zero numbers. If they do not meet this criteria, use the value `1.0` instead. The else part of the statement can be a single semicolon.

```
if (height <= 0)
    height = 1.0;
else
    ;
```

In the last section of the program, display the area of the geometric figure.

# LESSON 2

## The Java Environment

### OVERVIEW

*Java as an object-oriented language was designed from the outset to be platform independent, robust, and secure. Many of the concepts and syntax of Java are borrowed from C++, another object-oriented language. Java can be used to produce two types of programs. One type is an application, which is a standalone program that can be run by the Java interpreter. The other type is an applet, which is a mini-program that is typically built into World Wide Web pages.*

### LESSON TOPICS

- Introduction
- Java Characteristics
- Application/Applet Development
- Tools and Packages

## OBJECTIVES

By the end of this lesson, you should be able to:

- Describe the characteristics of Java environment.
- Edit, compile, and run Java programs and applets.
- Contrast Java with other programming languages.

## INTRODUCTION

Java (originally called Oak, until it was discovered that another company used Oak as a trademark) was developed at Sun MicroSystems in a forward-looking project to develop a new programming language. The primary design goal of Java was to provide code for programs that would run on many different hardware platforms. The second design goal was to produce robust programs—programs that would have as few bugs as possible. The third design goal was for the language to be easy to learn and use. Since the world of computers and application development was already populated with programmers who knew C++, the syntactical model of C++ was chosen along with some characteristics of *Smalltalk*. The effectiveness of Java as a programming language can be illustrated by considering how Java reduces bugs:

- The most common source of difficult-to-track bugs in traditional programming languages is improper handling of dynamically allocated memory. These *memory leaks* are caused by programmers allocating memory, and then losing track of its location or simply not releasing it. Java handles all memory management issues on its own, rather than leaving them to the programmer. Java allocates memory when an object is instantiated and releases it automatically when the object goes out of existence, through a process called *garbage collection*.
- Another frequent source of problems is out-of-range array indexing. C and C++ do not check for these errors, but Java does.
- Java avoids the syntactical headaches that result from multiple inheritance, which is an object oriented feature (of controversial value) in C++. Java uses interface files to provide most of the benefits of multiple inheritance (and polymorphism) while avoiding their problems.

## Uses for Java

Though it was originally intended as a language for programming embedded systems, Java has become very popular because of its natural applicability to programming network software for the World Wide Web (WWW). Java is best known as a language for programming *applets*. An applet is a "mini-application" that runs within the context of a larger application, such as a network browser.

For a program to be usable across such a diverse network as the World Wide Web, it must be portable; i.e., it must be able to run on many different systems. Java is designed to be portable.

Any software downloaded across a network must be considered untrustworthy software, suspect for viruses and other malicious programming. To overcome this liability, Java language and Java API (Application Program Interface) have built-in security features that provide safety from such attacks.

The structure of a Java program is modular. Instead of being bound into a single file, a Java application exists as a set of cooperating files, each file representing a discrete module of the program. This has a great advantage. Rather than downloading an entire program before running it, the user may download only those Java modules that are needed. To upgrade a Java program, the user may replace only those modules that have changed. This dynamic nature of the language also speeds application development.

Finally, the Java API includes a wide set of facilities for network communications. The Java programmer can readily create network connections and either work at the data stream level or at the higher network protocol level. The programmer can create network applets, agents, and servers in Java. All these features make Java an ideal platform for network software development.

It is important to note that though Java is ideally suited for the creation of network software, it is not limited to network software applications. The Java API also includes facilities for graphical user interfaces (GUI's), file access, parsing, and is rapidly expanding to include database access, remote procedure calls, component software, and other features. Java is a platform for general purpose application development.

## The World Wide Web (WWW)

Java is an outstanding language for developing applications that will run on the WWW. The WWW, which uses a client server framework, is a virtual network built on top of the Internet.

- The client is called a *browser*, which requests data from servers. Currently, the most popular browsers are Netscape Navigator and Microsoft Internet Explorer.
- The server is called a *Web server*. A Web server responds to requests by sending Hypertext Markup Language (HTML) documents back to the browser which are then displayed for the user.

Web servers can send many different file types (text, graphics, audio, etc.) to requesting browsers as part of an HTML document. Web servers typically run on systems using Unix or Windows NT. Browsers can request information from any Web server.

Web servers and browsers use the HyperText Transfer Protocol (HTTP) to communicate with one another. This protocol enables a browser to request data or other resources from a Web server. HTTP also enables a server to describe the file types that it can send to a browser. Although an understanding of HTTP may be of interest to a programmer, it is not necessary prerequisite for writing Java applications.

## HTML

Web servers send HTML documents to browsers. In other words, the data that servers provide to browsers is always coded in the HTML language. Browsers know how to display HTML documents that describe words, graphics and audio.

HTML uses a different model for describing the layout of information than most popular word processors do. Word processors like Microsoft Word or FrameMaker give the writer complete control over the way text is laid out. HTML gives the writer only partial control, turning the remainder of control over to the browser. For example, when putting a second level header into a FrameMaker template, the writer can control the font to be used, the exact position in which the header will be written, and dozens of other

details. By contrast, to put a second level header into an HTML document, the writer merely writes:

```
<H2>The Header Itself</H2>
```

and leaves the exact formatting details (font, position, etc.) to the browser. The current "official" HTML standard is Version 3.2. It is maintained by the WWW Consortium.

HTML is a rapidly evolving language. Unfortunately, not all browsers support the same HTML tags. Netscape, for example, continues to supplement standard HTML with many extensions of its own.

HTML tags describe not only text but graphic and audio information as well. For instance, the following HTML directive tells the browser to display a graphics image file named `logo.gif`:

```
<img src="logo.gif">
```

HTML tags also describe the hierarchical organization of the document for example:

- `<HTML>` - HTML document tag
- `<HEAD>` - Head section tag
- `<BODY>` - Body section tag

Finally, HTML also contains directives that tell the browser to run a particular Java applet. An overview of HTML tags can be found in *Appendix A*.

## Competing Technologies

Just a few years ago, C++ was hailed as the computer language that would revolutionize computer programming and lead to bug-free programs. Unfortunately, C++ was rather hard to learn and ended up creating many bug-filled programs. Nevertheless, although C++ was hardly the first object-oriented language, it did expose many programmers to object-oriented programming. Many programmers now feel that while the future belongs to object-oriented programming, C++ is not the best vehicle for it. Java's designers took the best features of C++ while rejecting its bad features.

Languages such as *Visual Basic* do a terrific job designing GUIs. Yet, Visual Basic lacks many important Java features that are needed for WWW applications. Microsoft has introduced ActiveX technology which can be used with *VBScript* (a subset of Visual Basic) to provide many of the same user interaction features as Java applets. VBScripts are embedded in HTML documents between `<SCRIPT>` and `</SCRIPT>` tags.

In spite of its name, JavaScript has nothing to do with Java. JavaScript is an object-based language which was developed at Netscape (a browser vendor). It is used to give dynamic capabilities to HTML documents, including client-side user interaction. JavaScript programs are not compiled, but are interpreted as part of the HTML document. JavaScript source code is embedded in an HTML document between the `<SCRIPT>` and `</SCRIPT>` tags. JavaScript can do many of the things that Java can do. However, JavaScript does not support the rich set of data types that regular Java does.

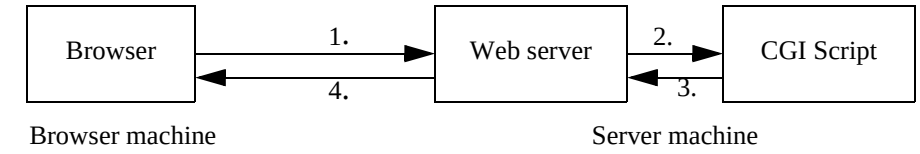
### *Common Gateway Interface (CGI) vs. Java*

The Common Gateway Interface (CGI) is a specification that defines how a browser sends data to Web servers. It is intended for use by separate server-side programs. *CGI scripts* are programs written to conform to this specification. That is, CGI scripts are external programs, invoked by a Web server, that receive information from the server and pass information back to it.

CGI scripts are usually written in an interpreted language such as Perl or Visual Basic, although they can be written in other languages such as C or C++. CGI scripts can do both simple tasks, like printing the number of users who have logged onto a particular Web site and more complex processing which involves access to sophisticated databases.

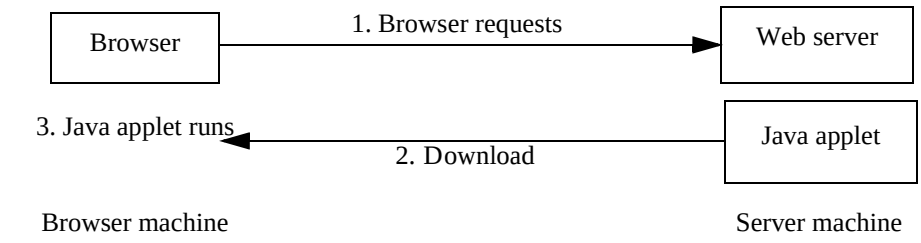
Programmers use CGI scripts to create dynamic HTML documents with server side processing. The Web server defines a set of environment variables that CGI scripts use to receive data from the browser. The data is then processed and a response sent back to the browser in the form of an HTML document, as shown in Figure 1.

FIGURE 1
*Interactivity with CGI scripts*




Java applets are stored on the Web server's machine but run on the browser's machine. Therefore, the Web server must download the Java applet to the browser. Figure 2 illustrates this.

FIGURE 2
*Interactivity with Java*




# JAVA CHARACTERISTICS

## Java Portability

A programming language is said to be *portable* if source code written in that language is readily compiled and run on a platform other than the originally intended target platform. The C programming language is portable in this sense, though most applications written in C rely on interfaces specific to a certain operating system or windowing system. The result of compiling C source code is machine code that runs on a particular type of processor. Thus, when we say C is portable, we refer to the source code, not the compiled binary code.

The Java system is not merely source code portable—it is binary portable. The result of compiling Java source code is Java byte code that can be run only by a Java Virtual Machine (JVM). The JVM is usually an interpreter, a program that executes the byte code instructions as it reads them. The JVM is source code portable; that is, designed to run on many different platforms.

Every JVM on every platform conforms to the same specifications. So any compiled Java program can run on any platform on which a JVM is available. Once the JVM has been ported to a new system, any pure Java program can be run on that system.

The Java language and JVM are standard and portable. The Java API, a set of libraries that accompanies every Java installation, is also standard and portable. When the JVM is ported to a new system, so are the standard Java libraries. This further guarantees that any compiled Java program runs on any platform on which a JVM is available.

## Advantages and Disadvantages

Java programs are generally more robust than C++ programs because Java's memory management is more efficient and cleaner. This also makes Java programs more secure. But because Java is interpreted, Java programs tend to run more slowly than C++ programs. Compared with other scripting languages such as Perl and Tcl, Java is faster, more powerful, and more robust.

If Java terminals become more commonplace, most of the speed disadvantages that they now suffer from may disappear. Java terminals may well become like X-terminals, (inexpensive machines with minimal local storage, retrieving class software as needed). Java terminals fall under the nebulous category of Network Computer (NC). As more and more quality just-in-time compilers become available for PCs, it is possible that an NC may not really possess any speed advantage.

## Applications vs. Applets

There are two categories of Java programs:

- Java applications (which run with a *Java interpreter*)
- Java applets

*Java applications* are stand alone programs in the traditional sense. *Java applets* are programs designed to be run from browsers such as Netscape Navigator and Sun's HotJava. The focus of an applet is the WWW, although several other tools (such as `appletviewer`) can also run Java applets. Applets are normally embedded within HTML documents.

An applet runs in a context that provides windowing support, multi media support (graphics, sound, etc.) and network support.

Java application source code looks somewhat different from Java applet source code. For example, Java programmers must write a method named `main` for every application, but may never do so for an applet. Despite the differences, it is fairly easy to convert Java application source code into Java applet source code.

# APPLICATION/APPLET DEVELOPMENT

*Steps to create a Java application:*

1. Using any text editor, write Java application source code.
2. Compile the Java source code. When this is done, the Java compiler creates a byte-code file.
3. Invoke the Java interpreter. It will run the byte-code file.
4. Debug as necessary, by invoking the Java debugger (jdb).

*Steps to create a Java applet:*

1. Using any text editor, write Java applet source code.
2. Compile the Java applet source code. The compiler creates a byte-code file.
3. Create an HTML document to contain the applet.
4. Invoke the byte-code file by specifying it in an HTML document and then loading the HTML document.
5. Debug as necessary.

## Writing Source Code

Writing Java source code is similar to writing source code in any other language. That is, an ASCII text editor is used to either add source code or modify it. Java source code is dealt with in greater detail in subsequent les-

sons. This lesson discusses two important features common to all Java applications discussed:

- As in C++, Java application source code is organized into classes. For example, the class in the following example is named `HelloWorld`.
- Every Java application must contain a method named `main`. As in a C program, the `main` method marks the application's starting point.

Following is a simple Java application. The source code would be entered into a file named `HelloWorld.java`:

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

The file containing Java source code must have the filename *`classname`*`.java`, where *classname* is the name of the class in your Java source code.

An ASCII text editor is used to write Java applet source code, as well. Writing applet source code is discussed in detail later on. A few common features of all Java applets are discussed below:

- As in C++ and Java applications, Java applet source code is organized into classes. For example, the `class` in the example below is named `HiWorld`.
- Unlike Java applications, Java applets must not contain a method named `main`. This is because a "behind-the-scenes" code already contains the `main` method.
- Most Java applets contain a method named `paint`. This method defines the actual appearance of the text and graphics that the Java applet will output.

**NOTE**

*The `paint` method will be automatically called by the Java environment when the applet needs to be drawn.*

Below is the applet version of `HelloWorld`:

```
public class HiWorld
extends java.applet.Applet
{
    public void paint(java.awt.Graphics g)
    {
      g.drawString("Hi, World!",50,25);
    }
}
```

## Compiling an Application

To compile source code from a shell prompt, run the compiler with:

```
javac name.java
```

The Java compiler (`javac`) will create a byte-code file named:

```
name.class
```

If a Java development environment is being used, the program is compiled by invoking a menu function rather than by running Java directly. The folder containing `javac` must be in the programmer's `PATH` environment variable.

To compile `HelloWorld.java`, type:

```
javac HelloWorld.java
```

This creates a file named `HelloWorld.class.`

If the source code file contains more than one class, the Java compiler will create a separate byte-code file for each class

**NOTE**

*Unlike traditional compilers, `javac` creates a machine-independent file as the output which gets interpreted by the Java virtual machine. Thus the contents of `name.class` will be the same whether you compiled in Unix, Windows, or any other OS.*

For example, given a source code file containing three classes, the Java compiler will generate three byte-code files, named:

- class1-name.class
- class2-name.class
- class3-name.class

Java applet source code is compiled the same way Java application source code is compiled.

To compile the source code from a shell prompt, the compiler has to be run with:

```
javac name.java
```

The Java compiler (`javac`) will create a byte-code file named:

```
name.class
```

If a Java development environment is being used, the applet can be compiled by invoking a menu function rather than by running Java directly.

## Running the Application

A Java application is basically run by invoking the Java interpreter on Java byte-code.

Java byte-code files have the filename *`name`*`.class`, where *`name`* is both the filename and the name of the code module it contains.

To run the program from a shell prompt, type:

```
java name
```

Note that no suffix is used. The interpreter will assume the suffix `.class` on the filename.

Remember that one of the goals of Java is to create programs that can run on any platform. Therefore, the Java compiler generates platform independent byte-code rather than platform dependent machine code. Then, to run a

program, the Java interpreter translates this byte-code into the machine code for whatever system the Java interpreter is on.

For example, Java source code is written and compiled on a Sun workstation running Solaris. If a Java interpreter running on Solaris invokes the byte-code, the Java interpreter will translate the byte-code into Sun machine code. Now, that same byte-code is copied over to a PC running Windows95. In this scenario, the Windows95 Java interpreter translates the byte-code into PC machine code.

Because of the time the Java interpreter needs to translate byte-code into machine code, Java applications run somewhat slower than applications stored in native machine code. The programmer's PATH must contain the directory where the interpreter `java` is stored.

The programmer may also need to add the directory containing his/her class files to the CLASSPATH environment variable.

**TIP**

*To pass a parameter that includes a white space, include the parameter in quotes, i.e. `java MyWork "This is just one parameter"`*

Parameters can be passed to a Java application when it is invoked. For example, to invoke a Java application named `MyWork` and to pass it two parameters (`42` and `/contacts/sales.txt`), the following would be typed:

```
java MyWork 42 /contacts/sales.txt
```

If the programmer is using a Java development environment, then the application can be run by invoking a menu function.

### *Running Java Applets*

Java applets are not run the same way as Java applications are run. To run a Java applet, the programmer must first insert the HTML tag <APPLET> into an HTML document. The <APPLET> tag must have the following form:

```
<APPLET CODE="name-of-byte-code-file" WIDTH=m
HEIGHT=n>
```

WIDTH and HEIGHT designate the size of the applet window (in pixels). The CODE, WIDTH, and HEIGHT fields are required in every APPLET tag.

For example, the following directive has to be placed inside an HTML document in order to invoke the `HiWorld` applet:

```
<APPLET CODE="HiWorld.class"
        WIDTH=150 HEIGHT=75>
</APPLET>
```

If the programmer is using a Java development environment, the environment itself may create the HTML document for them.

### *Viewing HTML Documents with Embedded Applets*

After creating the HTML document, the programmer needs to load it into a tool that can handle Java applets. The most common choice is a Java-enabled browser such as Netscape 2.0 (or a later version). The HTML document is loaded as one would load any HTML document from a browser.

1. Specify the full URL:

```
http://www.me.com/homepage.html
```

2. specify the pathname on the local disk:

```
homepage.html
```

Note that users with Java-enabled browsers can *disable* Java, preventing applets from running on their machines.

`HotJava` and `appletviewer` can also run and load HTML documents containing the `<APPLET>` tag.

To invoke `appletviewer` from a command prompt, type `appletviewer` followed by the name of the HTML file. For example:

```
appletviewer homepage.html
```

# TOOLS AND PACKAGES

## The **javadoc** and **jdb** Tools

Java provides an automatic documentation tool that can extract comments embedded into source files. This tool is called `javadoc`.

```
javadoc filenames
```

`javadoc` searches in source files for comments that begin with the symbols `/**`, then writes them out into HTML files. If the files contain no comments like this, then the tool cannot help. Therefore files created by programmers have to be commented extensively.

The Java Development Kit also includes the debugging tool `jdb`. This is a command line debugger similar to Unix `dbx`.

## Using **packages**

A `package` is a collection of classes. Java programmers wrap groups of related classes into packages. For example, a Java programmer might create a package called `geography` consisting of classes pertaining to geography.

To create a package, the programmer has to specify a `package` directive as the first statement in each source code file that will go into the package. For example:

```
package geography;
```

The primary purpose of packaging classes is to ensure a truly unique name for every class. For example, suppose that two programmers independently create two different classes both named `mapping`. If a program accesses `mapping`, it will not know which of the two classes to access. To eliminate this kind of ambiguity, the programmers can put their `mapping` classes into packages. If one programmer puts his `mapping` class into a `geography` package and the other programmer puts her mapping class into a `meteo-`

rology package, then the application programmers can be sure they are getting the geography mapping class by specifying:

```
geography.mapping
```

and the meteorology mapping class by specifying:

```
meteorology.mapping
```

The recommended names for package statements usually start with the reversed name of the programmer's Internet domain. By specifying the Internet domain name as a preface, the programmer makes the class name truly unique across the entire Internet. For example, suppose that two programmers create a mapping class in a geography package. If each programmer prefaces the package name with a domain name, the programmers make the class names even more unique. For example:

- `package edu.purdue.cs.geography`
- `package edu.mit.geo.geography`

## Searching for Classes

Java searches for classes in the following starting directories:

1. The current directory.
2. $JAVA/classes (where $JAVA is the parent location of the Java compiler, interpreter and other components).
3. $JAVA/classes.zip. This is a .zip file rather than a directory. Java knows how to extract class library components from a .zip file.
4. Any additional directories assigned to the environment variable CLASSPATH. For example, to add a directory to the existing class path (DOS), enter:

```
set classpath=%classpath%;c:\mywork
```

"System" directories are automatically appended to this list.

The entries specified can either be directories that contain classes or `.zip` files that contain classes. (Windows directories use \ instead of /.)

An alternative to assigning extra directories to `CLASSPATH`, is to specify a directory on the command line when running the compiler or interpreter. (To do this, use the option `-classpath` *`directory-name`*.)

## The `import` Statement

**TIP**

*`import` is especially useful with long package names with several dots to denote a certain class.*

Listing a long package name for every class used in a program can be tedious. The `import` statement may be used to reduce typing.

The `import` statement of Java is often confused with the `#include` directive of C. But the two are quite different. In fact, `import` does not include any code, it simply reduces the amount of typing a programmer has to do in order to access a class.

`import` has several forms. The following form allows a class to be called by a short name:

```
import package.class;
```

For example:

```
import geography.mapping;
```

means that wherever `mapping` appears in the program, the Java compiler will infer:

```
geography.mapping;
```

The following form allows *all* the classes in a package to be called by a short name:

```
import package.*;
```

The package `java.lang` is imported automatically by every program. The programmer need not specify:

```
import java.lang.*;
```

## Packages, Classes, Files, Directories

Java programs are assumed to be developed as *packages*. All of the files for a given package are required to be stored in a disk directory with the same name as the package.

A *fully qualified name* for a class, the data, and the methods they contain, includes the package name as a prefix with components separated by periods. Since directories on a given machine have unique names, this means a package and its components can always be distinguished from other packages.

For example, the fully qualified name of the method `snork()` includes the *package* name. As shown below:

```
mypackage.pictures.MyClass.snork()
```

This method `snork()` must be stored in a directory/file structure:

- on a Unix system as:

```
.../mypackage/pictures/MyClass.class
```

- on a Windows system as:

```
...\mypackage\pictures\MyClass.class
```

The compiler option `javac -d` *`directoryname`* can be used to tell the compiler where to store `.class` files that it produces.

# LESSON SUMMARY

In this lesson, you have learned:

- Java is purely object-oriented.
- Java borrows C++ syntax, but avoids many of C++'s problem areas.
- Java produces programs that are robust and secure.
- The interpreter verifies all variable and memory access.
- Java applets are ideal for the World Wide Web.
- The `package` statement clarifies class name space and provides a way to categorize related classes.
- The `import` statement reduces typing.

## REVIEW QUESTIONS

1. What command invokes the Java compiler from the command line?

2. What program usually runs a Java applet? Name another program that can run a Java applet.

3. To access a Java applet, you have to place what HTML directive inside an HTML document?

4. True or False: When writing a Java applet, the programmer must create a `main` method.

5. True or False: The `import` statement of Java is analogous to the `#include` statement of C.

*Answers on page 196*

## EXERCISE

1. Go to the `javasoln` directory in the on line materials.

   Compile the Java source file named `FirstApp.java` to create `FirstApp.class`.

   Set the environment variable USER to your name.

   Run the `FirstApp program`.

   Run the program again, providing the names of a few of your friends on the command line.

2. Go to the `pkg` subdirectory of the `javasoln` directory.

   Look for a Java source file named `SecondApp.java` there.

   Try to compile the source file to create `SecondApp.class`.

   What error does the compiler report? Edit the source file to correct the error.

3. Go back to the `javasoln` directory.

   Compile the source file named `FirstApplet.java` to create `FirstApplet.class`.

   Run the `FirstApplet` applet by invoking the applet viewer on the HTML file `FirstApplet.html`.

# LESSON 3

## Java Basics

### OVERVIEW

*Java syntax is based on those of C and C++. As a result, the basic methods of declaration, expression evaluation, flow control, and commenting in Java are virtually identical with those of C and C++. Java does include some enhancements though, such as improvements to boolean operations. This lesson gives an overview of these basic methods and improvements.*

### LESSON TOPICS

- Language Basics
- Expressions
- Statements

# OBJECTIVES

By the end of this lesson, you should be able to:

- Read and understand:
  - expressions,
  - statements, and
  - control structures written in Java.
- Write simple Java programs that work with variables of native data types.

# LANGUAGE BASICS

## Comments

*Comments* are blocks of readable text inserted into code for the benefit of other programmers; both those who must use the code and those who must modify it. Comments are ignored by the compiler, so they need not adhere to any particular syntax.

Java recognizes traditional C-style comments. Traditional C-style comments are delimited by `/* ... */` and can span more than one line. C compilers ignore all text after the opening `/*` up until a subsequent `*/`, which means that comments cannot be nested. (Some C++ compilers detect nested C-style comments and warn the user about them.)

Java also recognizes C++-style comments. In C++ comments are written starting with two slashes, like this: `// ...`, and run up to the end of that line of text. They cannot span more than one line: in a multi-line comment, each line must start with `//`. Comments like this are recommended because they can be nested inside C-style comments.

Java also extends the traditional C-style comment by starting with `/**`. This would be recognized as a conventional comment. In addition, a comment beginning with `/**` provides a documentation tool that can extract comments from source code to create external documentation. When a pro-

grammer embeds comments like this in code, that programmer is simultaneously writing its documentation.

Java has a special comment form which produces online documentation and is read by a `javadoc`, too.

```
/** This is a “doc” comment */
```

## Data Types

Java supports the *native* data types listed in Table C. Native data types are those provided automatically by the compiler in contrast to those that are defined as classes by programmers.

One of the major obstacles to cross platform porting of applications has been that there are very few standards for storage schemes. Every machine and every CPU has its own “word” size and its own storage format for basic data types. Java defines the sizes and formats of each of these native data types, and leaves it up to the interpreter to implement these sizes and formats on the local platform. This makes code fully portable.

TABLE C. *Native Data Types*

| Type | Contains | Default value | Size (bits) | Min and Max values |
|---|---|---|---|---|
| `boolean` | `true` or `false` | false | 1 | Not Applicable |
| `char` | Unicode character | \u0000 | 16 | \u0000 to \uFFFF |
| `byte` | signed integer | 0 | 8 | -128 to 127 |
| `short` | signed integer | 0 | 16 | -32768 to 32767 |
| `int` | signed integer | 0 | 32 | -2147483648 to 2147483647 |
| `long` | signed integer | - | 64 | -9223372036854775808 to 9223372036854775807 |
| `float` | IEEE754 flt. pt. | 0.0 | 32 | +/-3.40282347E+38 to +/-140239846E-45 |
| | IEEE754 flt. pt. | 0.0 | 64 | +/-179769313486231570E+308 to +/-4.94065645841246544E-324 |

### *The Boolean Type*

Most of Java's native data types are recognizable to C programmers, but boolean is new, inherited from more strongly-typed languages like Pascal.

boolean data contains only the specific values `true` or `false` and is the result of logical operations (equal to, less than, etc.) Java does not permit free conversion between boolean and other types. Wherever a boolean value is expected, as the condition in an `if` statement for example, the programmer must provide a value of boolean type. This may be surprising to C and C++ programmers who are accustomed to using integers or even pointers in place of true/false values.

### *Integer Types*

The Java type `int` represents a signed 32 bit integer. There are no unsigned integers in Java as there are in C and C++. Java does not recognize the C keyword `unsigned`. The types `long`, `short` and `byte` are integer types that differ from plain integers in the amount of storage space they require. In Java, the keywords `long` and `short` represent complete types. They are not declaration modifiers as in C.

The Java `char` type requires two bytes and not one byte as in C/C++. This allows Java to support the *Unicode* standard for international (non-Latin) character sets. To use single-byte data, use type `byte`. Java characters behave as *unsigned* integers.

### *Floating Point Types*

Java provides types `float` and `double` in IEEE754 format. These types may contain the expected floating point format values in addition to four special values:

- positive infinity
- negative infinity
- negative zero
- not-a-number (NaN)

These values are not normally assignable to variables, but could be obtained by unexpected arithmetic such as divide-by-zero, divide-by-negative-zero, etc. They differ in size and precision and may be combined in expressions.

Positive infinity, negative infinity and negative zero operations result in special floating point values. A not-a-number (NaN) is unordered (neither < nor > anything). It yields `false` when compared with anything, even itself.

## Declaring Variables

For variable declarations, Java syntax is similar to C and C++ syntax. For native types, the syntax may be summarized as follows:

- a one word type name
- a list of variable names, separated by commas
- an optional initializer following each variable name
- a terminating semicolon

The following are examples of variable declarations:

- This line of Java code declares a 32-bit integer variable named `i`:

```
int i;
```

- This line declares two 32-bit integer variables named `i` and `j`:

```
int i, j;
```

- This line declares `i` and `j` as above and initializes them:

```
int i = 3, j = 4;
```

It should be noted that `long` and `short` are data types unto themselves, not modifiers as in C:

```
short int i;// Not OK in Java
```

### *Identifiers*

Every named program component—class, package, function or variable—must have a legal *identifier* (or `name`). An identifier must contain only

alphanumeric characters and must begin with either an alphabetic character, a $, or an underscore. The following are legal identifiers:

- `dummy`
- `box$`
- `a37`
- `post_code`

The following are *not* legal identifiers:

- `rain@noon`
- `2by4`
- `post code`

**NOTE**

*Identifiers that begin with an uppercase letter are normally used for user-defined types, i.e. classes.*

The compiler distinguishes between uppercase and lowercase characters. For example, `abc`, `Abc`, `aBC` and `aBc` are all different identifiers.

By convention, variables begin with lowercase. Case can be a useful tool for making programs more readable as in: `areaOfCircle`.

The only other restriction is that keywords defined by the language cannot be used as identifiers—for instance, one may not call a variable an `int`.

## Literals

A *literal* is a program element that represents an exact value of a certain type. We have seen examples of integer literals. Here is another:

```
int i = 0xff;// a hexadecimal integer literal
int j = 0723;// a octal integer literal
int k = 123;// a decimal integer literal
```

### *Integer Literals*

Unless otherwise specified, an *integer literal* is normally of type `int` and 32 bits wide. Specifying an integer literal that requires more than 32 bits of storage, results in an overflow error at compile time; unless the integer literal

is of type `long`. To specify an integer literal of type `long`, append an `L` (either upper or lower case) to the number. For example:

```
long l1 = 5000000000L;
long l2 = 0xffffffffffffffL;
```

### *Floating Point Literals*

A numeric literal containing a decimal point is a floating point literal, type `double` by default. For example:

```
double d = 1.0;
```

To specify a floating point literal of type `float`, append an `F` (either upper or lower case) to the number. For example:

```
float f = 0.06F;
```

### *Character Literals*

Java uses two bytes for storage of character data in order to support the Unicode character set. This is a standard developed by the Unicode Consortium to accommodate international non-Latin characters. All of the traditional ASCII printing characters are in the same ordinal position in the Unicode character set, so that except for the storage size, printing characters are handled identically between ASCII and Unicode. Non-printing characters are supported in Java by the standard *escape* characters, e.g., `\b` for backspace, `\t` for tab, `\n` for newline, and `\\` for the single character "\". A simple character literal is a character enclosed within single quotes:

```
char term = '.';
```

Java supports the standard C escape sequences: `\n`, `\t`, `\b`, `\\`, *etc.* It also supports `\`*xxx* where *xxx* is 3 octal digits.

# EXPRESSIONS

Java, like C, is an *expression-based* language. Java expressions represent computations and sometimes flow of control. The simplest expression is a literal or variable that has a value and a type. A more complicated expression is composed of an operator applied to one or more operands, all of which are also expressions. These usually represent the machine code to be executed. This results in a new value and type. Examples of this are:

```
int a = 1;
int b = 45;
int c;
c = b // assignment expression
c = c + a
c += a // C-style plus-assignment
```

## Java Operators

Java supports most C++ operators. In addition, it supports a few that are unique to it.

Operators may be classified by their number of operands. Operators that take only one operand are *unary.* Operators that take two operands are *binary.* Java supports one *ternary* operator that takes three operands.

Many Java operators are similar to those in other programming languages. For example, the signs `+`, `-`, `*`, and `/` perform arithmetic.

More Java operators are familiar to C and C++ programmers such as

- `=` assignment
- `==, !=` equality
- `+=, -=, *=, etc.` compound assignment

Programmers familiar with C should note that in Java the relational operators (<, >, <= and >=) produce boolean and not integer results.

# Operations

## *Operations on Integers*

The unary increment and decrement operators ++ and -- can be confusing to those unfamiliar with C. They come in two forms, *pre fix* and *post fix*. They perform two *operations*. They *increment* (or decrement) their operand, and *return* a value for use in some larger expression. In prefix form, they modify their operand and then produce the new value. In postfix form, they produce their operand's original value, but modify the operand in the background.

```
int x = 4, y = 4;
int a, b;
a = x++;
b = ++y;
```

In the above code, if the values of both x and y are incremented to 5, a is assigned x's starting value of 4, but b is assigned y's final value of 5. To interpret these expressions correctly, the programmer must recognize that the ++ operator has precedence over assignment. This may seem obvious to non-C programmers, but it should be remembered that in C based languages, the assignment symbol = is just one of many operators and that it is subject to the same rules of interpretation.

Optimizing compilers often reorder the execution of code. There is no guarantee that the increment of x or y will take place before or after the assignment. The only requirement is that the values be determined in this logical order.

Note that expressions like the following are ambiguous because precedence specifies the logical interpretation of two or more operators, but not the two operands of a single binary operator.

```
a = x + x++;
```

In the above code, it is not known whether the second operand will be evaluated first, thereby modifying the first operand; or whether the first operand will be evaluated first. If x starts with value 4, a could end up with either 8 or 9.

The following is a list of operations on integers:

Unary integer operators

- `-` arithmetic negation
- `+` arithmetic constant
- `~` bitwise complement
- `++` increment
- `--` decrement

Binary integer operators:

- `+`, `-`, `*`, `/` addition, subtraction, multiplication, division
- `%` modulus (remainder)
- `&`, `|`, `^` bitwise AND, OR, XOR
- `<<` left shift
- `>>` right shift with sign fill
- `>>>` right shift with zero fill

There is a compound assignment operator for each of the above binary operators.

Integer operations produce a result of type `int` unless one or more operands is of type `long`, in which case the result type is `long` also. The result is never shorter than an `int` even if all the operands are shorter than an `int`.

The compiler automatically widens integer values:

```
byte b = 0;
int i = 0;
long l = i + b;// byte to int,
          // then int to long
```

The programmer must forcibly shorten integer values by using a *type cast*:

```
byte a = 1;
byte b = 2;
byte c;
c = a + b; // error - int to byte
c = (byte)a + b;// still wrong!
c = (byte) (a + b);// OK
```

Type casting between integer types works in Java as it does in C. Whenever an integer value is converted to a shorter type, the new value is the old value modulo the range of the new type.

### *Operations on Booleans*

Boolean variables or expressions can be combined logically to produce boolean results. The unary `!` operator is logical negation. The binary `&`, `|` and `^` operators correspond to logical AND, OR and XOR respectively. The `&&` and `||` operators are similar to `&` and `|`, but short circuit the evaluation of the right-hand operand if the result is known from an evaluation of the left-hand operand.

The boolean `&`, `|` and `^` operators are unique to Java while `&&` and `||` derive from C. In C and C++, the normal boolean combination operators terminate as soon as their final value is knowable. For example, if `a && b` are being tested, the result will be recognizable as `false` if `a` is `false`. For reasons of speed, `a` will be determined first, and if the result is false, `b` will be ignored. In expressions like `a && b()` where `b()` is a function to call, `b()` will never be called if `a` is `false`. Many programmers utilize this syntax feature to have one expression as a switch, controlling another's execution. The `&`, `^`, and `|` operators in Java are actually a reuse of the bitwise operators. When their operands are boolean types instead of `int` types, they perform logical combinations but with both sides always evaluated.

The equality operators `==` and `!=` work with boolean operands, as do the compound assignment operators `&=`, `|=` and `^=`.

The ternary `?:` operator works in Java as it does in C, except that in Java the condition operand must be of type `boolean`. The types of the second and third operands must be compatible.

### *Floating Point Operations*

Any of the arithmetic integer operators may be applied to floating point operands with the expected results. This includes the compound assignment operators and the increment and decrement operators but not the bitwise operators (&, |, ^, <<, >>, >>>). The increment and decrement operators add or subtract 1.0 to their operand.

The % operator can be applied to floating point operands in Java. Floating point modulus is defined as the floating point remainder, i.e., a %b is equivalent to:

```
a - ((int) (a / b) * b)
```

The binary operators +, -,* and / represent addition, subtraction, multiplication and division respectively.

If both operands of a floating point operation are type `float`, the result is type `float`. Otherwise, the result is type `double`. Keep in mind that floating point literals are type `double` by default. The compiler converts `float` to `double` automatically, but converting `double` to `float` requires a cast, as does converting a floating point value to any integer type. The compiler converts *any* integer type to a floating point type without a cast.

Undefined operations such as division by zero do not generate exceptions in Java as in other languages. Instead, an undefined operation produces one of the special floating point values (NaN, Inf, etc.). The programmer has to keep the special properties of NaN in mind when comparing floating point values.

The following code samples illustrate that `double` to `float` conversion requires a cast; floating point to integer conversion requires a cast; and integer to floating point conversion does not require a cast:

```
float f = 1.0F;
float f2;
f2 = (float) (f + 1.0);// OK
f2 = f + 1.0F;// also OK
```

```
f += 1;    // OK
f = (int) f;// also OK
```

## Precedence and Associativity

### *Precedence*

Java, like C, allows expressions to be arbitrarily complex. The compiler must apply some rules to determine the order of execution in expressions like this:

```
x = a * b + d * d / 15
```

Syntactically, this is made up of four variable names (`x`, `a`, `b`, and `d`) , a literal (`15`), and four operators (`=`, `*`, `+`, and `/`). When two or more operators occur in the same larger expression, the compiler applies the two rules of *precedence* and *associativity.*

*Precedence* determines which operators are interpreted before which others:

```
5 + 2 * 3
```

The above is interpreted as, "evaluate `2 * 3`, produce `6`, then evaluate `5 + 6`, produce `11`."

Java's precedence rules are similar to C's.

Arithmetic operators work as expected with multiplication and division always taking place before addition and subtraction.

The use of extra parentheses is harmless and makes the expression more readable:

```
x = ((a * b) + ((d * d) / 15))
```

There are two other noteworthy precedence rules:

1. Arithmetic operators always take precedence over relational operators.

The following is correct:

```
int index, upper;
...
boolean inBounds = index < upper - 1;
```

2. Relational operators take precedence over bit wise logical operators. The parentheses are necessary in the following:

```
int val;
...
boolean isZero = (val & 0xff00) == 0;
```

Where there are several operators with the same precedence, *associativity* determines which operator is interpreted first.

**TIP**

*Rule of thumb: when precedence is in doubt, use parentheses to specify the order of execution.*

```
40 / 5 / 4
```

The example above is interpreted as, "`40 / 5`, produce `8`, then evaluate `8 / 4`, produce `2`."

Arithmetic operators *associate* left to right.

```
x = a * b + d * d / 15
```

In the expression above the multiplication operators have highest precedence, the division occurs after `d * d`, then the addition happens, and finally the result is assigned into `x`. *Assignment* is an operator with a very low precedence.

Table D summarizes operator precedence.

TABLE D. *Operator Precedence*

| Prec. | Operator | Operand Type(s) | Assc. | Operation Performed |
|---|---|---|---|---|
| 1 | ++ | arithmetic | R | pre- or post-increment (unary) |
| | -- | arithmetic | R | pre- or post-decrement (unary) |
| | +, - | arithmetic | R | unary plus, unary minus |
| | ~ | integer | R | bitwise complement (unary) |

TABLE D. *Operator Precedence*

| Prec. | Operator | Operand Type(s) | Assc. | Operation Performed |
|---|---|---|---|---|
| | ! | boolean | R | logical complement (unary) |
| | (type) | any | R | cast |
| 2 | *, /, % | arithmetic | L | multiplication, division, remainder |
| 3 | +, - | arithmetic | L | addition, subtraction |
| | + | String | L | String concatenation |
| 4 | << | integer | L | left shift |
| | >> | integer | L | right shift with sign extension |
| | >>> | integer | L | right shift with zero extension |
| 5 | <, <=, >, >= | arithmetic | L | less than, greater than, or equal to |
| | instanceof | object, type | L | type comparison |
| 6 | ==, != | native | L | have same or different values |
| | ==, != | object | L | refer to the same or different object |
| 7 | & | integer or boolean | L | bitwise AND or boolean AND |
| 8 | ^ | integer or boolean | L | bitwise XOR or boolean XOR |
| 9 | \| | integer or boolean | L | bitwise OR or boolean OR |
| 10 | && | boolean | L | conditional boolean AND |
| 11 | \|\| | boolean | L | conditional boolean OR |
| 12 | ?: | boolean, any, any | R | conditional operator (ternary) |
| 13 | = | variable, any | R | assignment |
| | *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, \|= | variable, any | R | assignment after operation |

### *Associativity*

When two operators have the same precedence, *associativity* determines evaluation order.

For example, the multiplicative operators (`*`, `/`) have the same precedence and associate left to right. Therefore, the example below has the value 9, not 1.

```
6 / 2 * 3
```

For the assignment operators, right to left association is more appropriate. The two examples below are equivalent.

```
a = b = 3
```

```
a = ( b = 3 )
```

This means, "set b's value to 3; copy its value to a."

Assignment: This is an operator with right to left associativity which changes the left-hand argument's value and produces a value for the next operation.

The above example assigns 3 into b and produces the value of 3. The next assignment operator assigns 3 into a. This also produces a value of 3 (which is thrown away).

This *reverse* associativity makes it easy to assign several variables at once:

```
a = b = c = d = e = 0
```

## STATEMENTS

Java, like C, is an expression-based language. Expressions must always appear in the context of *statements*. A statement consists of an expression followed by a terminating semicolon (;).

Like in C, the Java format of text on a line has no significance to the compiler except for:

- C++ style comments which end at the end of a line
- quoted string literals (e.g., "`Hello World!`") which cannot span more than one line
- white space that the compiler recognizes as the separator between keywords, identifier names, operators, and other symbols

Statements can be made up of zero or more expressions, provided that their combination makes syntactic sense. Expressions (except for the return value

of `void` methods) have a value and a type. Many involve executable code, though this is not necessary. A singe value (e.g., `27`) is a legal expression. A terminating semicolon (e.g., `27;`) turns it into a legal statement even though it does not do anything useful. A *simple statement* consists of an expression and a semicolon. The semicolon is a required *terminator.*

The following are expressions:

- `a`
- `i * j`
- `z = x * y`

Adding semicolons to these expressions turns them into statements:

- `a;`
- `i * j;`
- `z = x * y;`

The first two are not legal and return errors with the 1.0.2 Java compiler. The third calculates a product and stores it into `z`.

## Control Structures

Flow control in Java uses similar syntax as in C. The syntax for the most commonly used control structures may be summarized as follows:

- `if`
  - `if ( boolean-expression ) statement`
  - `if ( boolean-expression ) statement else statement`

```
if (i > 0)
    x = y * z;
else       // else part is optional
    x = 0;
```

**NOTE**

*The difference between `while` and `do-while` refers to when the condition check is done—before or after executing the loop.*

- `while`

```
while (boolean-expression ) statement
```

```
while (getMember (i))
    i++;
```

```
do statement while (boolean-expression)
```

```
do
    a += a;
while (a <= limit / 2);
```

- `for`

```
for (initialization; boolean-expression; next-
iteration-setup) statement
```

```
for (i = 0; i < 22; i++)
    a = a + i;
```

The example above results in the following:

- initialize `i` to `0`
- continue only if `i < 22`
- do body of loop
- increment `i`

Note: The conditions must be of type `boolean`.

### *Compound Statements*

In many contexts, the program will need to execute several statements in a place where the syntax permits only one. A *compound statement* is syntactically a single statement, yet it may include as many other statements as needed. These statements can be of many kinds, including other compound statements. A *compound statement* (also called a block) is delimited by a pair

of curly braces ({}) and can contain any number of declarations and statements. A compound statement may be used wherever a simple statement is legal.

Compound statements allow the use of several statements where syntax would normally expect just one:

```
if (a < b)
{
    ijk = xx + 2;
    bz = ijk * 6 + 22;
    ...[more work]
}
```

The entire compound statement above is part of the original `if` statement.

### *The **continue** Statement*

The `continue` statement causes an immediate branch to the end of the innermost loop that encloses it, skipping over any intervening statements. It consists only of the keyword `continue` and a semicolon.

A `continue` does not cause an exit from the loop. Instead, it immediately initiates the next iteration. The loop control expression will be evaluated, and if it is *true*, another iteration of the loop will be performed. The increment expression of a `for` loop will also be evaluated. For example:

```
int i;
for( i=1 ; i<=10 ; i++ )
{
    if (i == 5)
        continue;
    ...[do more work]
}
```

`continue` can be thought of as meaning “get on with the next iteration.”

### *The **break** Statement*

Like `continue`, the `break` statement consists only of the keyword and a semicolon. A `break` causes an immediate jump out of a loop to the first statement after its end. The loop control expression is not re evaluated, and neither is the iteration expression of a `for` loop.

The `break` statement makes it possible to perform loop control in the middle of a loop and is useful for handling any errors or exceptions that may occur during iteration.

```
while ( 1 == 1 ) // always true!
{
    ...[do something]
    if (a == b) break; // time to leave the loop!
    ...[do more work]
}
```

If loops are nested, a `break` or `continue` affects only the innermost loop:

```
for ( i = 1 ; i <= 10 ; i++ )
{
    for ( j = 1 ; j <= 10 ; j++ )
    {
        if ( j == 5 )
           break;/* exits j loop, not i loop */
        ...[do work here]
    }
}
```

### *Labeled* ***break*** *and* ***continue***

**NOTE**

*In this example, a regular break (without `loop1`) will only get us out of the inner loop.*

In Java (as in C and C++) a *label* is an identifier followed by a colon:

```
Point_A:   a = x + y;
           ...[do more work]
```

In C and C++, this is a justifiable use for the troublesome `goto` statement. In Java, `break` and `continue` followed by a label can help in identifying an *enclosing* loop. For example:

```
loop1:   while (a != b)
{
        ...[do something useful]

        while (i != j)
        {
           ...[do some work]

// Should we quit both loops now?
           if (zztop > y+1)
           break loop1;

           ...[do more work]
        }
}
```

This enables the breaking out of an enclosing outer loop if two or more loops are nested within.

### *The* ***switch*** *Statement*

The `break`, `continue`, and `return` statements provide the ability to leave a block of code at one of several places. The `switch` statement provides variable *entry points* to a block.

```
switch( control-expression )
{
    case value : statements
    case value : statements
    … etc.
    default: statements// optional default section
}
```

The *control expression* is evaluated and compared in turn with each *value* prefaced by the `case` keyword. The values must be *constants* (i.e., determinable at compile-time) and may be of type `byte`, `char`, `short`, `int`, or `long`. Execution begins at the first statement following the `case` for which the value matches the control expression.

If no match is found, execution begins with the first statement following the `default` label. The `default` is optional, but it is good style to include it in every switch. If there is no match, and there is no `default` provided either, none of the statements in the switch will be executed, with the execution continuing with the next statement after the closing bracket.

Execution of statements does *not* stop when the next case is reached. Once an entry point has been found, *all* of the remaining statements in the `switch` will be executed unless an explicit branch is performed by usually a `break` statement.

When included in a `switch`, the `break` will cause an immediate branch beyond the terminating brace of the `switch`.

## Scope

Java has no global variables. All variables must be declared within either a *class scope* or a *function scope*. As in C++, all class variables are *in scope* within their entire class, even if their use comes before their declaration. Examples of class scope variables are explained and illustrated in a later chapter on Java classes.

Local variables, (i.e., variables declared within an executable block) must be declared before they are used. As in C++, but unlike in C, a variable declaration may follow the first executable statement in its block. A local variable is in scope from the point it is declared until the end of its block. Its block is defined by the compound statement that contains the variable declaration.

Once a local variable is declared there can be no other local variable declared of the same name until the variable goes out of scope. This is true even if the second declaration appears in a more deeply nested block. This is yet another departure from C/C++.

Like C++, Java allows the first clause of a `for` statement to be a variable declaration. Unlike C++, Java limits the scope of such a variable to the `for` loop only, which is convenient for the programmer.

A local variable is a variable declared within a block of executable code. It may be declared anywhere within a function or a compound statement. A local variable is in scope from the point of its declaration until the close of its block (compound statement). There may be only one local variable of a given name within scope at one time.

The first clause of a `for` statement can declare a variable that is local to the `for` loop:

```
{
    int x;

    for (int i = 0; i < 60; ++i)
    {
        int x;// error - two x's!
    }

    for (int i = 0; i < 60; ++i)
        // ok - prior i is out of scope!
    {
    }
}
```

These rules differ from C++ rules.

### *Functions*

Java uses a syntax similar to that of C and C++, and many other programming languages to represent a function call.

```
func-name ( param-list )
```

The parameter list is separated by commas. The function returns a value unless its return type is declared `void`.

Functions declared other than `void` return values of the declared type. Here are two examples:

```
long t = currentTime ();
double a = areaOfOval (width, height);
```

### *Simple Output*

Simple output in a Java program ( JDK 1.1) can be programmed by creating a PrintWriter object called `stdout` and using its `print` or `println` methods:

*In this example, `stdout` is being used as a regular programmer defined name and not a special name as in C.*

```
PrintWriter stdout = new PrintWriter(System.out, true);
stdout.println (“...”);
stdout.print (“...”);
```

The `println` method ends output with a newline character. The parameter of these methods should be a string, either a variable of type `String` or a quoted string literal. Other values can be appended to the first string with the + symbol. This works because:

- The compiler recognizes the + with the `String` class as meaning concatenate.
- All class-type variables inherit the method `toString()` from the superclass `Object` (from which all Java variables are derived). Thus, anything can be converted to a `String` and concatenated to the initial `String` for output.

## LESSON SUMMARY

In this lesson, you have learned:

- Java is purely object-oriented.
- Java syntax is similar to that of C and C++.
- Java's native data types include:
  - `byte`, `short`, `int`, `long`, `char`
  - `bool`
  - `float`, `double`

## REVIEW QUESTIONS

1. Given the following declarations, what are the result types of the Java expressions below?

```
int i;
byte b;
long l;
float f;
double d;
```

a. `i << 1`

b. `b + 1`

c. `l + i + b`

d. `f * 1.0`

e. `d % i`

2. What is the difference between:

a. the boolean operators & and && ?

b. the integer right shift operators >> and >>> ?

3. Find the compiler error in the following code:

```
int x = 1;
do
{
    int half = x;
    x *= 2;
}
while (half < 1024);
```

*Answers on page 196*

## EXERCISE

1. Go to the exercise subdirectory of the Java directory. Edit the file `DivTest.java`. In the area provided, insert Java code that prints one of the following two messages:

   ➤ `x` is evenly divisible by `y`

   ➤ `x` is not evenly divisible by `y`

   depending on whether the value of variable `y` evenly divides the value of variable `x` in the `DivTest` class.

   Modify your program to print the values of `x` and `y` instead of "`x`" and "`y`". Test your program by setting various values for `x` and `y`, recompiling and running your program. What happens if you set `y` to zero and run?

2. Still in the exercise subdirectory, look for a source file named `Digits.java`. Edit the file, which contains the beginnings of a program to take the value of the variable `x` and print an English word for each of its digits. For example, if the value of `x` is `496`, the program prints:

   ```
   four nine six
   ```

   Finish the program. (Hint: use a `switch` statement to translate a digit to a string.) Test your program. Make sure that it works for a value of zero.

# LESSON 4

## Classes in Java

### OVERVIEW

*A class is a user defined type created by grouping together pertinent methods and data members. The class is a key component of Java language, as it facilitates modularity and code reuse. Once created, a new type can be used in expressions in the same way as a built-in type (such as* `int` *or* `char`*). While Java classes are, on the whole, similar to C++ classes, there are still small but crucial differences.*

### LESSON TOPICS

- Java Is Object-Oriented
- Instantiating a Class
- Class-Type Variables
- Operations on Class-Type Variables
- The `null` Value
- Member Access
- Class Definitions

## OBJECTIVES

By the end of this lesson, you should be able to:

- Work with Java class instances.
- Declare and initialize instance variables.
- Access data members and methods of an instance.

## JAVA IS OBJECT-ORIENTED

Java is an object-oriented (OO) programming language. A Java program may be thought of as a system of interacting objects in which each object has its own state and behavior. The form and behavior of each object is defined by the object's *class*. A class is a data type and a code module. Designing a Java program means designing the classes that make up the program.

Object-oriented program design builds from the bottom up as much as it does from the top down. Objects are constructed from other objects, and they may, in turn, become the components of larger objects. The developer abstracts the behavior of similar objects into *class* definitions. Similar classes are further abstracted into *superclasses*, resulting in hierarchies of related classes. Viewed from the opposite perspective, the programmer defines classes that are highly abstract, derives more specialized classes from them, then creates the actual objects as instances of those classes.

Object-oriented programming results in programs whose structures more accurately map to the problem domain and are thus are more likely to meet users' needs. The object model carries modularity to a much higher level of abstraction than the structured model. The cross linking that makes both development and maintenance a painful exercise in tracing subtle side-effects is greatly reduced. This modularity, together with the more flexible structure of OO programs, makes software easier to develop, debug, enhance, and modify.

The generalization of highly modular objects into classes and classes into families of related types also permits much greater reuse of code than has been achieved hitherto by way of function libraries, database systems, and other traditional means. Reusability combines with streamlined development and maintenance to make programmers far more productive.

## Classes, Objects and Variables

A Java class definition establishes the interface and implementation of a category of objects.

Using a syntax rooted in that of C structures, the programmer defines *data members* and *function members* of a class. Data members specify the internal data representation. Function members serve two purposes: their headers (names, parameters, and return types) specify the interface visible to code outside the class, and their bodies specify the *methods* by which objects of the class perform the work they are designed to do.

User code takes advantage of a class specification by *instantiating* the class, i.e., by creating *instances* of it. A variable of a programmer defined type is declared with the same syntax that is used to define variables of built-in types. Each instance of a class is an *object*. The object's data members can take on values unique to that object, while its member functions implement the behavior common to all objects of the same class. There can be any number of instances of a given class at any one time.

A Java class-type variable identifies an instance. It does not *contain* an instance, as is the case in some other object-oriented programming languages. There can be any number of variables referring to a given object at any one time.

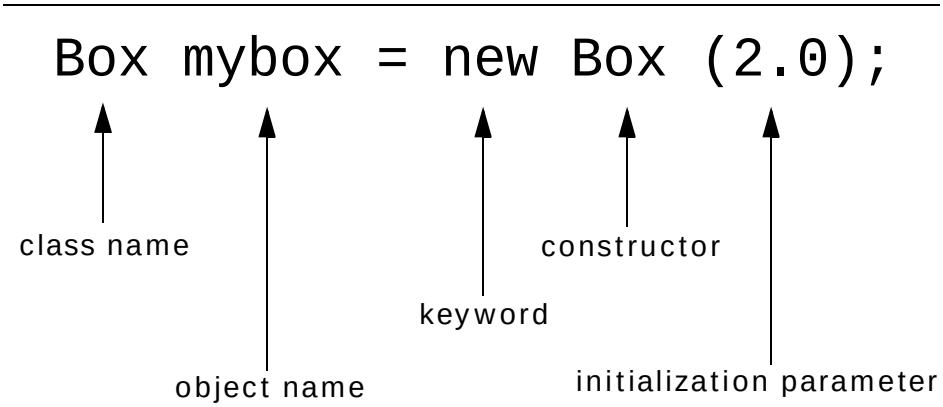
## INSTANTIATING A CLASS

With the exception of array objects, class-type objects may be instantiated only through the `new` operator. The keyword `new` must be followed by the name of the class to be instantiated, and then by a list of initialization parameters. The `new` operation allocates space for a new object, initializes the new object, and returns it. For example, the creation of an instance of `Box` (object) is shown in Figure 3:

**WARNING**

*Unlike C++, a declaration like* `Box myBox;` *does not create an instance of the* `Box` *class. The instance is created only after* `new` *is called on* `Box`.

FIGURE 3
*Creation of an instrance of* Box




The new operator allocates space for the new object, calls a class constructor and returns a reference to the object.

It should be noted that the class and its constructor have the same name.

## CLASS-TYPE VARIABLES

Creating the variable looks just like a native-data type definition. However, the variable is simply a reference. It does not actually instantiate the object. Class-type variables are *references*, not objects themselves. Instantiation of an object with the keyword new and declaration of a class-type variable are separate operations. These two steps may be combined in one statement as an initialization. For example:

```
Box myBox = new Box (2.0);
```

A class-type variable *refers to* (points to) an instance; it does not *contain* an instance. There can be any number of variables referring to a given instance at any one time. The difference between a class-type variable and a variable of native type which contains a value of its type is to be noted.

# OPERATIONS ON CLASS-TYPE VARIABLES

All class-type variables support the following operations:

- assignment
- ==
- !=

The assignment operator reassigns the object to which the variable refers. Assignment does not modify the value of any object.

Suppose there is a class named `Box:`

```
Box box1;
Box box2 = new Box (2.0, 3.0);
Box box3 = new Box (box2);

box1 = box2;
box3 = new Box ();
```

When a class-object variable is used in code, it does not imply the value stored in the variable. Rather, it implies the *object variable references*. In the example above, `box1` and `box2` are set to reference the same object. Then `box3` is given a brand new object. This is very different from the behavior of native-data type variables. In this sense, a class variable is more like a class pointer from C++ than a class object.

Below is an illustration of reference semantics:

➤ two variables initialized to refer to two separate objects:

```
Box box1 = new Box (1.0, 1.0);
Box box2 = new Box (2.0, 2.0);
```

➤ assignment:

```
box1 = box2;
```

➤ the effect of the assignment:

- `box1` and `box2` refer to the same object.
- The object `box1` referred to previously is available for *garbage collection*.
- Neither object changed in *value.*

## ➤➤➤ THE `null` VALUE

Another way in which Java class-type variables are similar to C and C++ pointers is that Java class-type variables may be `null`. A Java class-type variable with the special value `null` refers to no object. The value `null` is compatible with any class type, i.e., `null` may appear wherever a class-type value is expected. Attempting to access a member of a null variable results in a run-time error.

# MEMBER ACCESS

Once a class has been instantiated, variables for that instance are accessed by the "`.`" symbol (dot).

For example, suppose that `myCircle` is a variable of class `Circle`, and class `Circle` has a data member named `radius` of type `double`; then the following statements are legal:

```
myCircle.radius = 42.6;
double rad = myCircle.radius;
```

Programmers familiar with C have to note that there is no `->` operator because object variables are not pointers.

**NOTE**

*If `radius` were a class variable, it could be accessed as `Circle.radius`. Unlike C++, you don't need to use `::` after class name in Java.*

## Invoking a Method

Objects are usually thought of as self-contained entities whose characteristics are defined by, among other things, the set of actions they are intended to perform. A central concept of OO programming is that objects *pass messages* to each other. Objects can be seem as being self-contained and responsive, like living organisms reacting to signals from the world around them. External code triggers actions from, and gets information about, objects only by sending messages to them and interpreting the responses. A call to an object's method is a message to that object requesting that it perform some specified action.

A message has three components: a *target*, a *selector,* and a *parameter list*. Java reflects its derivation from C by implementing messages as *function calls,* using a syntax like that used to access members of structures. The user code references the object (the target), then invokes a function which is a *member* of that object (the selector), and supplies the arguments that function requires (the parameter list).

For example, suppose the object `myCircle` contains the following methods:

- `getArea`, which returns a `double`
- `moveTo`, which takes two double arguments

```
double area = myCircle.getArea ();
myCircle.moveTo (0, 0);
```

## CLASS DEFINITIONS

Every data item or method in a Java program, even the starting function `main()`, must be declared as a component of a class.

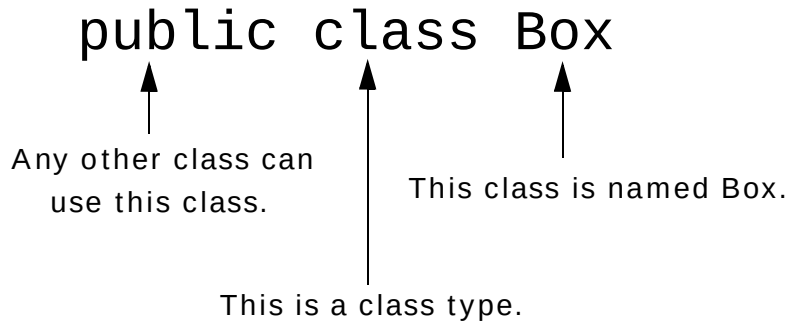
A class definition consists of the keyword `class` followed by the class name, then a set of curly brackets {} that designate the body of the class. Variables and methods declared here are called *class members*. Class member declarations, and the class itself, can be preceded by an access specifier. Public classes can be instantiated by any other code.

The following is an example of a Java class definition:

```
public class Box {
    double height, width;

    // Constructors
    public Box(double h, double w) {
        height = h;
        width = w;
    }
    public Box(double s) {
        height = width = s;
    }

    public double getHeight() {
        return height;
    }

    public double getWidth() {
        return width;
    }

    public double getArea() {
        return height * width;
    }
}
```

Figure 4 illustrates what the header communicates.

FIGURE 4
*Meaning of headers*




The class body (enclosed in `{...}`) defines the members of the class: variables, or data members, methods, or function members. Members can be

declared public—they are not public by default. A private member can only be accessed by another member of that class. A public member can be accessed directly from outside the object. The principle of *encapsulation* infers that data members should be private and that at least some of the function members should be public. The public methods define an object's interface.

## Declaring Instance Variables

All variables, unless declared `static` (discussed later), are replicated for every instance of the class and therefore are referred to as *instance variables.* For example, in the class definition for `Box`, the following code defines two instance variables:

```
double height, width;
```

Every instance of type `Box` receives its own `height` and `width`.

When a class is instantiated, an object of that type is created; the new instance obtains its own copy of every member variable including the methods. For example, consider the following class:

```
class Color {

    int r, g, b;

    public setColor(int red, int grn, int blu) {
        r = red;
        g = grn;
        b = blu;
    }
}
```

If `background` was the name of an instance of `Color`, then the color could be defined by calling its `setColor` method:

```
background.setColor(255, 100, 0);
```

Note that the color could *not* be set by assigning values to the data members directly because they are private and not public.

The following statements would all generate compilation errors:

- `background.r = 255;`
- `background.g = 100;`
- `background.b = 0;`

## Declaring Instance Methods

Methods defined inside a class are instance methods by default. An instance method must be applied to an instance of the class of which it is a member. For example, the class `Box` defines the methods:

- `public double getHeight() { ... }`
- `public double getWidth() { ... }`
- `public double getArea() { ... }`

These methods can be applied to any instance of the class `Box`.

Each method is defined fully within the class definition: the body of the method is enclosed within curly braces (`{...}`).

Methods can refer to instance variables of the target object directly:

```
public double getArea() {
    return height * width;
}
```

Suppose we have a variable of type `Color` named `background`. We might choose to "blacken" the background color by applying the method `setColor()` to the instance `background`, by calling:

```
background.setColor(255, 255, 255);
```

This applies the method `setColor()` to the instance `background` by using the member access symbol "`.`" (dot). Inside the method, the variables are used by name and they are taken as applying to this instance's copy of each. Methods like this are called *instance methods* because they can be applied to individual instances.

**TRIVIA**

*This conforms to the concept of encapsulation in object-oriented programming in which class data is held privately within the class and outside code uses functions rather than directly accessing class data.*

## The **this** Variable

Java provides a keyword called `this` which allows an object to refer to itself. Unlike the `this` pointer in C++, the Java version is simply another name for the object. For example, if an object has a data member `width`, then the following statements are equivalent when used within one of the object's function members:

```
width = w;
```

```
this.width = w;
```

The `this` keyword is not often needed, but there are times when it is useful. Whenever an instance method must refer to the object of which it is a member (or one of its constructors), it can use `this` to do so. It can also be used to distinguish between parameter variables and data members having the same name.

**WARNING**

*If a parameter name is the same as a class member name, an unqualified name will refer to the parameter and not to the class member. `this` must be used to refer to the class member. It is a good practice to avoid having the same name for parameters and members.*

```
public class Box {
    double height, width;
    public Box(double height, double width) {
    // Note that the parameter variables have the same
    // name as the data members, so we use the this
    // keyword when referring to the data members.

        this.height = height;
        this.width = width;
    }
    public Box(double s) {
    // Here we use the this keyword to invoke the
    // two-parameter constructor with the single
    // parameter,saving us the trouble of
    // rewriting the assignment statements.

    this(s, s);
    }
    // Other methods...
}
```

## Class Definitions and Source Files

Each class is defined entirely within its source file. Unlike C and C++, there are no header files in Java. While compiling a class, the Java compiler examines other compiled classes to check references and types across class boundaries.

There can be at most, one public class per Java source file. A Java source file may contain any number of non-public classes, but those classes are limited to use within the same package and are best used as "helper" classes for the `main` public class in the same source file. If a Java source file contains a public class, the name of the source file must match the name of the class, i.e., the source file must be named `<classname>.java`, where `<classname>` is the name of the public class defined within.

## LESSON SUMMARY

In this lesson, you have learned:

- Java classes are user defined types that facilitate code reuse and modularity.
- A user defined type is used in the same way as a built-in type.
- Class members are accessed using the dot (`.`) operator.
- The `this` keyword allows an object to refer to itself.

## REVIEW QUESTIONS

1. How would you declare an object of type `Animal` named `Lion` that takes a weight of `500` and length of `45` as parameters?

2. What code would change the weight to `250` and the length to `35` in the above `Lion` instance? Assume that `weight` and `length` are the names of the `Animal` data members.

3. How many public classes are allowed in a Java source file?

*Answers on page 197*

## EXERCISE

1. Create a new class named `Clock`. (You may put your new class in any package you like.) Class `Clock` has the following attributes

   - Hour (integer 1..12)
   - Minutes (integer 0..59)
   - Seconds (integer 0..59)
   - `isAM (boolean)`:

   These attributes are represented by data fields with default access protection. Class `Clock` supports the following public extractor methods:

   - `getHours()`
   - `getMinutes()`
   - `getSeconds()`
   - `getIsAM()`

   Class `Clock` may be constructed in any of the following ways:

   - no parameters (sets clock to midnight)
   - hour and `isAM` specified (sets minutes and seconds to zero)
   - all attributes specified

   Class `Clock` also supports a `setTime()` method that sets all four attributes from argument values. Compile the new class.

# LESSON 5

## Classes in Java—II

### OVERVIEW

*Java classes have some additional features that aid program development. For example, class methods may be called with varying numbers and types of parameters by using overloading. Two keywords,* `static` *and* `final`*, can be used to further adjust the behavior of class members. Another useful mechanism is the* `finalize` *method, which performs clean-up work when a class instance is destroyed.*

### LESSON TOPICS

- Method Overloading
- Constructors
- Encapsulation
- Access Specifiers
- Comparing Objects
- Class Variables
- Class Methods
- Finalization

## OBJECTIVES

By the end of this lesson, you should be able to:

- Apply overloading mechanisms.
- Employ the `static` and `final` keywords.
- Use initialization syntax.
- Understand finalization methods.
- Write complete Java classes.

## METHOD OVERLOADING

Overloading means declaring a method with the same name, more than once. To distinguish between the different versions, the compiler uses the supplied parameter list. In other words, a method is identified by its *signature*, made up of both the name and the types of its parameters. The return type of a method is not a part of its signature. An overloaded method must differ from other methods of the same name in number and/or types of arguments:

- `public double dothis (Circle cx) {...}`
- `public void dothis (double x) {...}`
- `public void dothis () {...}`

A method's return type does not distinguish it:

```
public double dothis (Circle cx) {...}
public void dothis (Circle cx) {...}
//
// ERROR: these methods conflict
//
```

When a call to an overloaded method is made, the compiler must perform overload resolution to determine which method is being called. The compiler considers all the methods of the same name and number of arguments

as the call, compares the actual types of arguments of the call with the formal argument types of the methods, and chooses the appropriate method.

## CONSTRUCTORS

A *constructor* is a special method that initializes a newly instantiated object. Constructors have the same name as the class and no return type, not even a void. They can take any number of arguments. They are called whenever a class is instantiated, and are therefore used to initialize that instance. Constructors can be overloaded, so a class programmer can offer various ways for a class user to instantiate and initialize the class.

**WARNING**

*While there may be multiple constructors for a single class, they must differ somehow in the number or types of arguments. There cannot be two constructors for one class with exactly the same number and types of arguments.*

The following example gives the `Box` class two constructors:

```
public class Box {
    double height, width;

    public Box(double hw) {
        height = width = hw;
    }

    public Box(double h, double w){
        height = h;
        width = w;
    }
    ...
}
```

If a class does not provide any constructor explicitly, the compiler generates one that uses no parameters and is sometimes called the *default constructor.* If a class provides *any* constructor, the compiler does not provide this no-parameter version.

**WARNING**

*A different constructor from within one constructor may not be called with the usual new method, since that would create a brand new object instead of calling a different constructor as part of building the current object.*

A constructor can use the keyword `this()` as a *method* to call a different overload of that class's constructor. This can help avoid redundant coding:

```
public Box(double s) {
    this(s, s);
}
```

Instance variables are *already initialized* by the time the constructor executes. The constructor need only perform additional initialization.

A new operation always calls a constructor to fully initialize a new object. Remember that the choice of constructor to call depends on normal overload resolution rules based on the initialization parameters:

```
Box b = new Box (1.0, 2.0);
// calls: Box (double h, double w)
```

## ENCAPSULATION

*Encapsulation* is combining of behavior and data into the discrete program units called *objects*. The programmer hides the data behind a specific interface (usually a purely functional one) that controls access to the data. Encapsulation prevents external code from modifying the data directly avoiding the possibility of unwanted side effects.

By hiding the underlying implementation details, the programmer hides the way in which an object is represented and manipulated from the rest of the program. In this respect, encapsulation contributes to program modularity, as it groups code and data components in discrete, self-contained, and self-protecting objects. By defining an interface made up only of methods, the programmer provides a clean interface to the object and limits the actions that can be performed on it.

A well-defined, fully encapsulated object provides a clear, comprehensive interface, and takes care of its own security. External code cannot access its internals—and has no reason to do so.

## ACCESS SPECIFIERS

Classes can be declared `public`. A public class is accessible by any other Java class. A class that is not declared public has *package access*, which means that only classes within the same package may access it.

Class *members* can be declared `public` or `private` to enforce proper encapsulation. Certain data elements or methods that are necessary for the class's internal behavior should be protected from access by outside classes. These should be declared `private`. Class members labelled `public` are accessible to all other classes; members labelled `private` are accessible only to the code of the class itself. There is also a `protected` access speci-

fier that will be discussed in the lesson on inheritance. A member with no declared access specifier has package access.

## COMPARING OBJECTS

Through the mechanism of inheritance (discussed in a later lesson), every Java object has a method called `equals()` that takes another object as its argument and returns true if the target object is equal to the argument object. This sort of equality is not necessarily the same as the one implemented by the == operator, which returns true if and only if its operands refer to exactly the same object. Types such as `String`, in which equality depends on the value of an object and not simply the object's identity, redefine the `equals()` method to compare values instead of references.

To compare the *values* of two objects, use the `equals()` method:

```
if (r2.equals (r1)) {
    ...
}
```

The behavior of `equals()` depends on the class. The default behavior is to compare object identities just as the == operator does. Where appropriate, it compares values.

## CLASS VARIABLES

Normally, variables declared in a class are instance variables, which appear once per instance of the class. It is sometimes useful to define variables that, like global variables in other programming languages, are unique and persist throughout the lifetime of the program.

A Java class can have data members that appear once per class, rather than once per instance, and persist for the duration of the program. These members are called *class variables*.

**NOTE**

*This is similar to the use of `static` for class members in C++.*

To declare a variable of this sort, add the modifier `static` to a class variable declaration:

```
public class Box {
    double height, width;
    static int BoxCount = 0;

    public Box(double h, double w) {
        height = h;
        width = w;
        BoxCount += 1;
    }
    ...
}
```

Within the class, a class variable may be referred to by just its name and no qualifier. Outside the class, the variable name must be qualified by either an instance of the class or the name of the class. Qualifying a class variable by an instance of the class is identical to accessing an instance data member. Since class variables are associated with a whole class and not any particular instance, accessing a class variable does not require an instance of the class. A class variable may be qualified by applying the dot operator to the class name, as if the class name were a class instance.

## Class Initialization

**TIP**

*Static initialization blocks can be used to assign the results of a complex computation to be the initial value of a certain static variable. You are not limited to using constant values for initialization.*

Class variables may be initialized in the class definition. The required syntax is identical to initialization of instance variables. Any variable not explicitly initialized receives the default value for its type.

The initialization takes place when the interpreter first loads the class. Additionally, a class may specify arbitrary initialization code to be run when the class is loaded. This is accomplished by using a *static initialization block*. The syntax for defining a static initialization block is the keyword `static` followed by the body of a method. A class may have more than one static initialization block. When a class is loaded, all its static initialization blocks and static variable initializations execute in the order in which they appear in the source file.

The following is an example of static initializer blocks:

```
public class Box {
    static int num_of_Boxs = 0;
    static float peculiar;

    static // a static initializer
    {
        peculiar = 2.7f + 15;
    }
    ...
}
```

## `final` Variables

A class-member variable can be declared with the modifier `final` meaning that its value cannot be changed in other parts of the code once the variable is initialized. Java has reserved C's keyword `const` though it currently has no meaning.

A class variable declared `final` is a class-wide constant. Constants are often declared public to make them available program-wide.

```
class Trig {
    public static final float PI = 3.14159;
... more stuff ...
}

    float zz;
    zz = 2.0f * Trig.PI;
```

**NOTE**

*Java does not have any preprocessor and thus does not support constants in the sense of C's `#define` directive.*

**TRIVIA**

*Java designers reserved the `const` keyword to avoid potential confusion from C/C++ programmers inadvertently using `const` in Java.*

Variables declared `static` and `final` are true program constants:

**WARNING**

*If a variable is declared final but not static, it could have different values in different instances of the class. This can be confusing if the programmer expects this variable to be a true constant. It seldom makes sense to use final without static.*

```
// create constant suit values
public class Suit {
    public final static int DIAMONDS = 0;
    public final static int CLUBS = 1;
    public final static int HEARTS = 2;
    public final static int SPADES = 3;
}

// create a mathematical constant
public class Trig {
    public static final float PI=3.1416f;
}
```

## CLASS METHODS

**WARNING**

*If a `static` method tries to access an instance variable, there will be a compiler error.*

As discussed in the preceding section, *class variables* (as opposed to *instance variables*) exist before any instances are created. Sometimes the class also needs methods that refer to those variables, and those methods need to be callable by outside code before any instances have been created. Such methods can be declared with the `static` modifier and are called *class methods*, because they apply to *class variables* and not to instances of the class. The following code gives an example of a class method:

```
public class Thing {
    static int value = 3;
    public static void resetValue (int val) {
        value = val;
    }
    ...
}
```

Outside of a class, class methods can be called either by class name or through an instance:

```
Thing.resetValue(9);
Thing th = new Thing(6);
th.resetValue(9);
```

Class methods can access *class* variables, but cannot access *instance* variables. Because class methods may be called with no instances available, they are not given a `this` reference like instance methods.

It is a common practice to define classes that contain only static members. Since such a class has no instance data or methods, instantiating it is useless. Such a class is nothing but a wrapper for a related set of global methods and/or data—a wrapper made necessary by the absence of true globals in Java.

## FINALIZATION

**NOTE**

*Notice this important distinction in how constructors are declared and finalized. Constructors have no return type, but finalize must have void as a return type. Unlike constructors, finalize has only one option for arguments—no arguments.*

When class objects are created, the appropriate constructor is called. What happens when objects go away? In C++, classes can implement a destructor that releases memory when the object is destroyed. Unlike C++, Java does not require a destructor, because memory is not directly allocatable. However, class objects may still create conditions that must be manually undone, such as closing files opened for I/O.

In Java, classes may define a method called `finalize()`. If `finalize()` exists for a class, it is called just before an instance of that class is destroyed. The compiler generates a default `finalize()` method if a class does not provide one. Note that the exact time that `finalize()` is called can never be predicted, simply because the exact time of garbage collection (i.e., actual destruction of objects) cannot be predicted. Under some circumstances, such as when the interpreter exits at the end of a program, it is possible that `finalize()` might never be called. Thus, the programmer of a class should never rely on `finalize()` to do critical tasks. `finalize()` is only useful to free system resources, such as open files and network connections. A finalizer method is declared by naming it `finalize()`with no parameters and a void return value.

# LESSON SUMMARY

In this lesson, you have learned:

- Overloading allows a method to have more than one signature.
- Variables declared `static` apply to the whole class not to specific instances.
- Variables declared `final` cannot be changed after they are initialized.
- Finalization carries out various tasks when a class instance is no longer needed.

## REVIEW QUESTIONS

1. If a class is not declared `public`, then who can instantiate the class?

2. What is the keyword that declares a variable to be constant?

3. What is the keyword that declares a method to be a class method rather than an instance method?

*Answers on page 197*

## EXERCISE

1. Expand the `Clock` class to support a `tick()` method. The `tick()` method increments the current time by one second. Add an empty `main()` method to class `Clock`, so that you can run the class (see `FirstApp.java` for an example of a runnable `main()` method).

   Write test code that instantiates and uses class `Clock` in `main()`.

   Test the clock's transitions from hour to hour, minute to minute, AM to PM.

   Compile and run the test.

2. Create a class called MathTechniques that supports the following constant and methods:

   - `pi` - 3.14159
   - `areaOfRectangle` that is passed a height and a width
   - `perimeterOfRectangle` that is passed a height and a width
   - `areaOfCircle` that is passed a radius
   - `perimeterOfCircle` that is radius

3. Create a separate class to test out the MathTechniques class.

# LESSON 6

## Arrays and Strings

### OVERVIEW

*The array is a useful construct that stores an indexed collection of identical objects. Perhaps the most useful application of arrays in Java is the character string. A string is an object and may be readily assigned, copied, and accessed. A string cannot, however, be modified.*

### LESSON TOPICS

- Java Arrays
- Array Constants
- Using Arrays
- Copying Array Elements
- String Objects
- String Methods
- String Concatenation
- Converting Objects to Strings
- Converting Strings to Numbers

# OBJECTIVES

By the end of this lesson, you should be able to:

- Work with Java arrays.
- Work with Java string objects.

# JAVA ARRAYS

**NOTE**

*Arrays are neither a true class (there is no class called Array), nor a built-in type (Arrays have members like a class). Arrays in Java are in between a class and a data type although they are implemented as a class internally by Java.*

*Arrays* are ordered collections of identical objects that can be accessed via an index. In C and C++, an array is implemented as a contiguous block of memory accessed via pointers to the elements. In Java, arrays are first-class objects. Java arrays have their own behavior that is encapsulated into their definition when the compiler creates them. Since arrays are objects, array variables behave like class-type variables. It is important to distinguish between the array variable and the array instance to which it refers. Declaring an array only declares the array variable. To instantiate the array, the `new` operator must be used with the `[]` operator to enclose the array size. The array size can be any integer expression.

The following code declares an array variable and initializes it to null:

```
char data[] = null;
```

This can also be written as:

```
char[] data = null;
```

To declare an array variable and initialize it to refer to a new instance, use the following syntax:

```
int length = 60;
char[] data = new char [length];
```

Array elements are always initialized according to their type as shown in Table E.

TABLE E. *Array Initialization*

| Type | Initialization |
|---|---|
| integers and floating point | zero |
| characters | null |
| booleans | false |
| class-type objects | null |

## ARRAY CONSTANTS

The declaration of an array must include its type, but not its size. Arrays may be initialized with conventional initialization syntax. An array initializer is either an expression (such as a new expression) that evaluates to the correct array type, or a list of initial element values enclosed in { }. This latter notation is borrowed from C/C++. In Java, this is called an *array constant*. Any expression can be used for the initial array element values. (C requires initializers be compiler-time calculable; both C++ and Java allow initializers to be run-time calculable.)

The following array constant provides initial values for its elements:

```
int[] int_array = {1, 3, 4, 15, 0};
```

Each element of an array of class-type objects behaves like a class-type variable:

- Elements may be allocated by new, making them initially null:

```
Box[] boxes = new Box [100];
```

- Elements may be initialized in an array constant:

```
Box b = new Box (2.2, 3.3);
Box[] boxes = { b, new Box (1.1,2.2) };
```

# USING ARRAYS

Array elements can be accessed using C notation (an index in square brackets, `[n]`), where the index must be an integer. Like arrays in C, arrays start with index number 0, not 1. The index can be thought of as an offset from the beginning of the array.

The index may not be less than zero or greater than the declared size. If the array is declared size `n`, the index must be in the range `0` to `n-1`. Unlike C, arrays are not implemented as pointers, so programming techniques like negative array indexing are not allowed. Any attempt to index an array with an illegal value will cause an exception to be thrown.

All arrays have a data field called `length` that indicates its size. This value is read-only, and contains the number of elements in the array:

**NOTE**

*The `length()` in String is a method while the length in array is a data member.*

```
int ia[] = new int[100];
for (int i = 0; i < ia.length; i++)
{
    ia[i] = i * i;
}
```

Since `length` is read-only, its value cannot be changed manually:

```
ia.length = 20; // error
```

# COPYING ARRAY ELEMENTS

The library method `System.arraycopy()` is useful for copying a number of elements from one array to another. The method can be used on any type of array and is declared as follows:

```
public static void
arraycopy (Object src, int src_position,
    Object dst, int dst_position, int length)
```

The method copies elements from the given source array, beginning at the specified position to the destination array at the specified position. It copies

the number of elements specified by the `length` argument. The destination array must already be allocated.

Any type of array may be copied. If range exceeds bounds of either array, a run-time error results.

## STRING OBJECTS

The Java `String` class (`java.lang.String`) is a class of object that represents a character array of arbitrary length. While this external class can be used to handle string objects, Java integrates internal, built-in strings into the language. This string literal is one way these strings can be used:

```
"This is a quoted string literal"
```

This is actually an object of type `String` having the value of the quoted character array, which is the most common way to instantiate a string object. However, the `String` class does contain several constructors, listed in Table F.

TABLE F. *String Constructors*

| Description | Syntax |
|---|---|
| construct an empty string | `String ()` |
| copy a string | `String (String str)` |
| initialize with a char array | `String (char[] chars)` |
| initialize with an ASCII byte array | `String (byte[] bytes, int hibyte)` |

Strings can be used like any other type of Java object:

```
void printMsg (String userName)
{
    String msg1 = "Hello, ";
    String msg2 = userName;
    if (msg2 == null)
        msg2 = "anonymous user";
    System.out.println (msg1 + msg2);
}
```

An important attribute of the `String` class is that once a string object is constructed, its value cannot change (note that it is the value of an *object* that

cannot change, not that of a string variable, which is just a reference to a string object). All `String` data members are private, and no string method modifies the string's value.

## STRING METHODS

Although a string represents an array, standard array syntax cannot be used to inquire into it. Instead, the `String` class provides equivalent methods that allow access to a string's value or any part of it. These are detailed in Table G.

TABLE G. *String Methods*

| Method | Description | Example |
|---|---|---|
| `int length()` | Returns the number of characters in the string | `String str = "Another string";`<br>`int len = str.length ();` |
| `char charAt (int index)` | Extracts one character from the string | `char first = str.charAt (0)`<br>`char last = str.charAt (len - 1)` |
| `void getChars (int srcBegin, int srcEnd, char[] dest, int destBegin)` | Extracts a range of characters from the string into a character array | `getChars(3, 6, word, 0);` |
| `String substring (int begin)` | Return the substring of this string delimited by character position begin and the end of the string | `String s3 = str.sub-string (8);` |
| `String substring (int begin, int end)` | Returns the substring of this string delimited by character position begin (inclusive) and character position end (exclusive) | `String s1 = str.sub-string (0, 7);` |

### String Comparison

Since string variables are Java class-type variables, the equality (==) operator compares whether two strings are the same object, not whether they have

the same value. To compare string values, use the `equals()` method common to all objects, or one of the other methods listed in Table H.

TABLE H. *More String Methods*

| Method | Description | Example |
|---|---|---|
| `boolean equalsIgnoreCase (String otherString)` | Returns true if this string value equals the other, considering letters equivalent regardless of case | `if (s1.equalsIgnoreCase (s2)) { ... }` |
| `int compareTo (String otherString)` | Returns zero if the string values are equal, a negative value if this string is less than the other, or a positive value if this string is greater than the other | `if (s1.compareTo (s2) > 0) { String temp = s1; s1 = s2; s2 = temp; }` |
| `boolean startsWith (String prefix)` | Returns true if this string starts with the other string | `if (s1.startsWith (s2)) { ... }` |
| `boolean endsWith (String suffix)` | Returns true if this string ends with the other string | `if (s1.endsWith (s2)) { ... }` |
| `boolean regionMatches (int thisBegin, String otherString, int otherBegin, int length)` | General string region comparison | `if (s1.regionMatches(0, s2, 5, 10) {...}` |
| `boolean regionMatches (boolean ignoreCase, int thisBegin, String otherString, int otherBegin, int length)` | General string region comparison, with option of case insensitivity | `if (s1.region-Matches(true, 0, s2, 5, 10) {...}` |

## String Searching

The `String` class also provides methods that search a string for the occurrence of a single character or substring. These return the index of the matching substring or character if found, or `-1` if not found.

- int indexOf (char ch)
- int indexOf (char ch, int begin)
- int lastIndexOf (char ch)
- int lastIndexOf (char ch, int fromIndex)
- int indexOf (String str)
- int indexOf (String str, int begin)
- int lastIndexOf (String str)
- int lastIndexOf (String str, int fromIndex)

The following example shows the usage of these functions:

```
if (s1.indexOf (':') >= 0) { ... }

String suffix =
    s1.substring (s1.lastIndexOf ('.'));

int spaceCount = 0;
int index = s1.indexOf (' ');
while (index >= 0)
{
    ++spaceCount;
    index = s1.indexOf (' ', index + 1);
}

int index = s1.indexOf ("that");
```

## Other String Methods

The `String` class provides miscellaneous methods for manipulating string data. Each of the methods listed below returns a new string instead of modifying its given string. (string objects are immutable by design).

```
String replace (char oldChar, char newChar)
// Replaces all occurrences of oldChar with newChar.
// For example:
s2 = s1.replace (' ', '_');

String toLowerCase ()
// Converts all upper case characters to lower case.
// For example:
s2 = s1.toLowerCase ();

String toUpperCase ()
// Converts all lower case characters to upper case
// For example:
s2 = s1.toUpperCase ();

String trim ()
// Removes leading and trailing white space
// For example:
s2 = s1.trim ();
```

## STRING CONCATENATION

The `String` class provides a method for concatenating two strings:

**NOTE**

*C++ programers should note that the + and += `String` operators are built into the Java language; and that Java does not allow programmer- defined operators.*

```
String concat (String otherString)
```

The + and += `String` operators are more commonly used:

```
String fname = s1 + ".txt";
String path = s1 + "/" + s2 + ".txt";
```

The Java compiler recognizes the + and += operators as `String` operators. For each + expression, the compiler generates calls to methods that carry out the concatentation. For each += expression, the compiler generates calls to methods that carry out the concatenation and assignment.

## CONVERTING OBJECTS TO STRINGS

The `String` + operator accepts a non-string operand, provided the other operand is a string. The action of the + operator on non-string operands is to convert the non-string to a string, then to do the concatenation. Operands of native types are converted to string by formatting their values. Operands of class types are converted to a string by the method `toString()` that is defined for all classes. Any object or value can be converted to a string by explicitly using one of the static `valueOf()` methods defined in class `String`:

```
String str = String.valueOf (obj);
```

If the argument to `valueOf()` is of class type, then `valueOf()` calls that object's `toString()` method. Any class can define its own `toString()` method, or just rely on the default. The output produced by `toString()` is suitable for debugging and diagnostics. It is not meant to be an elaborate text representation of the object, nor is it meant to be parsed.

These conversion rules also apply to the right-hand side of the `String +=` operator.

## CONVERTING STRINGS TO NUMBERS

Methods from the various wrapper classes, such as `Integer` and `Double`, can be used to convert numerical strings to numbers. This is often necessary for command line arguments where the parameter list is available to the `main ()` method as a `string` array.

The wrapper classes contain `static` methods ( such as `parseInt()` ) which convert a string to its own internal data type. These can be used with class names rather than creating a separate object. Be aware that these par-

ticular conversion methods throw a `NumberFormatException` and must be used in the context of a `try` and `catch` block combination.

```
import java.io.PrintWriter;

public class ArgTest {
    public static void main(String[] args) {
        int i, j;
        // Create an stdout object
        PrintWriter stdout =
            new PrintWriter(System.out, true);
        // Make sure we have parameters
        if (args.length == 0) {
            stdout.println("No arguments");
            System.exit(1);
        }
        // Loop through the args array
        for (i = 0; i < args.length; i++) {
            try {
              j = Integer.parseInt(args[i]);
              stdout.println(j + " is an integer");
            }
            catch (NumberFormatException e) {
              stdout.println(args[i] + " is not an integer");
            }
        }
    }
}
```

## LESSON SUMMARY

In this lesson, you have learned:

- Arrays are fixed-size, indexed collections of data elements, all of the same type.
- The Java run-time system catches out-of-bounds array indexes.
- Array variables behave like class-type variables.
- If an array contains class-type elements, then each element behaves like a class-type variable.
- Java strings are objects of class `String`.
- Java strings are immutable.
- Java recognizes + and += concatenation operators.

## REVIEW QUESTIONS

1. What is wrong with the following code:

```
int arr[] = new int[50];
arr.length = 25;
for (int i = 0; i < arr.length; i++)
{
    arr[i] = 0;
}
```

2. Using two different methods, add the text “ball” onto the end of a `String sport` already containing the text “base.” Store the result in `String new_sport`.

*Answers on page 197*

## EXERCISE

1. Write a class named `FloatArray` that represents an array of double-precision floating point values. Class `FloatArray`'s first constructor takes one argument of type `integer` that specifies the size of the array. The initial value of the array elements is zero. Do not try to handle invalid size values.

   Class `FloatArray`'s second constructor takes two arguments: the array size and a double that specifies the initial value of all the array elements.

   Write a `getValue()` method that returns a value from the array, given the index of the element.

   Write a `setValue()` method that sets a value in the array, given the index of the element.

   Write a `getAverage()` method that returns the average of all values in the array.

   Write a runnable `main()` method that tests the class `floatarray`.

   Compile and run the class.

2. Write a static method called `removeString`. (You may create a new class for your method, or work in one you have already written.) `removeString` has the following signature:

```
public static String removeString
     (String string, String substring)
```

   The function of `removeString` is to remove all occurrences of the given substring in the given string. For instance, the following call results in the string `dated`:

```
String result = removeString (“undaunted”,"un")
```

# LESSON 7

## Inheritance

### OVERVIEW

*Inheritance is a powerful feature that allows new classes to be derived from existing ones. The derived class inherits the interface of the base class and can add new methods or change existing ones to suit its purpose. Any number of classes can be added to an inheritance scheme to produce a hierarchy of related types. The Java implementation of inheritance supports polymorphism, which allows objects of different types to be treated as if they were of the same type.*

### LESSON TOPICS

- Introduction to Inheritance
- Protected Access
- Overriding Methods
- Constructor Chaining
- Inheritance and Finalization
- Abstract Classes
- Interfaces
- Casting Between Class Types

## OBJECTIVES

By the end of this lesson, you should be able to:

- Read and understand source code for derived classes.
- Derive a new class.
- Define a set of classes that behave polymorphically.
- Take advantage of Java's run-time typing.

## INTRODUCTION TO INHERITANCE

Developers working with object-oriented languages visualize classes as being related to each other in hierarchies of "kind of" relationships. Just as programmers generalize the common features of objects in an abstract data type, they also identify and abstract common features of similar types in a *superclass*. Once the superclass is defined, each *subclass* is defined as a "kind of" the superclass, so that it can then inherit the common features. The programmer need only specify the behavior that sets the subclass apart from its parent and its siblings. Java programmers say that the subclasses are derived from the superclass, and refer to them as *derived* types and *base* types, respectively.

Programmers conceptualize programs both upward and downward, through levels of abstraction during the design and implementation stages of a development cycle. However, it is observed that they usually tend to think predominantly from the bottom up during the design phase, and predominantly top down during the implementation phase. Once the characteristics of a superclass are well described, its interface can be written before the subclasses are developed. Each subclass inherits both its superclass's interface and its implementation. It modifies and extends the former, or extends only the latter, as needed. The use of inheritance produces a hierarchy of classes.

For example, it is relatively easy to proceed downward from a base class like `bank-account` to more specialized classes of accounts, provided that the analysis and design phases are complete. By contrast, it is much more difficult proceeding upward, i.e., to try and code many different kinds of bank accounts first, and then coordinating all their differing interfaces into a superclass.

## Example of Inheritance

An example of inheritance where a programmer might identify a need for two types of text objects would be `CompactText` and `WordProcText`. `CompactText` emphasizes storage efficiency rather than functionality and `WordProcText` is used for word processing that contains font information. While the differences between the text objects are significant enough to warrant implementing them as different classes, the similarities that exist between them suggest the presence of certain shared properties and the possibility of their belonging to one superclass.

In this case, rather than creating two largely similar class definitions, a superclass called `TextObject` can be first defined. This establishes a common interface for the two. From this base class, the classes `CompactText` and `WordProcText` are derived. Each derived type inherits its interface and implementation from the base type. For example, all of the classes have a `length()` method. The base class might or might not provide a default implementation for this method. If the base class does provide a default implementation, the derived classes can inherit the implementation as their own. A derived class can override or extend the implementation of an inherited method as appropriate. A derived class may not change the interface of a method. `CompactText` might use the implementation of the `length()` method defined in `TextObject`, while `WordProcText` might override this implementation with its own.

Perhaps it is only later that an `EncodedText` type is needed. If an inheritance relationship exists, then implementing the new type becomes much simpler. Whether or not derived classes are created at the same time as their base class, each of the derived types extends the interface or implementation of the base class.

Establishing a sensible hierarchy of related types is one of the most challenging aspects of object-oriented program design. It is not easy to generalize a base type in a manner that not only abstracts the common behavior of the types already known, but also provides a good basis for the later derivation of new types. An important reward for utilizing inheritance is a dramatic increase in code reuse, which can save a programmer time and valuable resources.

## Derivation Syntax

The keyword `extends` in the subclass definition below declares one class as a subclass of another. A Java class may derive from at most one base class.

```
public class TextObject
{
    public int length () { ... }
}

public class CompactText
    extends TextObject
{
}

public class WordProcText
    extends TextObject
{
}
```

Any user-defined class that does not explicitly derive from another class derives from the library class `java.lang.Object`. The `Object` class is therefore the ultimate base class of all other Java classes. The `Object` class provides the `equals()` method as well as other features common to all Java objects.

## Effects of Inheritance

In Java, private methods of the base class are not inherited (the derived class cannot call them anyway), but everything else is inherited.

A derived-type object can be said to have a dual identity. It is a type of both its new class and its parent type. For example, if class `Bass` is derived from `Fish`, an object of type `Bass` is both a `Bass` and a `Fish`. However, the reverse is not necessarily true. The programmer, in this case, must force a conversion from base class type to derived class type with an explicit cast:

```
BaseClass Obj1; /*...*/
DerivedClass Obj2=(DerivedClass) Obj1;
```

## PROTECTED ACCESS

**NOTE**

*Since Java has the concept of a package, (a grouping of classes), Java offers a few additional combinations of access options as compared with C++.*

In addition to `public`, `private` and default (package-friendly) member access, there are two other access specifiers that apply only to derived classes. Any member that is declared `protected` is accessible only by a class that is derived from the current class or is in the same package as the containing class. Any member that is declared `private protected` is accessible only by a class that is derived from the containing class.

Consider the following example:

```
public class A
{
    protected int x;
    private protected int y;
}
```

Member `x` is accessible to any class that derives from class `A` and to any class in the same package as class `A`. Member `y` is only accessible to any class that derives from class `A`.

Figure 5 below shows the relationship between classes with different access specifiers.

FIGURE 5
*Accessibiliy in an inheritance relationship*

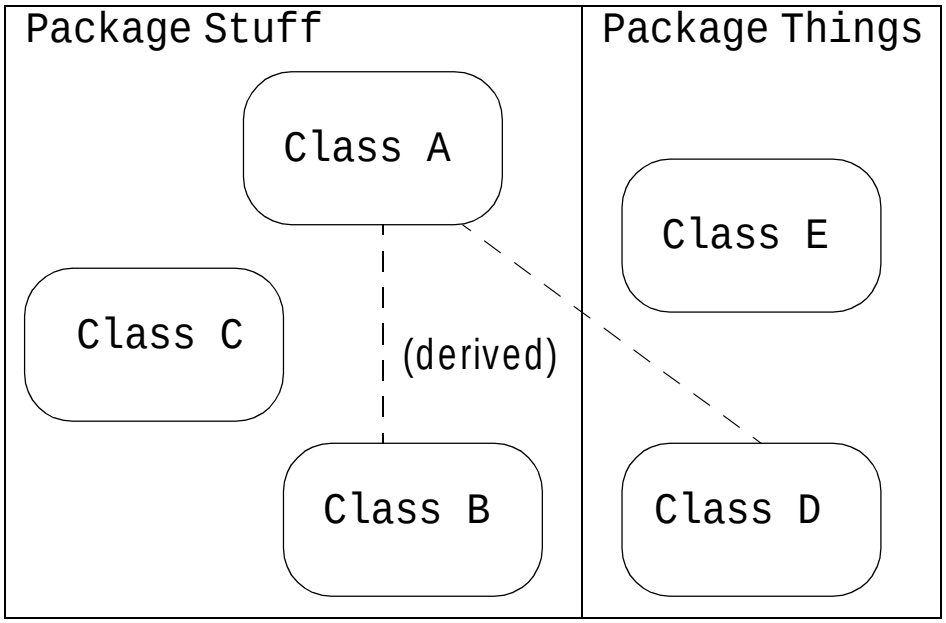



Table I details access to class A's members.

TABLE I. *Access to class A's members*

| Access specifier | Accessible by classes: |
|---|---|
| `public` | A, B, C, D, E |
| `protected` | A, B, C, D |
| default | A, B, C |
| `private protected` | A, B, D |
| `private` | A |

## OVERRIDING METHODS

The process of a derived class redefining a method contained in the parent class (with the same parameter types) is called *overriding* the method. Whenever a method is called for a derived-type object, the compiler calls the overriding version instead of the inherited version. The method is thus

implemented in a way that is unique to the derived class. The following is a syntax example:

```
public class TextObject
{
    public int length ()
    {
        // TextObject's implementation
    }
}

public class CompactText
    extends TextObject
{
    public int length ()
    {
        // CompactText's implementation
    }
}
```

This allows a programmer to use a parent type variable to point to a derived type object, call a method for that object, and have the compiler call the specialized version. This is how polymorphism is implemented. C++ programmers will recognize this as a *virtual function*. In Java, all methods are *virtual* unless specified otherwise.

## Dynamic Method Dispatching

Consider the following example:

```
class Fish { void eat(), other items... }

class Bass extends Fish { void eat()...}
class Shark extends Fish { void eat()...}

Fish redfish, bluefish;
redfish = new Bass();
bluefish = new Shark();

redfish.eat();
bluefish.eat();
```

The compiler cannot tell at compile-time which method to call here. The solution entails waiting until run-time to determine which function is the

correct one. In other words, the run-time code must check the actual type rather than the apparent type of an object based on the name of the variable used in the code. This is known as *dynamic method dispatching*.

## Polymorphism

Dynamic method dispatching is a major component of *polymorphism*. Polymorphism is a very powerful feature of object-oriented languages. It allows a program to treat objects of different types as if they were all of the same type.

**TRIVIA**

*The key difference here, compared to a traditional non-polymorphic approach, is that differences in behavior are internalized in different derived classes. This differs from an external body of code selecting a certain behavior from a list of options.*

The creator of a group of related data types designs them to respond to the same message in ways that are nominally identical, no matter how different they may be in detail. The code which defines and uses objects of these types can then send the same message to all of them and get an appropriate response from each.

A user program can create heterogeneous lists of objects of related types and invoke a virtual function for each one of them. At run time, the program will branch to the version of the function that is appropriate to that object.

This is accomplished by the following manner. When a class declares a member function, the compiler includes a type code in every object of that class. Wherever the function is invoked, the Java run-time system checks the type of each object and invokes the function appropriate to that object's specific class. By automating type checking, polymorphism eliminates the need for many of the large `switch` statements that tend to infest C programs. Coding these is notoriously tedious, mechanical, and error-prone. Such statements also require maintenance whenever a new class of data is introduced, even if that class is closely related to others.

In contrast, the addition of a new data type typically has much less effect on programs that use polymorphism. Since code that requests a string's length pays no attention to the particular string type, the compiler neither knows nor cares that some of the string types were introduced *after* the code was written. Such code therefore requires less initial development effort as well as less maintenance.

The following polymorphism example invokes the `length()` method of each element of an array of `TextObject`s:

```
TextObject[] tarr =
{
    new EncodedText (),
    new WordProcText (),
    new CompactText (),
    new TextObject ()
};

int totalLen = 0;

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

The code that invokes the `length()` method ignores type differences. New types of `TextObject`s can be introduced without forcing changes in calling code.

## The **super** Keyword

It is often desirable for an overriding implementation of a method to call the inherited implementation of the method. Often, the inherited code need not be completely replaced, only augmented to add some new actions. Any method in a derived class that overrides a method in the base class can call the corresponding base class method by using the `super` keyword as shown below:

```
public class WordProcText
    extends TextObject
{
    public int length ()
    {
        int len = super.length ();
        ...
        return len;
    }
}
```

Note that there is no way to access a method overridden more than once. The following code is illegal:

```
super.super.f()
```

## `Final` Methods and `Final` Classes

**NOTE**

*`final` in Java is the counterpart of `virtual` in C++. In C++ you need to specify `virtual` to get dynamic binding. In Java you need to specify `final` to get static binding.*

Dynamic lookup takes time. Therefore, there are some situations where it should be disabled. Also, there are some methods for which the compiler must be able to resolve a required action, or for which an overriding version would be too complicated to implement.

Methods labeled `final`, `private`, or `static` are not subject to dynamic lookup because they may not be overridden. `private` methods are simply not inherited because they would never be callable anyway. `static` methods apply to a particular class's static data and thus make no sense in a derivation. `final` methods are those designated as not-overrideable for reasons of complexity or safety.

## CONSTRUCTOR CHAINING

When a derived-type object is instantiated, it is sometimes desirable to have its constructor call a particular overload of its parent class. This is done with what looks like a call to the method `super()`. Often this is used to pass a particular variable up into a parent class's constructor. This must be the first statement in a constructor.

If the programmer does not include a `super()` call, the compiler will provide one. The only exception to this occurs when a constructor calls another constructor via the form `this()`. In this case, the compiler knows that the first constructor will call `super()`.

Constructors do not use dynamic lookup simply because each class has its own constructor.

Consider the following two classes:

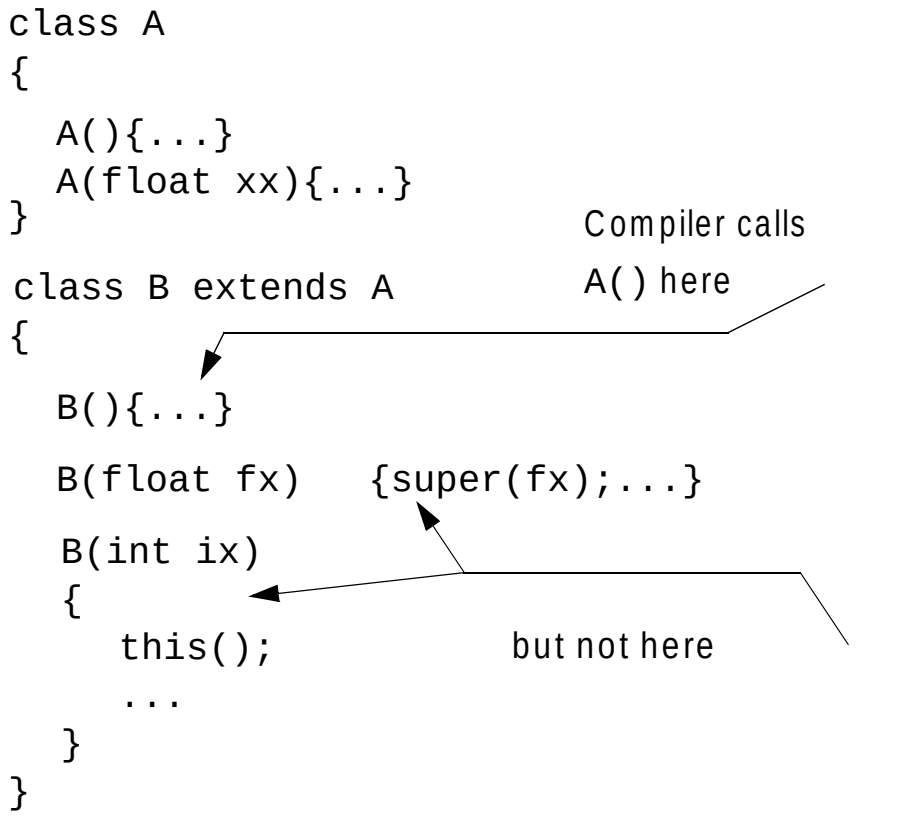
```
public class A
{
    A(int ix){...}
    A(float fx){...}// overloaded
}

public class B extends A
{
    B(int ix){...}
}
```

The keyword `super` can be used to specify which of A's constructors to call from B's constructor:

```
class B extends A {
{
    B(int ix)
    {
        super(ix);// calls A(int ix)
    }
}
```

This use of `super()` is legal only as the first statement of a constructor. If a constructor calls neither `super()` nor `this()`, the compiler implicitly calls the base class's *no-parameter* constructor as seen in Figure 6.

FIGURE 6
*Constructor Chaining*




One common mistake is to define a particular constructor in a parent class, and then derive from it without calling `super()`. In this case, the compiler would call the parent class's no-parameter constructor, but would not have defined one. This can produce some mystifying error messages.

The appropriate overload must exist in the parent class. Recall that if any constructor is written, the compiler will not provide the default no-parameter version.

The following will produce a compiler error:

```
class A {
    int value;
    A(int input) { value = input; }
}
class B extends A {
    int myvalue;
}
```

## INHERITANCE AND FINALIZATION

**NOTE**

*in C++, a parent class's destructor is called automatically. This does not happen in Java.*

If a parent class contains a `finalize()` method, it must be called explicitly by the derived class's `finalize()` method. Unlike C++, Java destructors are not called automatically. If it is needed, the derived class's `finalize()` method must call it manually with:

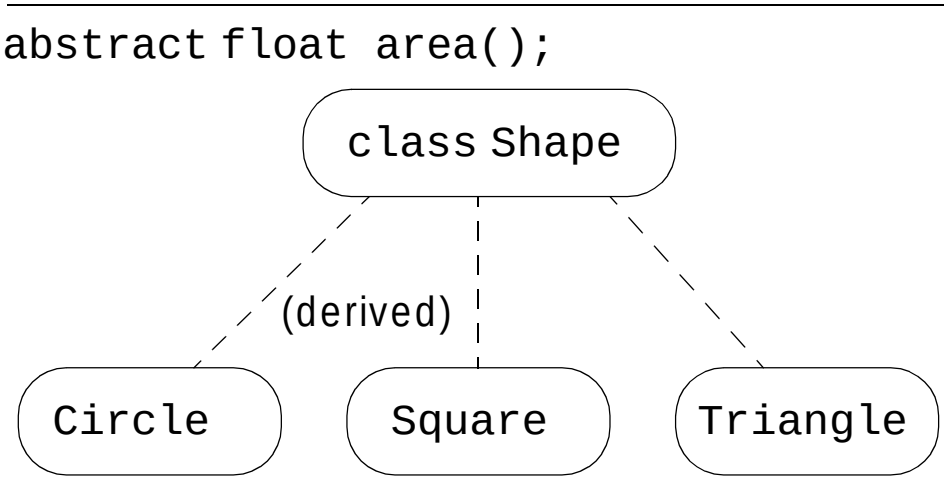
```
super.finalize();
```

## ABSTRACT CLASSES

Methods declared with the keyword `abstract` define a skeleton for a method but do not implement it. This requires a derived class to provide the code for this class. A class with one or more abstract methods must itself be declared abstract and cannot be instantiated. It can only be used for derivation.

C++ programmers will recognize this as a *pure virtual function* that makes an abstract class. A method can be declared abstract with no brackets to enclose any code.

FIGURE 7
*The relationship between an abstract class and its derived classes*




A generic shape cannot exist. Each subtype implements its own `area()` method.

## INTERFACES

**TIP**

*The rule of thumb in choosing between an abstract class and an interface is as follows: if you want any code in the base class to be reused by derived classes, use an abstract class. Otherwise use an interface.*

Some languages support *multiple inheritance* in which a child class may derive some of its features from one parent class and other features from another (or others). There is a continuing controversy over whether the benefits promised by this feature justify the extra complexity that comes with it. Java does not support multiple inheritance. However, it does provide a mechanism called an *interface* whereby the properties of one class can be used by another without formal inheritance. This enables Java to avoid most of the syntactical complications that multiple inheritance requires. When a parent class is to be defined as abstract, instead of defining one or more methods as abstract, the class can be defined with the keyword `interface` instead of `class`. This defines the minimum set of methods a class must contain.

An interface type is named for derivation with the keyword `implements` instead of `extends`. All of its methods are implicitly `public` and `abstract`. If desired, they can be declared that way for clarity. Also, any data defined in an interface must be constant (i.e., `final static`).

## The `implements` Declaration

A derived class cannot have two parent classes, as multiple inheritance leads to problems of how to decide which method to use for inherited methods. Yet multiple inheritance is desirable because it lets an object be assigned into, and thus used as, variables of different types. For example, a programmer might instantiate a `Circle` as a class derived from `Shape`, yet wish to pass it into a method requiring a `Graphic` object parameter.

The `implements` declaration in Figure 7, seen as a way of deriving from an `interface`, also means that the new class can be treated as being of that new type. Though a class can have only one parent class, it can implement any number of classes, and thus behave as more than one type of object. For example:

```
interface Shape{...}

class Circle extends Drawable
    implements Shape{...}

class Square extends Drawable
    implements Shape{...}

Shape sp1 = new Circle(),
        sp2 = new Square();

sp1.area();
sp2.area():
```

The classes `Circle` and `Square` are derived from class `Drawable`, but by implementing `Shape`, they also inherit `Shape`'s interface and are enabled to act as polymorphic instances of `Shape`.

With `interface` and `implements`, Java supports the polymorphic features of multiple inheritance, while yet avoiding the complications of methods implemented in multiple ways.

## CASTING BETWEEN CLASS TYPES

The Java run-time system keeps track of the type of each object. While this implies some memory space overhead for each object, the programmer can achieve a type-safe run-time environment. The Java language makes some of

its run-time typing power available to the programmer through the `instanceof` operator. The form of an `instanceof` expression is as follows:

```
object-expression instanceof class-name
```

The left-hand term is any expression that evaluates to a class type. The right-hand term is the name of a class or interface. An `instanceof` expression returns true if the actual type of the object on the left-hand side is of the type named by the right-hand side, and false otherwise. The object is of the named type if it is of exactly the named type or is of a type that directly or indirectly extends the identified type.

The `instanceof` operator can be use for safe casting between class types:

```
if (tobj instanceof CompactText)
{
    CompactText ct = (CompactText) tobj;
}
```

# LESSON SUMMARY

In this lesson, you have learned:

- Inheritance involves defining a class as a specific type of another class.
- Inheritance increases code reuse and allows for polymorphism.
- Java supports dynamic method dispatch at run-time.
- Methods and classes may be declared `final`.
- Abstract methods have no implementation.
- Interface types support limited multiple inheritance.
- Java's run-time typing allows safe casting between class types.

## REVIEW QUESTIONS

1. Write the class declarations for the following relationship, assuming that all classes are public: a Bulldog is a kind of Dog, and a Dog is a kind of Animal.

2. What is the difference between the `protected` and `private protected` access methods?

3. What is purpose of the `super` keyword?

*Answers on page 198*

## EXERCISE

1. Create an interface called `SecondObserver` that defines a single method, `tick()`. Go back to your `Clock` class and make class `Clock` implement the `SecondObserver` interface.

   Create a new class called `PrintClock` that extends your `Clock` class. `PrintClock` provides a method called `toString()` that produces a string representation of the current time. Test your class.

2. Create a new class called `TickingPrintClock` that extends your `PrintClock` class. `TickingPrintClock` extends the `tick()` method to print the new time on each tick, in addition to the normal function of the method.

   Test your new class. You can insert the following code block to cause your program to sleep for one second:

```
try
{
    Thread.sleep (1000);
}
catch (InterruptedException ie)
{
}
```

# LESSON 8

## Writing Java Applets

### OVERVIEW

*Java applets are graphical mini-programs that run in the context of another program. Thanks to the growth of the Internet, applets have become popular additions to World Wide Web pages. This lesson introduces basic applet methods and the means to embed applets in HTML documents.*

### LESSON TOPICS

- What Is an Applet?
- The `Applet` Class
- The Delegation Event Model—Action Events
- The `paint()` Method
- The `Graphics` Class
- Java Fonts
- Drawing Lines and Shapes
- Drawing with Color
- The `Color` Class

# OBJECTIVES

By the end of this lesson, you should be able to:

- Create a Java applet.
- Embed an applet in an HTML document.
- Use graphics library methods to draw in applets.
- Use the event model to capture click type events.

# WHAT IS AN APPLET?

**TRIVIA**

*The applets make static documents come alive. This feature of Java was a major reason for its quick popularity as an internet language.*

An *applet* is not a standalone application. It is a mini program invoked within a larger program, such as an applet viewer or a Java-enabled World Wide Web (WWW) browser. An applet depends on the viewer or browser to provide a context in which to run. This means that the viewer or browser implements common applet functions and provides those functions to the applet by way of a *Java Application Program Interface (Java API)*. The viewer or browser also controls the execution of the applet.

An applet is graphical by nature. Each applet occupies a rectangular panel in its viewer's window. Applets are meant to be embedded within a graphical document, such as an HTML document.

## A "Hello, World" Applet

The following is a simple applet named `HelloWorldApplet.java`:

**NOTE**

*Without the `import` statement, use `java.applet.Applet` instead of `Applet`; and `java.awt.Graphics` instead of `Graphics`.*

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

public class HelloWorldApplet extends Applet
{
    public void paint (Graphics g)
    {
        g.drawString (“Hello World”, 25, 50);
    }
}
```

These import statements bring the classes into the scope of our applet class:

- `java.applet.Applet`
- `java.awt.Graphics`

Without those import statements, the Java compiler would not recognize the classes `Applet` and `Graphics`, which the applet class refers to.

## THE `Applet` CLASS

Every applet is an extension of the `java.applet.Applet` class. The base `Applet` class provides methods that a derived `Applet` class may call to obtain information and services from the browser context. These include methods that do the following:

- get applet parameters
- get the network location of the HTML file that contains the applet
- get the network location of the applet class directory
- print a status message in the browser
- fetch an image
- fetch an audio clip
- play an audio clip
- resize the applet

Additionally, the `Applet` class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may:

- request information about the author, version and copyright of the applet
- request a description of the parameters the applet recognizes
- initialize the applet
- destroy the applet
- start the applet's execution
- stop the applet's execution

The `Applet` class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the `paint` method.

## Invoking an Applet

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser. The `<applet>` tag is the basis for embedding an applet in an HTML file. Below is an example that invokes the "Hello, World" applet:

```
<html>
<title>The Hello, World Applet</title>
<hr>
<applet code="HelloWorldApplet.class" width=320
height=120>
If your browser was Java-enabled, a "Hello, World"
message would appear here.
</applet>
<hr>
</html>
```

The `code` attribute of the `<applet>` tag is required. It specifies the `Applet` class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. Note that the size of the applet

panel is determined in the document, not from within the applet. An applet may call its `resize()` method, but the browser is not guaranteed to update the display. The applet directive must be closed with a `</applet>` tag.

If an applet takes parameters, values may be passed for the parameters by adding `<param>` tags between `<applet>` and `</applet>`. The browser ignores text and other tags between the applet tags. Non-Java-enabled browsers do not process `<applet>` and `</applet>`. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers. The programmer can take advantage of this feature to provide filler content for users with older browsers.

The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the `codebase` attribute of the `<applet>` tag as shown:

```
<applet codebase="http://not.a.real.site/applets"
code="HelloWorldApplet.class" width=320 height=120>
```

Whether a code base is specified or the default is used, it is important to keep in mind that the code base is the root directory of the package tree. If the code is organized into packages, the Java class loader expects to find the class files in like-named subdirectories of the code base directory. The code base directory itself is the home of the default package.

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (`.`) to separate package/class components. For example:

```
<applet code="mypackage.subpackage.TestApplet.class"
width=320 height=120>
```

## Getting Applet Parameters

The following example demonstrates how to make an applet respond to setup parameters specified in the document. This applet displays a *checkerboard* pattern of black and a second color. The second color and the size of each square may be specified as parameters to the applet within the document.

`CheckerApplet` gets its parameters in the `init()` method. It may also get its parameters in the `paint()` method. However, getting the values and saving the settings once at the start of the applet, instead of at every refresh, is convenient and efficient. The applet viewer or browser calls the `init()` method of each applet it runs. The viewer calls `init()` once, immediately after loading the applet. (`Applet.init()` is implemented to do nothing.) Override the default implementation to insert custom initialization code.

The `Applet.getParameter()` method fetches a parameter given the parameter's name (the value of a parameter is always a string). If the value is numeric or other non-character data, the string must be parsed.

The following is a skeleton of `CheckerApplet.java`:

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

public class CheckerApplet extends Applet
{
    int squareSize = 50;// initialized to default size

    public void init () {}

    private void parseSquareSize (String param) {}

    private Color parseColor (String param) {}

    public void paint (Graphics g) {}
}
```

Here are `CheckerApplet`'s `init()` and `private parseSquareSize()` methods:

```
public void init ()
{
     String squareSizeParam = getParameter (“squareSize”);
     parseSquareSize (squareSizeParam);
     String colorParam = getParameter (“color”);
     Color fg = parseColor (colorParam);
     setBackground (Color.black);
     setForeground (fg);
}

private void parseSquareSize (String param)
{
     if (param == null) return;
     try {
            squareSize = Integer.parseInt (param);
     }
     catch (Exception e) {
         // Let default value remain
     }
}
```

The applet calls `parseSquareSize()` to parse the `squareSize` parameter. `parseSquareSize()` calls the library method `Integer.parseInt()`, which parses a string and returns an integer. `Integer.parseInt()` throws an exception whenever its argument is invalid. Therefore, `parseSquareSize()` catches exceptions, rather than allowing the applet to fail on bad input.

The applet calls `parseColor()` to parse the color parameter into a Color value. `parseColor()` does a series of string comparisons to match the parameter value to the name of a predefined color. For more information concerning `parseColor()`, refer to the sample library.

## Specifying Applet Parameters

The following is an example of an HTML file with a `CheckerApplet` embedded in it. The HTML file specifies both parameters to the applet by means of the `<param>` tag.

```
<HTML>
<TITLE>Checkerboard Applet</TITLE>
<HR>
<APPLET CODE="CheckerApplet.class" WIDTH=480
HEIGHT=320>
<PARAM NAME="color" VALUE="blue">
<PARAM NAME="squareSize" VALUE="30">
</APPLET>
<HR>
</HTML>
```

Parameter names are not case sensitive.

# THE DELEGATION EVENT MODEL—ACTION EVENTS

Events, such as the clicking of a button or the movement of a scroll bar, can be intercepted and processed by an applet. The technique used by Java to do this is called the *Delegation Event Model*. This model uses the concept that an *object* (e.g., a command button) is the source of events; and another object (e.g., an applet) *listens* to a source for a specific type of event.

A listener informs the source that it is interested in listening to that source. The source adds that listener to its list of objects. When an event occurs, the source informs all interested listeners, and passes information regarding the event to the listener.

If a listener is interested in an event it must implement a set of methods that the source can call when the event occurs. This set of methods is called an *interface*. The concept of an interface will be discussed in greater depth later in the course. Having implemented the interface, the source is then able to call the listener. When a method of the interface is called, a single argument or an event object is passed that conveys information about the event.

The essential steps required for handling events include:

1. Use of the `implements` keyword by the listener.
2. Adding the listener to the source's event list.
3. Implementing the appropriate methods.
4. Using the event object to process the event.

*Step 1*: Implement the listener:

```
...
import java.awt.event.*;
public class MyApplet extends Applet implements
ActionListener
{
```

The `implements` keyword is used in the class definition to inform the java compiler that this class supports a specific type of interface. There are several types of event interfaces supported by Java. The `ActionListener` and `AdjustmentListener` interfaces are described in this section. The `ActionListener` interface deals with high level events such as the pressing of a button. The `AdjustmentListener` is used with controls such as a scroll bar. Other high and low level interfaces exist.

*Step 2*: Add the listener to the source's event list:

```
public void init ()
{
    Button b = new Button (“Help”);
    b.addActionListener(this);
    add (b);
}
```

The `addActionListener` method is used to add the source (the current applet) to the list of listeners interested in listening to the button. The `this` keyword identifies the current applet as the listener. The method is applied to the button thus making the button the source object.

*Step 3*: Implement the appropriate method:a

```
public void actionPerformed(ActionEvent evt)
{
```

The `ActionListener` interface has only a single method, `actionPerformed`. When an interface is implemented by a class, all of its methods must be overridden by that class. The `actionPerformed` method is passed as a single argument, `ActionEvent`. This argument contains information about that event.

*Step 4*: Process the event:

```
if (evt.getSource() instanceof Button)
{
    Button source = (Button)evt.getSource();
    if (source.getLabel() == “Help”)
      / * ... handle “Help” button ... */
```

The `actionPerformed` method may be invoked by more than one source if the listener has added itself to other sources. One of the primary purposes of the `ActionEvent` object is to differentiate between different sources. The `getSource` method returns a reference to the object that caused the event. It is used with such methods as `getLabel`, that returns the button label, to determine the source of the event.

## The **`ActionEvent`** Class

The `ActionEvent` class contains the event type `ACTION_PERFORMED`. An `ActionEvent` can be fired by:

- clicking a button
- double clicking a list
- choosing a menu item
- pressing `<enter>` when in a text field

The `getSource()` method is used to determine the object in which the event originated.

An action command is associated with each `ActionEvent`. This command is a simple `String` that is assigned to a source object. It is assigned when the object is created using the s`etActionCommand` method. Later, the `getActionCommand` method can be used within such methods as `actionPerformed`, to uniquely identify the source of the event. This

technique avoids having to use a label, or some other attribute of the source, that may change with the use of the object.

The user may hold down a modifier key such as <ALT> or <CTRL> when an event is fired. The `getModifiers()` method can be used to determine which key was held down. The following masks are provided with the `ActionEvent` class for comparison with the modifier constants:

- `ALT_MASK`
- `CTRL_MASK`
- `META_MASK`
- `SHIFT_MASK`

## Adjustment Events

Adjustment events are fired when the user manipulates an `Adjustable` item, such as a scrollbar or slider control. They are handled similarly to `Action` events.

To respond to a scrollbar manipulation, implement the `adjustmentValueChanged` method of the `AdjustmentListener` interface, and query the `AdjustmentEvent` object:

```
...
import java.awt.event.*;
public class MyApplet extends Applet implements
AdjustmentListener
{
    public void init ()
    {
        Scrollbar sb = new Scrollbar ( ... );
        sb.addAdjustmentListener(this);
        add (sb);
    }

    public void adjustmentValueChanged (AdjustmentEvent evt)
    {
        switch (evt.getAdjustmentType ())
        {
           case AdjustmentEvent.UNIT_INCREMENT:
           case AdjustmentEvent.UNIT_DECREMENT:
           case AdjustmentEvent.BLOCK_INCREMENT:
           case AdjustmentEvent.BLOCK_DECREMENT:
           case AdjustmentEvent.TRACK:
             Scrollbar sb = (Scrollbar)evt.getAdjustable();
             int orientation = sb.getOrientation ();
             int value = sb.getValue ();
             /* ...update display to match value */
        }
    }
}
```

### *The **`AdjustmentEvent`** Class*

The `AdjustmentEvent` class contains the event type `ADJUSTMENT_VALUE_CHANGED`. This event can be fired by:

- manipulating a scrollbar
- manipulating a custom component (such as a slider)

The `getAdjustable()` method returns a reference to the component that fired the event. If that type of adjustment is needed, the `getAdjust-`

ableType() method will return one of the following values which are defined in the AdjustableEvent class:

- BLOCK_DECREMENT
- BLOCK_INCREMENT
- TRACK
- UNIT_DECREMENT
- UNIT_INCREMENT

## THE paint() METHOD

The Applet class derives a class within Java's Abstract Windowing Toolkit (AWT) library, java.awt.Panel. The Java AWT forms a layer between the viewer or browser and the applet running within it. Every graphical component in the AWT library, including Panel, derives from java.awt.Component. The Component class supports a set of methods for drawing, positioning, sizing, etc. The paint() method is one of those common methods.

The Java AWT calls the paint() method to completely render the component. For example, when the window containing the component is resized, it may be necessary to *repaint* certain window components. The default implementation of paint() does nothing. To draw custom graphical content in your applet, you must override the paint() method. Its signature is:

```
public void paint (Graphics g);
```

## THE Graphics CLASS

The paint() method takes one parameter, an object of class Graphics. The object holds a *context* in which drawing operations may be performed. The Graphics class provides the methods for drawing; including drawLine() (the method that draws a line), and drawString() (the method that draws text). The paint() method in the "Hello, World" applet calls

`Graphics.drawString()` to draw the text string "Hello, World" at position (25, 50) within the applet's panel:

FIGURE 8
*String placement in Graphics Class*

The `Graphics` object clips all graphics operations to within the current draw region.

As shown in the diagram above, the `x` coordinate of an AWT graphics point represents the number of pixels horizontally from the left of the panel. The `y` coordinate represents the number of pixels vertically from the top.

## JAVA FONTS

A Java font is represented by an object of class `java.awt.Font`. A `Font` object has three attributes:

- family
- style
- size

Java provides five font families, each of which maps to a true font on the host platform. A font family is identified by a string:

- `"Helvetica"`
- `"TimesRoman"`
- `"Courier"`
- `"Dialog"`
- `"DialogInput"`

There is also a default font family to be used in the event that a font is missing.

Java provides plain, bold, and italic font styles. Bold and italic may be combined, resulting in four styles, listed in Table J.

TABLE J. *Java Styles and Their Symbols*

| Style | Symbol |
|---|---|
| Plain | Font.PLAIN |
| Bold | Font.BOLD |
| Italic | Font.ITALIC |
| BoldItalic | Font.BOLD \| Font.ITALIC |

The third attribute of a font is its point size.

## Selecting a Font

To select a font, the programmer must:

- construct a `Font` object
- install a `Font` object into the `Graphics` context

The `Font` constructor takes three arguments:

- the family (a string)
- style
- point size

For example:

```
Font myFont = new Font ("Helvetica", Font.PLAIN, 18);
```

To install a font into a `Graphics` context, use the `Graphics.setFont()` method:

```
public class NewHelloWorldApplet extends Applet
{
    static Font myFont =
        new Font ("Courier", Font.PLAIN, 18);

    public void paint (Graphics g)
    {
        g.setFont (myFont);
        g.drawString ("Hello, World", 25, 50);
    }
}
```

## DRAWING LINES AND SHAPES

Below is an example of an applet (called `ShapesApplet.java`) that demonstrates some of the drawing functions available through class `Graphics`.

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

public class ShapesApplet extends Applet
{
    public void paint (Graphics g)
    {
        int left = 0;
        int top = 0;
        int right = 319;
        int bottom = 119;

        g.drawRect (left, top, right - left, bottom - top);
        g.drawLine (left, top, right, bottom);
        g.drawLine (right, top, left, bottom);
        g.drawOval (left, top, right - left, bottom - top);
    }
}
```

The arguments to `drawLine()` are the (`x,y`) coordinates of the first endpoint and the (`x,y`) coordinates of the second endpoint. `drawLine()` draws a line one pixel wide from the first endpoint to the second endpoint, inclusive.

The arguments to `drawRect()` are: the `x` coordinate of the left edge of the rectangle, the `y` coordinate of the top edge of the rectangle, the width of the rectangle in pixels, and the height of the rectangle in pixels

`drawOval()` draws an oval within the rectangle specified by its parameters, which are similar in meaning to those of `drawRect()`.

## DRAWING WITH COLOR

The following example is another applet that demonstrates the use of color for graphics.

```
ColorsApplet.java:

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

public class ColorsApplet extends Applet
{
    public void paint (Graphics g)
    {
        int left = 10;
        int top = 50;
        int width = 50;
        int height = 50;

        g.setColor (Color.red);
        g.fillRect (left, top, width, height);

        left += 100;
        g.setColor (Color.yellow);
        g.fillRect (left, top, width, height);

        left += 100;
        g.setColor (Color.blue);
        g.fillRect (left, top, width, height);
    }
}
```

This applet displays three square tiles of various colors. The method `Graphics.setColor()` is used to set the current color.

## THE `Color` CLASS

The method `Graphics.setColor()` installs a color into a graphics context. The color setting remains until another call to `setColor()`. At the start of the `paint()` method, the color installed in the graphics context is the foreground color of the applet, usually black.

`color.red`, `color.blue`, etc., are constant objects of type `color`, defined as static data members in class `Color`. The entire list of predefined colors appears in Table K.

TABLE K. *RGB Color Values*

| Color | Red Value | Green Value | Blue Value |
|---|---|---|---|
| white | 255 | 255 | 255 |
| lightGray | 192 | 192 | 192 |
| gray | 128 | 128 | 128 |
| darkGray | 64 | 64 | 64 |
| black | 0 | 0 | 0 |
| red | 255 | 0 | 0 |
| pink | 255 | 175 | 175 |
| orange | 255 | 200 | 0 |
| yellow | 255 | 255 | 0 |
| green | 0 | 255 | 0 |
| magenta | 255 | 0 | 255 |
| cyan | 0 | 255 | 255 |
| blue | 0 | 0 | 255 |

If the desired color is not one of the 13 predefined `color` objects, a new `color` object can be constructed. The `color` constructor constructs a new `color` object from RGB values in the range (0, 255). For example,

```
Color c = new Color (255, 128, 0);
```

It is sometimes more convenient to express a color in terms of *HSB* (hue, saturation, brightness) values. A static method `Color.getHSBColor()` exists for this purpose:

```
c = Color.getHSBColor (0.5, 0.7, 0.2);
```

HSB values are floating point numbers in the range (0.0, 1.0).

Although these methods allow you to construct millions of different color values, the number of different colors displayable, at one time, is far fewer. The Java library internals map each color to the closest available color in the current model of the display device.

## Foreground and Background Colors

Because an applet is an Abstract Windowing Tookit (AWT) component, it can store two colors: foreground and background. By default, the foreground and background colors of the applet are inherited from the document. The colors are usually black on white or black on gray.

These colors may be accessed through the methods `getForeground()` and `getBackground()`, for example:

```
Color c = getBackground ();
```

The colors may be set through the methods `setForeground()` and `setBackground()`, for example:

```
setForeground (new Color (64, 64, 64));
```

Before calling the applet's `paint()` method, the Java AWT fills the panel with the applet's background color. (The viewer or browser calls the `init()` method before the first call to `paint()`.) At the start of the `paint()` method, the color installed in the graphics context becomes the applet's foreground color.

`CheckerApplet` takes advantage of this behavior in the `paint()` method. Because the Java AWT has already filled the panel with the background color (black), the `paint()` method does not need to draw the black squares. Because the foreground color is already installed in the graphics

context at the start of `paint()`, the paint method does not need to call `Graphics.setColor(). paint()` simply iterates over the visible squares, drawing a foreground square whenever the counter is even:

```
public void paint (Graphics g)
{
    Dimension sz = size ();
    for (int y = 0; y < sz.height; y += squareSize) {
        int count = y / squareSize;
        for (int x = 0; x < sz.width; x += squareSize) {
           if ((count % 2) == 0) {
             g.fillRect (x, y, squareSize, squareSize);
           }
           ++count;
        }
    }
}
```

The method `size()` returns an object of type `Dimension`, giving the current width and height of the applet's pane.

# LESSON SUMMARY

In this lesson, you have learned:

- An *applet* is a graphical mini-program that runs in the context of another program, such as an applet viewer or a Java-enabled WWW browser.
- The `<applet>` directive is used to embed an applet in HTML documents.
- The Java applet API allow programmers to:
  - draw text and graphics into an applet
  - work with fonts and colors

## REVIEW QUESTIONS

1. What is the fully qualified name of the base class of all applets?

2. What is the name of the method that fetches the value of an applet parameter?

3. Write the full HTML applet tag that starts an applet named `Shape.class` at a width of 200 and a height of 100. It also takes a color parameter equal to red and a size parameter equal to 10.

*Answers on page 198*

## EXERCISE

1. Modify the "Hello, World" applet to take four parameters named `text`, `font`, `style` and `size`. These specify what text to display, and what family, style and size of the font to use. Valid values for `style` include: plain, bold, italic, bolditalic. `size` is an integer, and must be parsed (see `CheckerApplet` for an example). If any of the parameters is unspecified, use a default value.

2. Implement an applet that contains a single button. When the user clicks on that button, display your name at location (100, 100) on the applet. The code used to illustrate the event handler earlier in this chapter can be used as the basis for this application. Add a string identifier to hold your name and initialize it to an empty string in the `init` method. In the `actionPerformed` method, assign your name to the string. The `repaint` method will force redrawing of the applet. Place this method right after your name has been assigned. Display the string in the paint method. Ensure that the following import statements are included at the beginning of the program:

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

# APPENDIX A

# Hypertext Markup Language (HTML)

## OVERVIEW

*This appendix contains a brief overview of some common HTML tags, as well as standard Netscape extensions. Many of the Netscape extensions have been submitted for inclusion in the HTML standard.*

# HTML HISTORY AND SGML

HTML is an application of the ISO certified Standard Generalized Markup Language (SGML) standard. SGML was first published in 1988 as a standard for electronic document exchange, archival and processing. There are other de facto standards that have the same objectives, such as Adobe's Acrobat, or Microsoft's RTF (Rich Text Format).

As a subset of SGML, HTML is much simplified. SGML documents are generally more complex and programming-like than HTML documents. You can use SGML to define HTML, which allows for greater standardization and interchangeability. HTML's advantage comes from its combination of SGML tags and contructs and standard English text markup notation. HTML is therefore easy to interpret, yet powerful enough for its purpose. The HTML standard is maintained by the Internet Engineering Task Force (IETF).

The next sections list some common HTML tags, and their usage.

## STRUCTURE

| Tag | Description | Comments |
|---|---|---|
| <HTML></HTML> | Document type | Should surround whole document |
| <TITLE></TITLE> | Document title | Should be in header |
| <HEAD></HEAD> | Document header | |
| <BODY></BODY> | Body | |
| <!-- *text* --> | Comment | (not displayed by the browser) |

## HEAD ELEMENTS

| Tag | Description | Comments |
|---|---|---|
| <ISINDEX> | Searchable Document | Adds search prompt, only works if document is setup for search |
| <BASE HERF="*URL*"> | Document's base URL | |
| <LINK REV="*text*"REL="*text*" | | |
| <HREF="*URL*"> | Relationship | |
| <META> | Meta information | |
| <NEXTID> | Identifier | Used by HTML editors, not humans |
| <ISINDEX PROMPT= "*text*"> | Prompt text | Netscape extension |

## FORMATTING: BLOCKS AND SEPARATORS

| Tag | Description | Comments |
|---|---|---|
| <H*n*></H*n*> | Headings | *n* may be a value from 1 to 6 |
| <Hn ALIGN=LEFT\|CENTER\|RIGHT> </H*n*> | Align heading | HTML 3 |
| <P></P> | Paragraph | Usually double line break. Closing tag is optional. |
| <P ALIGN=LEFT\|CENTER\|RIGHT> </P> | Align text | HTML 3 |
| <ADDRESS></ADDRESS> | Address field | Frequently in italics |
| <BLOCKQUOTE></BLOCKQUOTE> | Quotation | Usually indented |
| <PRE></PRE> | Preformatted | Display text as-is; line breaks are retained |
| <PRE WIDTH=n></PRE> | Width | *n* is the number of characters |
| <CENTER></CENTER> | Center | Netscape |
| <BR> | Link break | A single line break |

## FORMATTING: PHYSICAL

| Tag | Description | Comments |
|---|---|---|
| <B></B> | Bold | |
| <I></I> | Italic | |
| <S></S> | Strikeout | Not widely implemented |
| <U></U> | Underline | Not widely implemented |
| <TT></TT> | Typewriter | Monospaced font |
| <BLINK></BLINK> | Blink | Netscape |
| <FONT COLOR="color specs" SIZE=*n* FACE="typeface name"></FONT> | Font size, color, typeface | Netscape, *n*=1-7<br>HTML 3 |
| <BASEFONT SIZE=*n*> | Base font size | Netscape, *n*=1-7; default is 3 |

## LIST

| Tag | Description | Comments |
|---|---|---|
| <LI> | List Item | Use in UL, OL, MENU, and DIR |
| <UL></UL> | Unordered List | Use <LI> before each list item |
| <OL></OL> | Ordered List | Use <LI> before each list item |
| <MENU></MENU> | Menu List | Use <LI> before each list item |
| <DIR></DIR> | Directory List | Use <LI> before each list item |
| <DT> | Term | Used with Definition Lists (DL) |
| <DD> | Definition | Used with Definition Lists (DL) |
| <DL></DL> | Definition List | Use DT and DD, not LI |

## NETSCAPE LIST EXTENSIONS

| Tag | Description | Comments |
|---|---|---|
| <UL TYPE=DISC\|CIRCLE\|SQUARE> | Bullet type | Entire list |
| <LI TYPE=DISC\|CIRCLE\|SQUARE> | Bullet type | Current and following items |
| <OL TYPE=A\|a\|I\|i\|1> | Number type | Entire list |
| <LI TYPE=A\|a\|I\|i\|1> | Number type | Current and following items |
| <OL VALUE=*n*> | Initial number | Entire list |
| <LI VALUE=*n*> | Initial number | Current and following items |

## LINKS

| Tag | Description | Comments |
|---|---|---|
| <A HREF="*URL*"></A> | Link | |
| <A HREF="*URL#label*"></A> | Link to specific location | Destination is in another document |
| <A HREF="*#label*"></A> | Link to specific location | Destination is in current document |
| <A HREF=**mailto**:user@host | Link for mail send | Brings up the browser's mail applet with address set |
| <A HREF=**news**:newsgroup | Link to newsgroup | Brings up the browser's news reader and opens that group |
| <A NAME="*label*"></A> | Define location | |

## IMAGES

| Tag | Description | Comments |
|---|---|---|
| <IMG SRC=*"URL" attribute*> | Display Image | |
| IMG Attributes | | |
| ALIGN=TOP\|BOTTOM\|MIDDLE\|RIGHT\| LEFT\|ABSMIDDLE\|BASELINE\|TEXTTTOP | Alignment | some are HTML 3 |
| ALT="*text*" | Alternate text | Text to display if image is not displayed |
| BORDER="pixels" | Pixel size of border | |
| ISMAP | Imagemap | Requires a mapfile |

# FORMS

| Tag | Description | Comments |
|---|---|---|
| <FORM ACTION="*URL*" | | |
| METHOD=GET\|POST></FORM> | Define form | |
| Select | | |
| <SELECT *attributes*></SELECT> | Selection field | Pulldown selection list |
| Select Attributes | | |
| NAME="*value*" | Field label | *value* is the name of the input field |
| SIZE=*n* | #of options | Not implemented in all browsers |
| MULTIPLE | Multiple selections allowed | Can select more than one |
| <OPTION>*option text* | Option | Item that can be selected |
| <OPTION SELECTED>*option text* | Default option | |
| Text Area | | |
| <TEXTAREA *attributes*> </TEXTAREA> | Multiple line input | |
| Textarea Attributes | | |
| ROWS=*n*COLS=*n* | Size of box | |
| NAME="*value*" | Field label | *value* is the name of the input field |
| Input | | |
| <INPUT *attributes*> | Input field | |
| Input Attributes | | |
| TYPE="TEXT\|HIDDEN\|IMAGE\| PASSWORD\|CHECKBOX\|RADIO\| SUBMIT\|RESET" | Input type | |
| NAME="*value*" | Field label | *value* is the name of the input field |
| CHECKED | Sets default to checked | For use with checkboxes and radioboxes |
| SIZE=*n* | Box size in characters | For use with TEXT |
| MAXLENGTH=*n* | Maximum length for input | For use with TEXT |

## TABLES (HTML 3)

| Tag | Description | Comments |
|---|---|---|
| <TABLE></TABLE> | Define Table | |
| <TABLE BORDER></TABLE> | Table Border | Draw borders around table |
| Table Row <TR> | | |
| <TR *attribute*></TR> | Table Row | |
| TR Attributes | | |
| ALIGN=LEFT\|RIGHT\|CENTER | Horizontal alignment | |
| VALIGN=TOP\|MIDDLE\|BOTTOM | Vertical alignment | |
| Table Cell<TD>and Header<TH> | | |
| <TD *attribute*></TD> | Table cell | Must appear within table rows |
| <TH *attribute*></*TH*> | Table header | |
| TD and TH Attributes | | |
| ALIGN=LEFT\|RIGHT\|CENTER | Horizontal alignment | |
| VALIGN=TOP\|MIDDLE\|BOTTOM | Vertical alignment | |
| NOWRAP | No line breaks | |
| COLSPAN=*n* | Columns to span | |
| ROWSPAN=*n* | Rows to span | |
| <CAPTION></CAPTION> | Table caption | |
| <CAPTION ALIGN=TOP\|BOTTOM> | Location | Above or below table |

## MISCELLANEOUS NETSCAPE EXTENSIONS

| Tag | Description | Comments |
|---|---|---|
| <BODY BGCOLOR="*#nnn*" | Background color | order is red/green/blue |
| <BODY BACKGROUND="*URL*" | Background picture | |
| <BODY TEXT="*#nnn*"> | Text color | |
| <BODY LINK="*#nnn*"> | Link color | |
| <BODY VLINK="*#nnn*"> | Visited link | |
| <BODY ALINK="*#nnn*"> | Active link | |
| <BR CLEAR=LEFT\|RIGHT\|ALL> | Clear text wrap | |
| <NOBR> | No break | Prevents line breaks |
| <WBR> | Word break | Where line may be broken |

## JAVA APPLETS

| Tag | Description | Comments |
|---|---|---|
| <APPLET *attributes*></APPLET> | Run Java applet | |
| APPLET ATTRIBUTES | | |
| ALIGN={"left" \| "right" \| "top" \| "middle" \| "bottom" | Applet alignment | Tells browser where the applet should be placed relative to entire browser frame |
| ALT="*string*" | Default text | Text to show if browser doesn't support Java |
| CODE="*URL*" | Applet program | Applet program, ending with "class" |
| CODEBASE="*URL*" | Applet location | Needed if .class file not found in same directory as URL |
| HEIGHT=*integer* | Applet pixel height | |
| WIDTH=*integer* | Applet pixel width | |
| <PARAM *attributes*></PARAM> | | |
| Param Attributes | | |
| NAME="parameter name" | Applet param name | Applet will ask for the value for the param of this name |
| VALUE="parameter value" | Applet param value | This value is always specified and returned to program as a string |

## JAVA SCRIPT

| Tag | Description | Comments |
|---|---|---|
| <SCRIPT *attributes*></SCRIPT> | Run Script | Designed to support new scripts in the future |
| SCRIPT ATTRIBUTES | | |
| LANGUAGE="*JavaScript*" | Specifies which language is being used | Tells browser that the language code that follows is Java Script |
| function *func_name*(form) {**...**} | Script | The actual script is embedded at this location in the HTML document |

# APPENDIX B

## Java Sample

### OVERVIEW

*This appendix contains a sample Java applet. It comes from the demos supplied with the Java Development Kit. The applet can be viewed using any java-capable browser or using the Applet Viewer, which is part of the JDK release. To run this applet with the Applet Viewer, just specify the URL with the* `appletviewer` *command. See the course postings for a downloadable version of the applet.*

# USAGE AND COPYRIGHT NOTIFICATION

Copyright (c) 1994-1996 Sun Microsystems, Inc. All Rights Reserved.

Sun grants you ("Licensee") a non-exclusive, royalty free, license to use, modify and redistribute this software in source and binary code form, provided that i) this copyright notice and license appear on all copies of the software; and ii) Licensee does not utilize the software in a manner which is disparaging to Sun.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee represents and warrants that it will not use or redistribute the Software for such purposes.

# THE XYZAPP.JAVA SOURCE

```
/*
 * @(#)XYZApp.java1.3 96/12/06
 * A set of classes to parse, represent and display Chemical compounds in
 * .xyz format (see http://chem.leeds.ac.uk/Project/MIME.html)
 */
import java.applet.Applet;
import java.awt.Image;
import java.awt.Event;
import java.awt.Graphics;
import java.awt.Dimension;
import java.io.StreamTokenizer;
import java.io.InputStream;
import java.io.BufferedInputStream;
import java.io.IOException;
import java.net.URL;
import java.util.Hashtable;
import java.awt.image.IndexColorModel;
import java.awt.image.ColorModel;
import java.awt.image.MemoryImageSource;
/** The representation of a Chemical .xyz model */
class XYZChemModel {
    float vert[];
    Atom atoms[];
    int tvert[];
    int ZsortMap[];
    int nvert, maxvert;
    static Hashtable atomTable = new Hashtable();
    static Atom defaultAtom;
    static {
    atomTable.put("c", new Atom(0, 0, 0));
    atomTable.put("h", new Atom(210, 210, 210));
    atomTable.put("n", new Atom(0, 0, 255));
    atomTable.put("o", new Atom(255, 0, 0));
    atomTable.put("p", new Atom(255, 0, 255));
    atomTable.put("s", new Atom(255, 255, 0));
    atomTable.put("hn", new Atom(150, 255, 150)); /* !!*/
    defaultAtom = new Atom(255, 100, 200);
    }
    boolean transformed;
    Matrix3D mat;
    float xmin, xmax, ymin, ymax, zmin, zmax;
    XYZChemModel () {
    mat = new Matrix3D();
    mat.xrot(20);
    mat.yrot(30);
    }
```

```
    /** Create a Cehmical model by parsing an input stream */
    XYZChemModel (InputStream is) throws Exception
    {
       this();
       StreamTokenizer st;
          st = new StreamTokenizer(new BufferedInputStream(is, 4000));
       st.eolIsSignificant(true);
       st.commentChar('#');
       int slot = 0;

       try
       {
scan:
          while (true)
          {
             switch ( st.nextToken() )
             {
                case StreamTokenizer.TT_EOF:
                   break scan;
                default:
                   break;
                case StreamTokenizer.TT_WORD:
                   String name = st.sval;
                   double x = 0, y = 0, z = 0;
                   if (st.nextToken() == StreamTokenizer.TT_NUMBER)
                   {
                      x = st.nval;
                      if (st.nextToken() == StreamTokenizer.TT_NUMBER)
                      {
                         y = st.nval;
                         if (st.nextToken() == StreamTokenizer.TT_NUMBER)
                            z = st.nval;
                      }
                   }
                   addVert(name, (float) x, (float) y, (float) z);
                   while( st.ttype != StreamTokenizer.TT_EOL &&
                          st.ttype != StreamTokenizer.TT_EOF )
                      st.nextToken();
             }   // end Switch
          }  // end while
          is.close();
       }  // end Try
       catch( IOException e) {}
       if (st.ttype != StreamTokenizer.TT_EOF)
          throw new Exception(st.toString());
    }  // end XYZChemModel()
    /** Add a vertex to this model */
    int addVert(String name, float x, float y, float z) {
    int i = nvert;
    if (i >= maxvert)
```

```
    if (vert == null) {
    maxvert = 100;
    vert = new float[maxvert * 3];
    atoms = new Atom[maxvert];
    } else {
    maxvert *= 2;
    float nv[] = new float[maxvert * 3];
    System.arraycopy(vert, 0, nv, 0, vert.length);
    vert = nv;
    Atom na[] = new Atom[maxvert];
    System.arraycopy(atoms, 0, na, 0, atoms.length);
    atoms = na;
    }
Atom a = (Atom) atomTable.get(name.toLowerCase());
if (a == null) a = defaultAtom;
atoms[i] = a;
i *= 3;
vert[i] = x;
vert[i + 1] = y;
vert[i + 2] = z;
return nvert++;
}
/** Transform all the points in this model */
void transform() {
if (transformed || nvert <= 0)
    return;
if (tvert == null || tvert.length < nvert * 3)
    tvert = new int[nvert * 3];
mat.transform(vert, tvert, nvert);
transformed = true;
}
/** Paint this model to a graphics context.  It uses the matrix associated
with this model to map from model space to screen space.
The next version of the browser should have double buffering,
which will make this *much* nicer */
void paint(Graphics g) {
if (vert == null || nvert <= 0)
    return;
transform();
int v[] = tvert;
int zs[] = ZsortMap;
if (zs == null) {
    ZsortMap = zs = new int[nvert];
    for (int i = nvert; --i >= 0;)
    zs[i] = i * 3;
}
/*
 * I use a bubble sort since from one iteration to the next, the sort
 * order is pretty stable, so I just use what I had last time as a
 * “guess” of the sorted order.  With luck, this reduces O(N log N)
```

```
 * to O(N)
 */
for (int i = nvert - 1; --i >= 0;) {
    boolean flipped = false;
    for (int j = 0; j <= i; j++) {
    int a = zs[j];
    int b = zs[j + 1];
    if (v[a + 2] > v[b + 2]) {
        zs[j + 1] = a;
        zs[j] = b;
        flipped = true;
    }
    }
    if (!flipped)
    break;
}
int lg = 0;
int lim = nvert;
Atom ls[] = atoms;
if (lim <= 0 || nvert <= 0)
    return;
for (int i = 0; i < lim; i++) {
    int j = zs[i];
    int grey = v[j + 2];
    if (grey < 0)
    grey = 0;
    if (grey > 15)
    grey = 15;
    // g.drawString(names[i], v[j], v[j+1]);
    atoms[j/3].paint(g, v[j], v[j + 1], grey);
    // g.drawImage(iBall, v[j] - (iBall.width >> 1), v[j + 1] -
    // (iBall.height >> 1));
}
}
/** Find the bounding box of this model */
void findBB() {
if (nvert <= 0)
    return;
float v[] = vert;
float xmin = v[0], xmax = xmin;
float ymin = v[1], ymax = ymin;
float zmin = v[2], zmax = zmin;
for (int i = nvert * 3; (i -= 3) > 0;) {
    float x = v[i];
    if (x < xmin)
    xmin = x;
    if (x > xmax)
    xmax = x;
    float y = v[i + 1];
    if (y < ymin)
```

```
            ymin = y;
            if (y > ymax)
            ymax = y;
            float z = v[i + 2];
            if (z < zmin)
            zmin = z;
            if (z > zmax)
            zmax = z;
        }
        this.xmax = xmax;
        this.xmin = xmin;
        this.ymax = ymax;
        this.ymin = ymin;
        this.zmax = zmax;
        this.zmin = zmin;
        }
}
/** An applet to put a Cehmical model into a page */
public class XYZApp extends Applet implements Runnable {
    XYZChemModel md;
    boolean painted = true;
    float xfac;
    int prevx, prevy;
    float xtheta, ytheta;
    float scalefudge = 1;
    Matrix3D amat = new Matrix3D(), tmat = new Matrix3D();
    String mdname = null;
    String message = null;
    Image backBuffer;
    Graphics backGC;
    Dimension backSize;
    private synchronized void newBackBuffer() {
    backBuffer = createImage(size().width, size().height);
    backGC = backBuffer.getGraphics();
    backSize = size();
    }
    public void init() {
    mdname = getParameter(“model”);
    try {
        scalefudge = Float.valueOf(getParameter(“scale”)).floatValue();
    } catch(Exception e) {
    };
    amat.yrot(20);
    amat.xrot(20);
    if (mdname == null)
        mdname = “model.obj”;
    resize(size().width <= 20 ? 400 : size().width,
           size().height <= 20 ? 400 : size().height);
    newBackBuffer();
    }
```

```
public void run() {
InputStream is = null;
try {
    Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
    is = new URL(getDocumentBase(), mdname).openStream();
    XYZChemModel m = new XYZChemModel (is);
    Atom.setApplet(this);
    md = m;
    m.findBB();
    float xw = m.xmax - m.xmin;
    float yw = m.ymax - m.ymin;
    float zw = m.zmax - m.zmin;
    if (yw > xw)
    xw = yw;
    if (zw > xw)
    xw = zw;
    float f1 = size().width / xw;
    float f2 = size().height / xw;
    xfac = 0.7f * (f1 < f2 ? f1 : f2) * scalefudge;
} catch(Exception e) {
    e.printStackTrace();
    md = null;
    message = e.toString();
}
try {
    if (is != null)
    is.close();
} catch(Exception e) {
}
repaint();
}
public void start() {
if (md == null && message == null)
    new Thread(this).start();
}
public void stop() {
}
public boolean mouseDown(Event e, int x, int y) {
prevx = x;
prevy = y;
return true;
}
public boolean mouseDrag(Event e, int x, int y) {
tmat.unit();
float xtheta = (prevy - y) * (360.0f / size().width);
float ytheta = (x - prevx) * (360.0f / size().height);
tmat.xrot(xtheta);
tmat.yrot(ytheta);
amat.mult(tmat);
if (painted) {
```

```
    painted = false;
    repaint();
}
prevx = x;
prevy = y;
return true;
}
public void update(Graphics g) {
if (backBuffer == null)
    g.clearRect(0, 0, size().width, size().height);
paint(g);
}
public void paint(Graphics g) {
if (md != null) {
    md.mat.unit();
    md.mat.translate(-(md.xmin + md.xmax) / 2,
            -(md.ymin + md.ymax) / 2,
            -(md.zmin + md.zmax) / 2);
    md.mat.mult(amat);
    // md.mat.scale(xfac, -xfac, 8 * xfac / size().width);
    md.mat.scale(xfac, -xfac, 16 * xfac / size().width);
    md.mat.translate(size().width / 2, size().height / 2, 8);
    md.transformed = false;
    if (backBuffer != null) {
    if (!backSize.equals(size()))
        newBackBuffer();
    backGC.setColor(getBackground());
    backGC.fillRect(0,0,size().width,size().height);
    md.paint(backGC);
    g.drawImage(backBuffer, 0, 0, this);
    }
    else
    md.paint(g);
    setPainted();
} else if (message != null) {
    g.drawString("Error in model:", 3, 20);
    g.drawString(message, 10, 40);
}
}
private synchronized void setPainted() {
painted = true;
notifyAll();
}
private synchronized void waitPainted()
{
   while (!painted)
   {
      try
      {
         wait();
```

```
            }
            catch (InterruptedException e) {}
        }
        painted = false;
    }
}   // end class XYZApp
class Atom {
    private static Applet applet;
    private static byte[] data;
    private final static int R = 40;
    private final static int hx = 15;
    private final static int hy = 15;
    private final static int bgGrey = 192;
    private final static int nBalls = 16;
    private static int maxr;
    private int Rl;
    private int Gl;
    private int Bl;
    private Image balls[];
    static {
    data = new byte[R * 2 * R * 2];
    int mr = 0;
    for (int Y = 2 * R; --Y >= 0;) {
        int x0 = (int) (Math.sqrt(R * R - (Y - R) * (Y - R)) + 0.5);
        int p = Y * (R * 2) + R - x0;
        for (int X = -x0; X < x0; X++) {
        int x = X + hx;
        int y = Y - R + hy;
        int r = (int) (Math.sqrt(x * x + y * y) + 0.5);
        if (r > mr)
            mr = r;
        data[p++] = r <= 0 ? 1 : (byte) r;
        }
    }
    maxr = mr;
    }
    static void setApplet(Applet app) {
    applet = app;
    }
    Atom(int Rl, int Gl, int Bl) {
    this.Rl = Rl;
    this.Gl = Gl;
    this.Bl = Bl;
    }
    private final int blend(int fg, int bg, float fgfactor) {
    return (int) (bg + (fg - bg) * fgfactor);
    }
    private void Setup() {
    balls = new Image[nBalls];
    byte red[] = new byte[256];
```

```
    red[0] = (byte) bgGrey;
    byte green[] = new byte[256];
    green[0] = (byte) bgGrey;
    byte blue[] = new byte[256];
    blue[0] = (byte) bgGrey;
    for (int r = 0; r < nBalls; r++) {
        float b = (float) (r+1) / nBalls;
        for (int i = maxr; i >= 1; --i) {
       float d = (float) i / maxr;
       red[i] = (byte) blend(blend(Rl, 255, d), bgGrey, b);
       green[i] = (byte) blend(blend(Gl, 255, d), bgGrey, b);
       blue[i] = (byte) blend(blend(Bl, 255, d), bgGrey, b);
        }
        IndexColorModel model = new IndexColorModel(8, maxr + 1,
           red, green, blue, 0);
        balls[r] = applet.createImage(
       new MemoryImageSource(R*2, R*2, model, data, 0, R*2));
    }
    }
    void paint(Graphics gc, int x, int y, int r) {
    Image ba[] = balls;
    if (ba == null) {
        Setup();
        ba = balls;
    }
    Image i = ba[r];
    int size = 10 + r;
    gc.drawImage(i, x - (size >> 1), y - (size >> 1), size, size, applet);
    }
}
```

# APPENDIX C

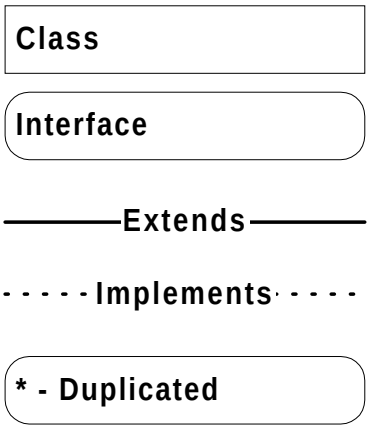
## Java Class Hierarchy

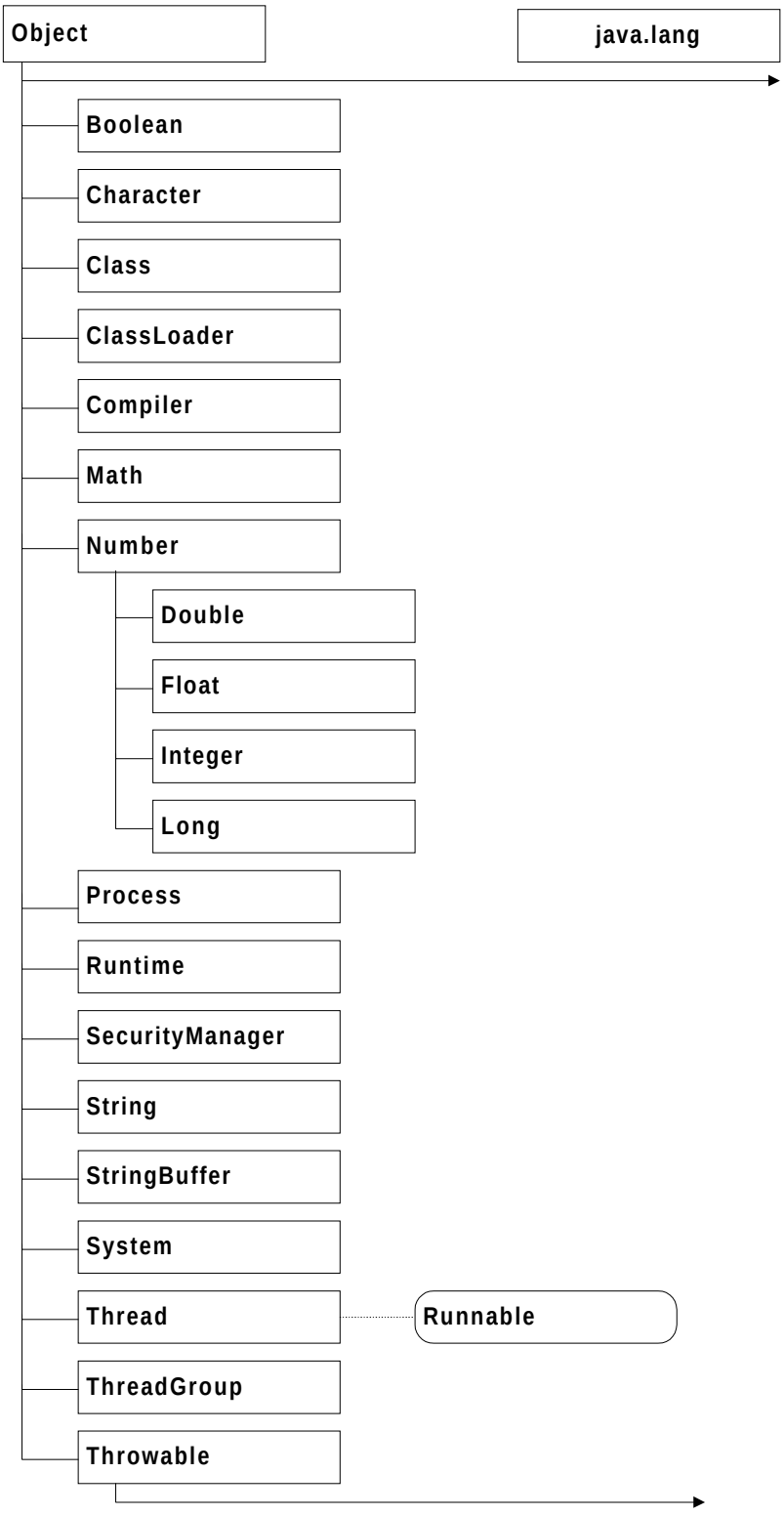
### OVERVIEW

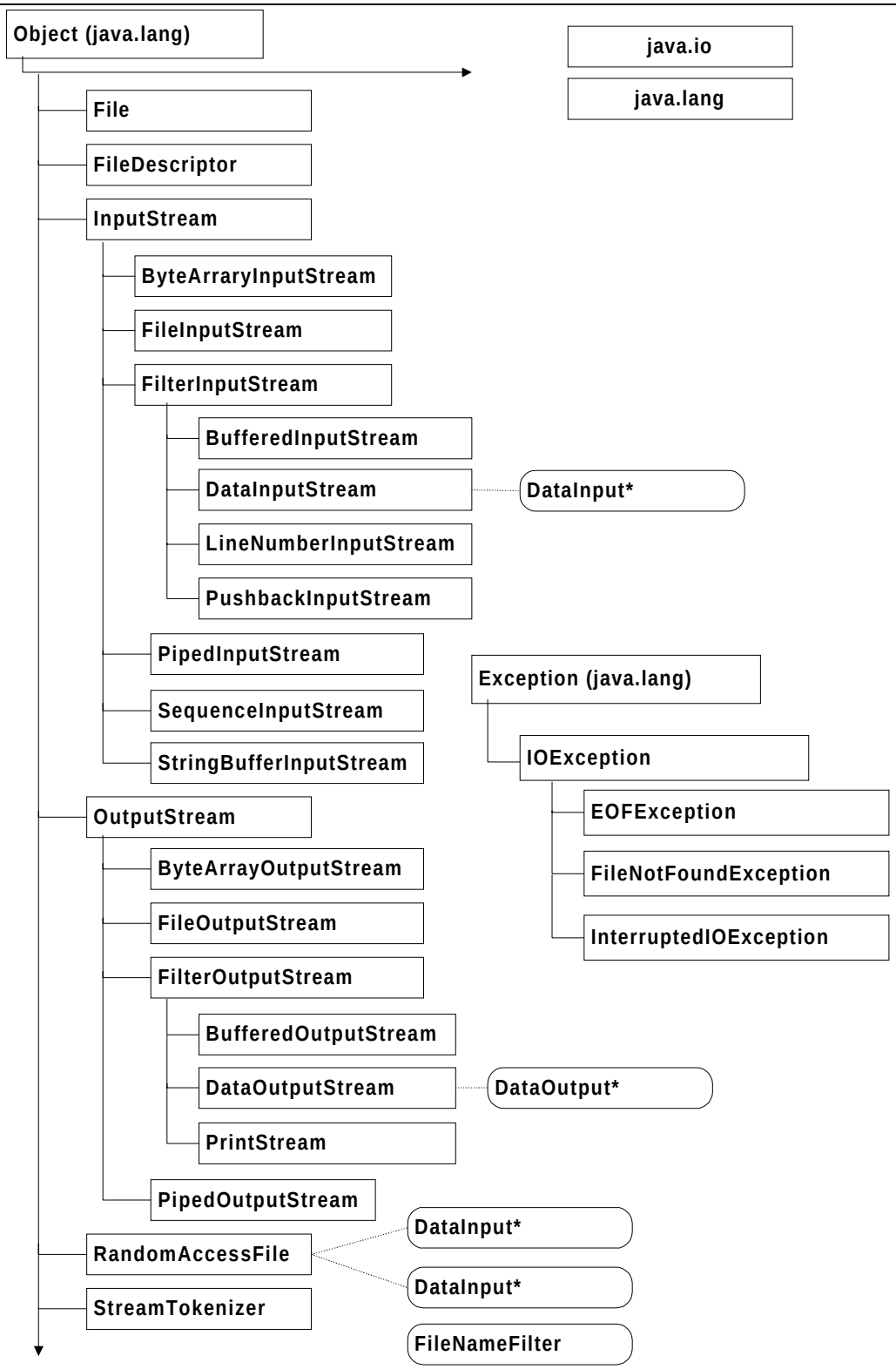
*The Java Class Hierarchy is shown in the next series of pages. Generally, pages are divided by packages, although some packages are too large to fit on a single page. The boxes that are not part of the class library with the centered text serve as keys to the packages represented on that particular page.*

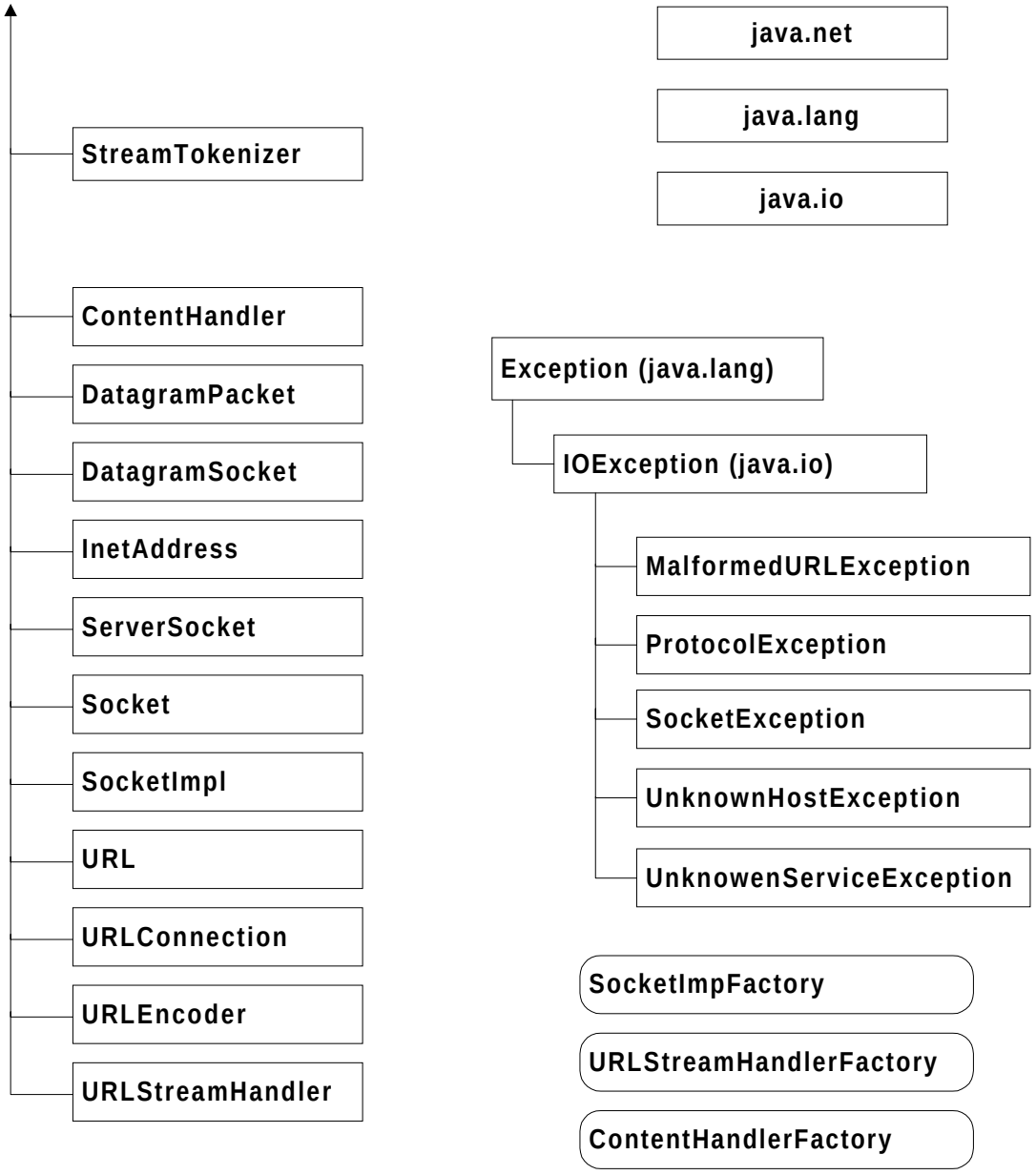
# JAVA CLASS HIERARCHY

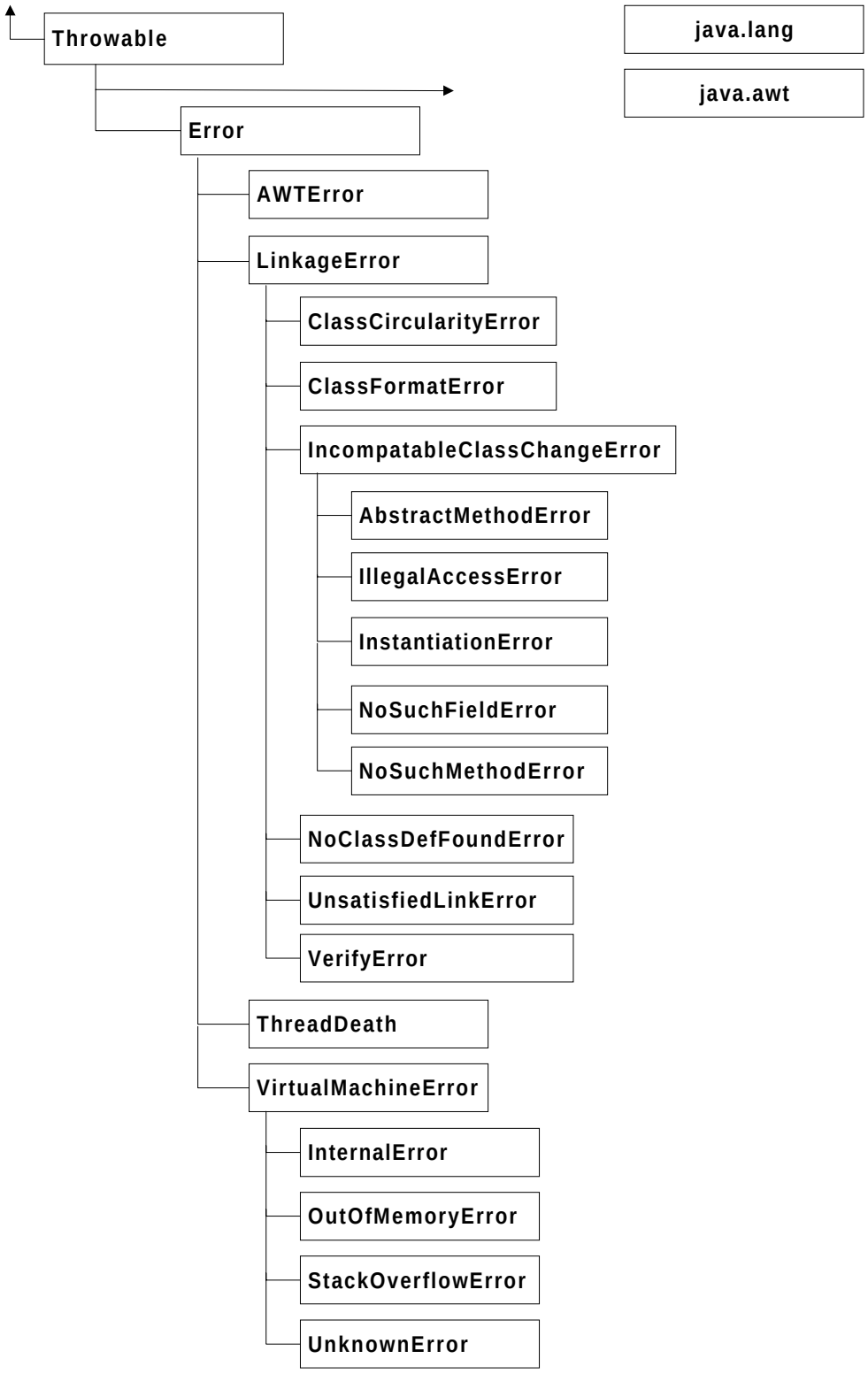
FIGURE 9
*KEY*

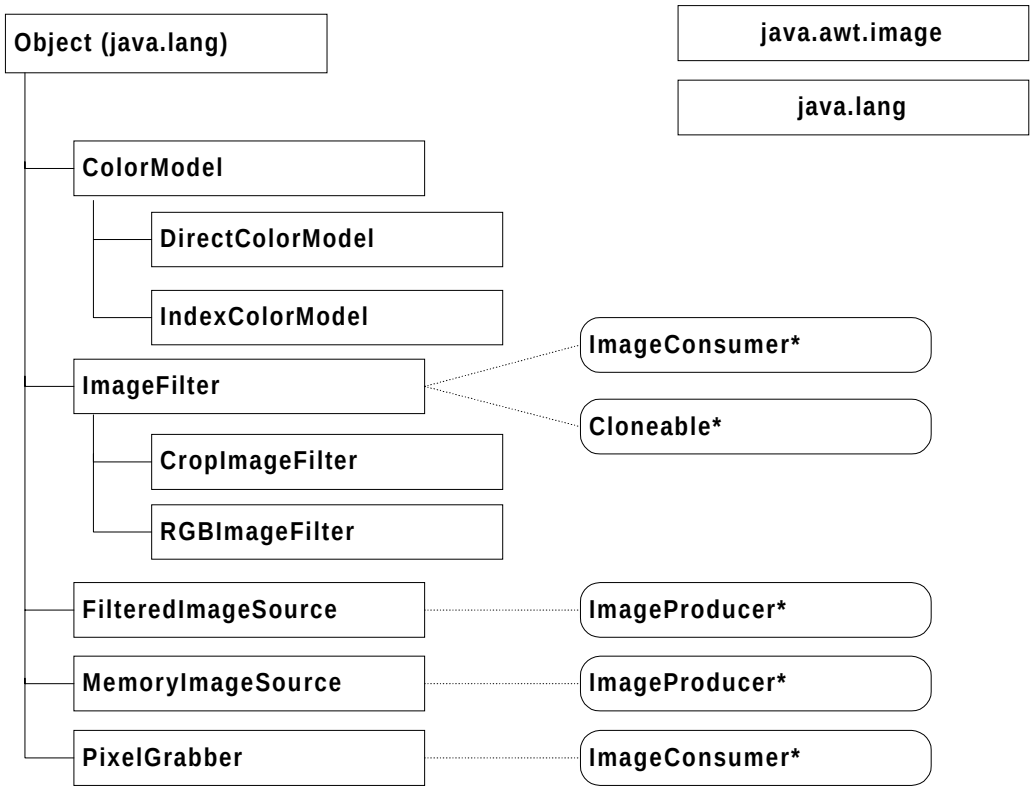


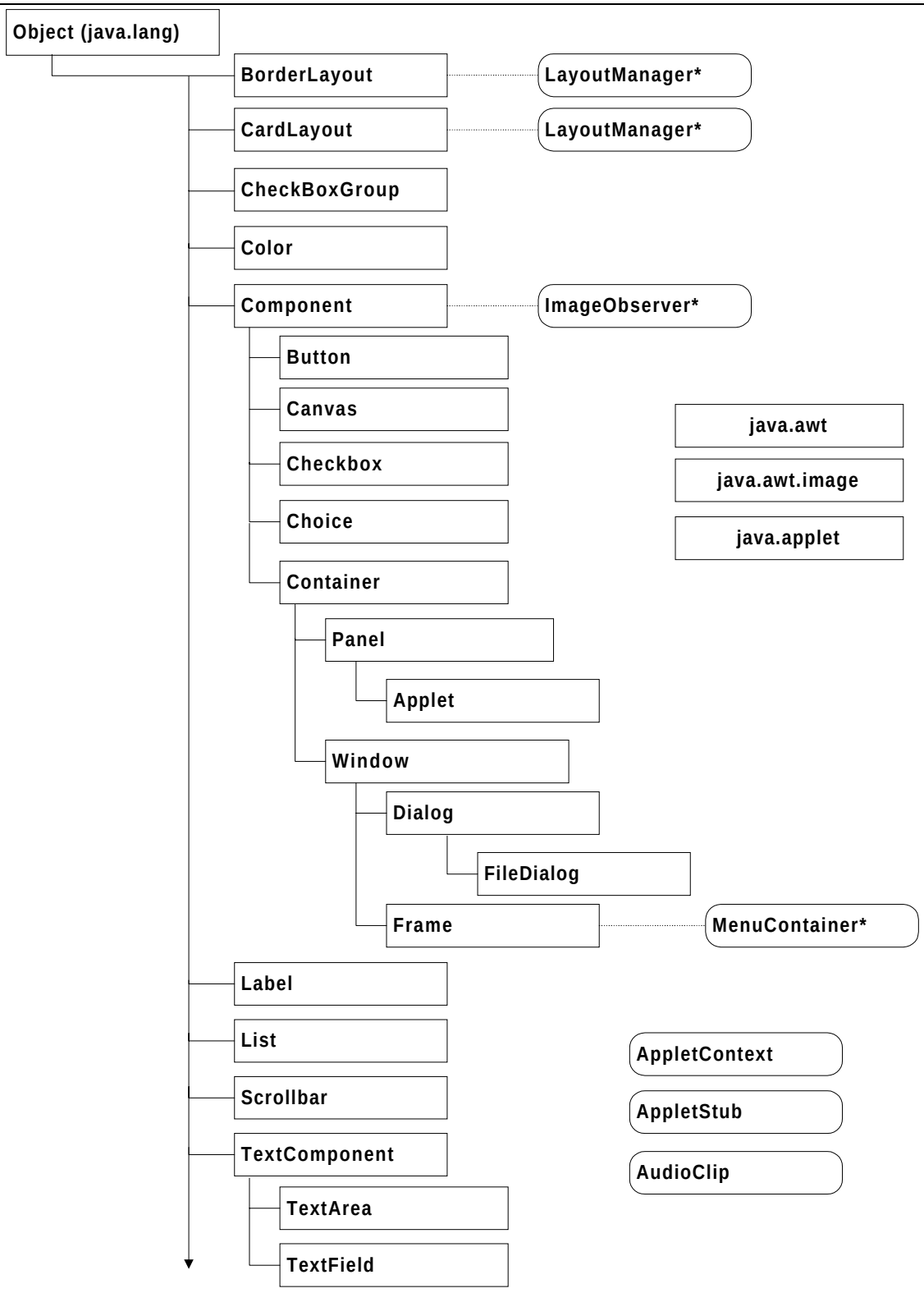

Object
java.lang
Boolean
Character
Class
ClassLoader
Compiler
Math
Number
Double
Float
Integer
Long
Process
Runtime
SecurityManager
String
StringBuffer
System
Thread
Runnable
ThreadGroup
Throwable

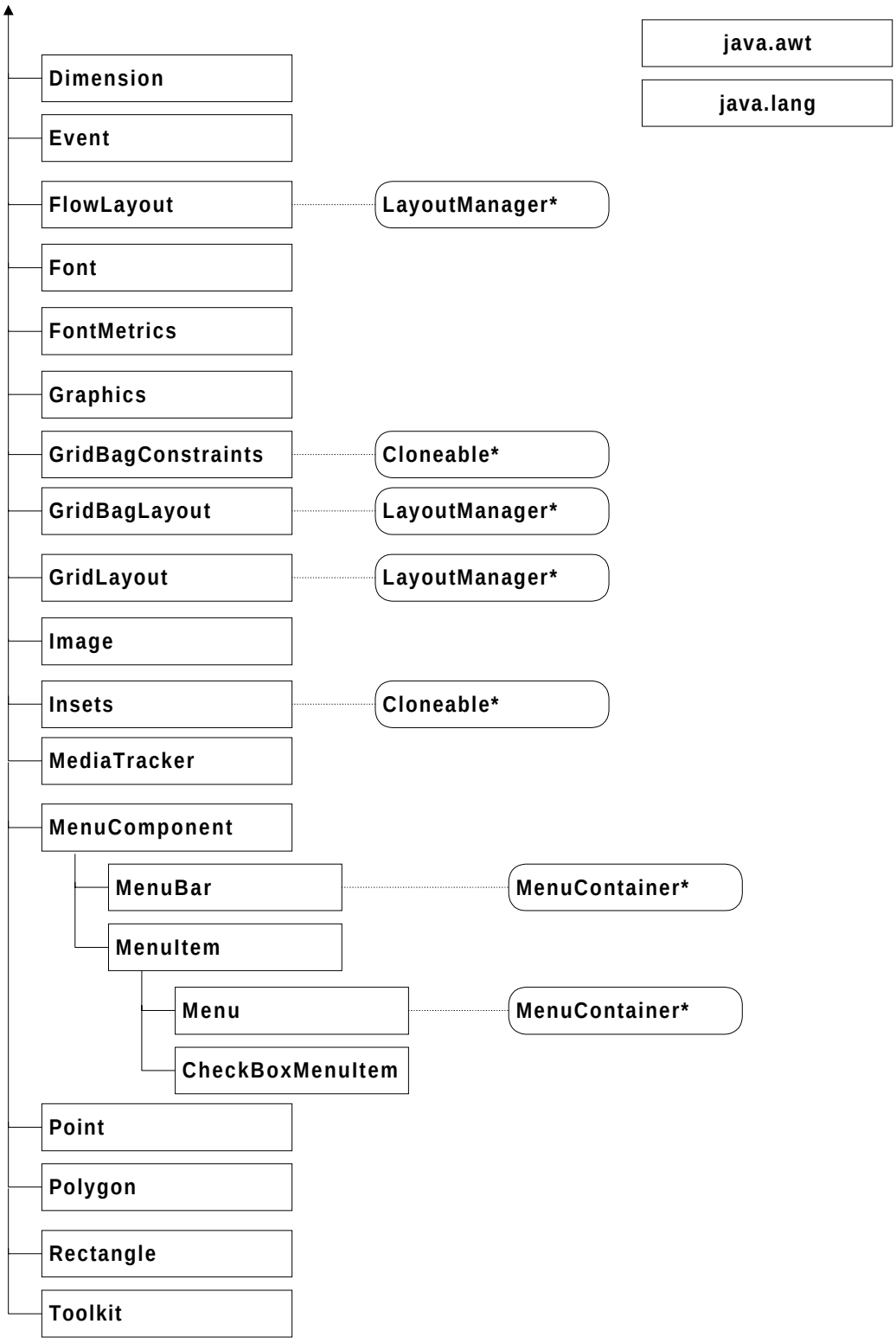

Object (java.lang)
java.io
java.lang
File
FileDescriptor
InputStream
ByteArraryInputStream
FileInputStream
FilterInputStream
BufferedInputStream
DataInputStream
DataInput*
LineNumberInputStream
PushbackInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
Exception (java.lang)
IOException
EOFException
FileNotFoundException
InterruptedIOException
OutputStream
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
BufferedOutputStream
DataOutputStream
DataOutput*
PrintStream
PipedOutputStream
RandomAccessFile
DataInput*
DataInput*
StreamTokenizer
FileNameFilter

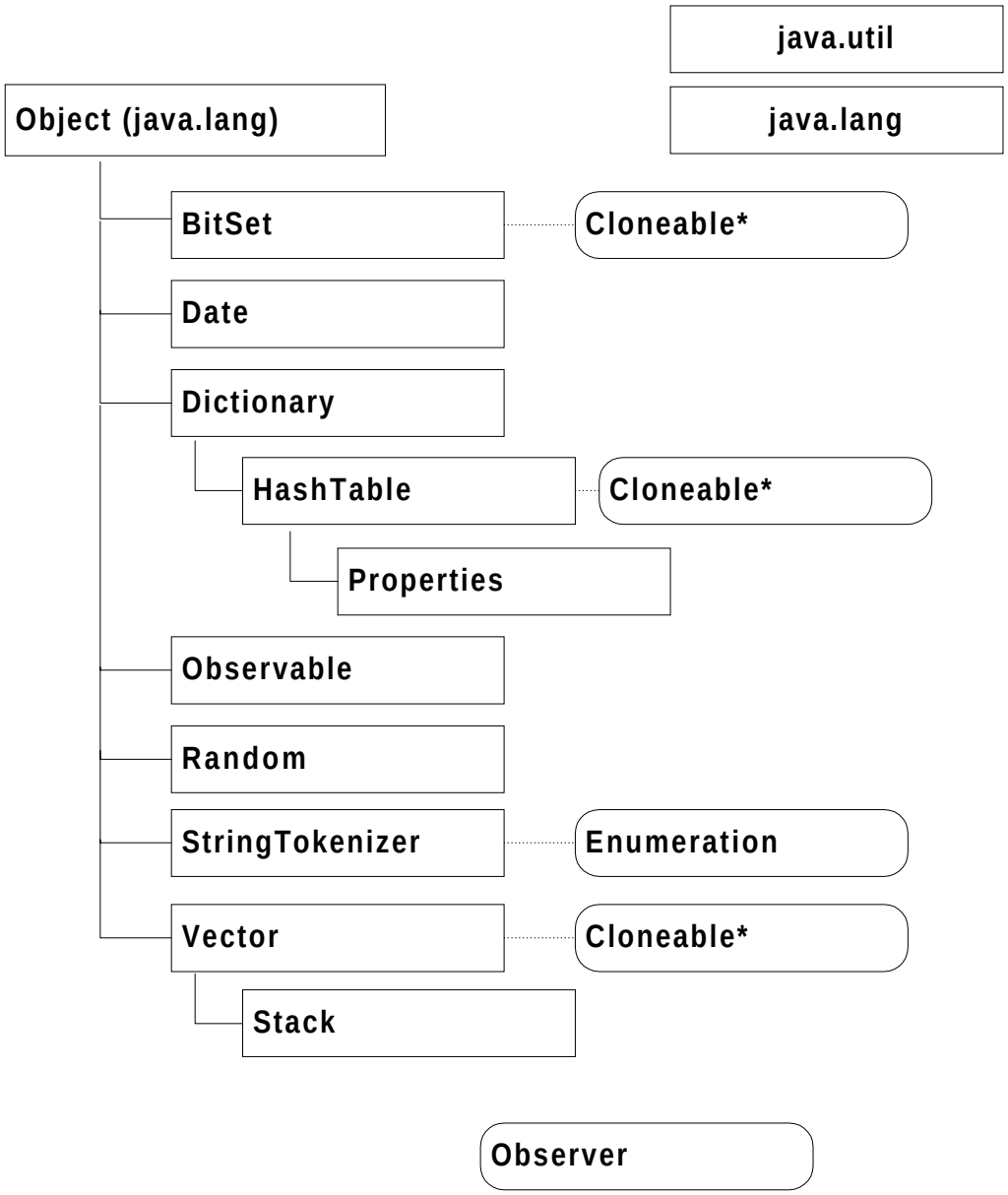

java.net
java.lang
java.io
StreamTokenizer
ContentHandler
DatagramPacket
DatagramSocket
InetAddress
ServerSocket
Socket
SocketImpl
URL
URLConnection
URLEncoder
URLStreamHandler
Exception (java.lang)
IOException (java.io)
MalformedURLException
ProtocolException
SocketException
UnknownHostException
UnknowenServiceException
SocketImpFactory
URLStreamHandlerFactory
ContentHandlerFactory


Throwable
java.lang
java.awt
Error
AWTError
LinkageError
ClassCircularityError
ClassFormatError
IncompatableClassChangeError
AbstractMethodError
IllegalAccessError
InstantiationError
NoSuchFieldError
NoSuchMethodError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
ThreadDeath
VirtualMachineError
InternalError
OutOfMemoryError
StackOverflowError
UnknownError


Object (java.lang)
java.awt.image
java.lang
ColorModel
DirectColorModel
IndexColorModel
ImageFilter
CropImageFilter
RGBImageFilter
ImageConsumer*
Cloneable*
FilteredImageSource
MemoryImageSource
PixelGrabber
ImageProducer*
ImageProducer*
ImageConsumer*


Object (java.lang)
BorderLayout
LayoutManager*
CardLayout
LayoutManager*
CheckBoxGroup
Color
Component
ImageObserver*
Button
Canvas
Checkbox
Choice
Container
Panel
Applet
Window
Dialog
FileDialog
Frame
MenuContainer*
Label
List
Scrollbar
TextComponent
TextArea
TextField
java.awt
java.awt.image
java.applet
AppletContext
AppletStub
AudioClip


java.awt
java.lang
Dimension
Event
FlowLayout
LayoutManager*
Font
FontMetrics
Graphics
GridBagConstraints
Cloneable*
GridBagLayout
LayoutManager*
GridLayout
LayoutManager*
Image
Insets
Cloneable*
MediaTracker
MenuComponent
MenuBar
MenuContainer*
MenuItem
Menu
MenuContainer*
CheckBoxMenuItem
Point
Polygon
Rectangle
Toolkit


java.util
java.lang
Object (java.lang)
BitSet
Cloneable*
Date
Dictionary
HashTable
Cloneable*
Properties
Observable
Random
StringTokenizer
Enumeration
Vector
Cloneable*
Stack
Observer

# ANSWERS

To Review Questions

# LESSON 2

*Review questions, page 46*

1. `javac`
2. Browsers usually run Java applets, though several programs (including `appletviewer` and `HotJava`) can also run Java applets.
3. `<applet code="name-of-class" width=w height=h>`
4. False. Java applications require a `main` method, but Java applets do not.
5. False. `import` merely allows programmers to type shorter names when accessing classes or methods.

# LESSON 3

*Review questions, page 74*

1. a. `int`

   b. `int`

   c. `long`

   d. `double`

   e. `double`

2. a. The & operator always evaluates both its left and right operands. The && operator always evaluates its left operand, then evaluates its right operand only if the result of the left operand is true.

   b. The >> operator fills the high bits of the result with the sign bit of the operand, retaining the original sign in the result. (This is an arithmetic right shift, because it is equivalent to division by a power of two.) The >>> operator fills the high bits of the result with zero. (This is a logical right shift.)

3. The variable named `half` has gone out of scope at the close of the compound statement within the `do..while` loop, but it appears in a relational expression afterward.

# LESSON 4

### *Review questions, page 91*

1. The declaration is as follows:

```
Animal Lion = new Animal(500, 45);
```

2. The following two lines are needed:

```
Lion.weight = 250;
Lion.length = 35;
```

3. There may be only one public class in any particular source file.

# LESSON 5

### *Review questions, page 103*

1. A class with default protection can be accessed (and instantiated) only by other classes in the same package.

2. `final`

3. `static`

# LESSON 6

### *Review questions, page 117*

1. The `length` data member is read-only, and as a result can not be explicitly modified.

2. The two methods are:

```
new_sport = sport.concat(“ball”);
// or
new_sport = sport + “ball”;
```

3. `StringBuffers` are needed to modify character arrays, since regular `Strings` are immutable.

# LESSON 7

*Review questions, page 136*

1. The declarations are as follows:

```
public class Animal
{
}

public class Dog
    extends Animal
{
}

public class Bulldog
    extends Dog
{
}
```

2. Both `protected` and `private protected` members are accessible to derived classes. However, a `protected` member is accessible to any class in the same package, while a `private protected` member is not.

3. The `super` keyword is used in a derived class constructor to call the base class's constructor.

# LESSON 8

*Review questions, page 160*

1. `java.applet.Applet`

2. `getParameter`

3. The HTML code is:

```
<APPLET CODE="Shape.class" WIDTH=200 HEIGHT=100>
<PARAM NAME="color" VALUE="red">
<PARAM NAME="size" VALUE="10">
</APPLET>
```