

Java™ Programming Language for Non-Programmers

SL-110

Student Guide



Enterprise Services
MS BRM01-209
500 Eldorado Blvd.
Broomfield, Colorado 80021
U.S.A.

Rev C, June 2000

Copyright © 2000 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303, U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun Logo, Solaris, JVM, JDK, Forte, Write Once Run Anywhere, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Netscape is a trademark of Netscape Communications Corporation.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

U.S. Government approval required when exporting the product.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Govt. is subject to restrictions of FAR 52.227-14(g) (2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015 (b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS, AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.



Please
Recycle



Adobe PostScript

Contents

About This Course	xiii
Course Goal	xiii
Course Overview	xiv
Module-by-Module Overview	xv
Course Objectives.....	xviii
Skills Gained by Module.....	xix
Guidelines for Module Pacing	xx
Topics Not Covered	xxi
How Prepared Are You?	xxii
Introductions	xxiii
How to Use Course Materials	xxiv
Course Icons and Typographical Conventions	xxvi
Icons	xxvi
Typographical Conventions	xxvii
Syntax Conventions	xxviii
Java Programming Language Overview	1-1
Objectives	1-1
Relevance.....	1-2
Additional Resources	1-3
What You Can Get From This Course	1-4
Java Is a Robust Programming Language	1-4
This Course Is a Starting Point.....	1-5
Java Is a Fun Programming Language.....	1-5
The Origin of the Java Programming Language	1-6
Building a Better Language	1-8
Applets and Applications	1-10
Applications.....	1-10
Applets.....	1-11
Referring to Applets and Applications.....	1-11

Exercise 1: Applets and Applications	1-12
Tasks	1-12
The Computer Communication Problem	1-15
Basic Components of a Computer	1-15
What Programs Need in Order to Communicate	1-17
Two Ways of Solving the Communication Problem	1-19
Compiled Programs	1-20
Interpreted Programs	1-21
How Java Technology Solves the Communication Problem....	1-22
How Applications and Applets Run	1-24
Java Technology Products and Terminology	1-26
Java API Documentation	1-29
Exercise 2: Using the API Specification	1-31
Tasks	1-31
Exercise Summary	1-32
Check Your Progress	1-33
Think Beyond	1-34
Object-Oriented Analysis and Design	2-1
Objectives	2-1
Relevance	2-2
Additional Resources	2-3
Case Study	2-4
Overview of Object Orientation	2-6
Identifying Objects	2-8
Identifying Object Attributes and Operations	2-10
Modeling Objects	2-12
Types of Attributes	2-13
Testing an Object	2-15
Relevance to the Problem Domain	2-16
Independent Existence	2-17
Attributes and Operations	2-18
Exercise: Analyzing a Problem Domain	2-19
Tasks	2-19
Exercise Summary	2-24
Classes	2-25
Object Orientation and Encapsulation	2-27
Example: Comparing Procedural and OO Approaches	2-29
Advantages of Encapsulation	2-32
Interface and Implementation	2-33
Re-Use	2-35
Software Development Stages	2-36
Check Your Progress	2-39
Think Beyond	2-40

Getting Started With Java Technology Programming	3-1
Objectives	3-1
Relevance.....	3-2
Additional Resources	3-3
New Terminology	3-4
The First Step in Writing Java Applications.....	3-5
Basic Java Application.....	3-6
Primary Application Components	3-7
Class Code Block	3-7
Data (Variables).....	3-8
Method Code Block (main in example program).....	3-8
Using Semicolons and Braces.....	3-8
Compiling and Running a Program.....	3-9
Requirements for Your Source File	3-9
Compiling.....	3-10
Running the Program	3-10
Debugging.....	3-11
Exercise 1: Running, Compiling, and Modifying a Basic Program.....	3-12
Tasks	3-12
Computer Data Storage.....	3-15
Storing Data in Variables Using Primitive Data Types	3-18
Primitive Data Types	3-18
Integral Primitive Types	3-20
Floating Point Primitive Types	3-22
Textual Primitive Type: char	3-24
Logical Primitive Type: boolean.....	3-26
Variable Identifier Conventions and Rules	3-27
Picking a Variable Identifier.....	3-28
Variable Identifier Naming Rules.....	3-29
Variable Identifier Naming Conventions	3-31
Exercise 2: Selecting Primitive Types.....	3-32
Tasks	3-32
Creating Primitive Type Variables	3-33
Step 1 – Declaring a Variable.....	3-34
Step 2 – Assigning a Value	3-35
Kinds of Values You Can Assign.....	3-36
Using Variables in a Program	3-38
Constants.....	3-39
How Primitives and Constants Are Stored in Memory	3-41
Exercise 3: Using Primitive Type Variables in a Program	3-42
Tasks	3-42
Check Your Progress	3-43
Think Beyond	3-44

Objects and Programming Structure	4-1
Objectives	4-1
Relevance.....	4-2
Object Overview.....	4-3
Creating Object Reference Variables.....	4-5
Step 1 – Declaration	4-6
Step 2 – Initialization	4-7
Step 3 – Assigning Values to the Object Variables.....	4-8
Reference Variable Information	4-10
How Reference Variables Are Stored in Memory	4-11
Primitive Data Type Storage	4-11
Values of Primitive Variables and Object Reference Variables.....	4-12
Variable Storage in Memory.....	4-13
Assigning One Object Reference to Another	4-14
Using the String Class as a Data Type.....	4-16
Using String and the new Modifier	4-17
Using String Without the new Modifier	4-19
Summary	4-20
Values You Can Assign to Strings.....	4-21
How Strings Are Stored in Memory	4-22
Using String Reference Variables	4-23
Exercise 1: Using Objects and Strings in Programs.....	4-24
Tasks	4-24
Using the main Method.....	4-25
main Method Syntax.....	4-26
main Examples	4-27
Pre-Written Code to Use in Your Programs.....	4-28
Using System.out.println	4-29
Using the main Method to Take Command-Line Input.....	4-32
Converting String Arguments to Integers	4-35
Declaring Classes	4-36
Class Declaration Syntax.....	4-37
Class Example.....	4-38
Exercise 2: Using Command-Line Arguments and String Conversion	4-39
Tasks	4-39
Adding Comments to Programs.....	4-40
Comment Structures.....	4-41
Comment Conventions	4-42
Exercise 3: Commenting Your Code	4-43
Tasks	4-43
Exercise Summary.....	4-44
Check Your Progress	4-45
Think Beyond	4-46

Operators, Casting, and Decision Constructs.....	5-1
Objectives	5-1
Relevance.....	5-2
Arithmetic Operators.....	5-3
Standard Operators	5-3
Increment and Decrement Operators (++ and --)	5-4
Operator Precedence	5-6
Promotion and Typecasting	5-8
Examples of Data and Data Type Mismatch.....	5-9
Promotion.....	5-11
Typecasting.....	5-12
Definition.....	5-12
Syntax and Examples	5-12
Integral and Floating Point Data Types.....	5-14
Integral Data Types and Operations	5-14
Floating Point Data Types and Assignment	5-15
Example	5-16
Exercise 1: Using Operators and Casting	5-17
Tasks	5-17
Logical and boolean Operators.....	5-19
Logical Operators for Primitive Data Types	5-19
boolean Operators	5-20
Basic Parts of an if Construct.....	5-21
Basic if Construct.....	5-23
Choosing Between Two Statements (if/else).....	5-25
Choosing From More Than Two Statements (if/else if/else).....	5-27
Testing Equality Between Strings.....	5-33
Nested if Statements	5-34
Exercise 2: Using the if Construct.....	5-37
Tasks	5-37
The switch Keyword	5-38
if Construct Example	5-38
if Construct Rewritten Using switch	5-39
Syntax for switch Constructs	5-41
When to Use switch Constructs	5-44
Exercise 3: Using the switch Construct	5-46
Tasks	5-46
Exercise Summary.....	5-47
Check Your Progress	5-48
Think Beyond	5-49

Loop Constructs	6-1
Objectives	6-1
Relevance.....	6-2
Loops.....	6-3
The while Loop.....	6-5
Nested Loops.....	6-7
Exercise 1: Using the while Loop.....	6-9
Tasks	6-9
The for Loop.....	6-10
Exercise 2: Using the for Loop.....	6-14
Tasks	6-14
The do Loop.....	6-15
Rewriting a while Loop With a do Loop	6-17
Exercise 3: Using the do Loop.....	6-18
Tasks	6-18
Exercise Summary.....	6-19
Comparing Loop Constructs.....	6-20
The continue Keyword.....	6-21
When to Use continue	6-22
Check Your Progress	6-23
Think Beyond	6-24
Using Methods	7-1
Objectives	7-1
Relevance.....	7-2
Overview of Method Use.....	7-3
Advantages of Method Use	7-4
Worker Method and Calling Method	7-6
Calling Method.....	7-8
Worker Method	7-9
Declaring Methods.....	7-12
Syntax	7-13
Calling a Method.....	7-15
Overview	7-15
Syntax and Examples	7-16
Passing Arguments.....	7-20
Receiving Return Values.....	7-24
Exercise 1: Using Methods.....	7-28
Tasks	7-28
Object Methods and Static Methods	7-29
Object Methods.....	7-30
Static Methods	7-31
Static Methods in the Java API.....	7-33
When to Declare a Static Method	7-34
Method Overloading	7-35

Exercise 2: Using Overloaded Methods.....	7-40
Tasks	7-40
Exercise Summary.....	7-41
Check Your Progress	7-42
Think Beyond	7-43
Object-Oriented Java Programs	8-1
Objectives	8-1
Relevance.....	8-2
Encapsulation	8-3
Private Members	8-3
Private Implementation, Public Interface.....	8-4
Encapsulation Examples	8-5
Elevator: The Wrong Way	8-5
Elevator: The Right Way	8-6
Time Program.....	8-10
Implementing Encapsulation	8-13
Syntax	8-14
Examples	8-14
get and set Methods.....	8-17
Exercise 1: Writing Encapsulated Applications.....	8-19
Tasks	8-19
main Method Placement	8-21
Variable Scope	8-22
Constructors.....	8-24
Example.....	8-25
How Constructors Are Made Available	8-26
The Default Constructor	8-27
What Constructors Can Do.....	8-29
Declaring and Using a Constructor.....	8-30
Overloading Constructors	8-33
Exercise 2: Using Constructors.....	8-34
Tasks	8-34
Exercise Summary.....	8-35
Check Your Progress	8-36
Think Beyond	8-37
Arrays	9-1
Objectives	9-1
Relevance.....	9-2
Array Overview	9-3
The Problem: Creating Many Variables of the Same Type.....	9-4
The Solution: Arrays.....	9-6
Arrays and the main Method	9-7
Arrays and Type.....	9-8

Creating Primitive-Type Arrays	9-9
Declaring Primitive Arrays.....	9-10
Instantiating Primitive Arrays	9-10
Initializing Primitive Arrays	9-11
Declaring, Initializing, and Instantiating Primitive Arrays.....	9-12
Creating Reference-Type Arrays	9-13
Declaring Reference Arrays.....	9-13
Instantiating Reference Arrays	9-13
Initializing Reference Arrays.....	9-14
Declaring, Initializing, and Instantiating Reference Arrays.....	9-15
Accessing a Variable Within an Array	9-16
Exercise 1: Creating and Using Arrays	9-17
Tasks	9-17
How Arrays Are Stored in Memory	9-18
Primitive Variables and Arrays	9-19
Reference Variables and Arrays.....	9-20
Array Bounds	9-21
Finding the Length of an Array	9-22
Setting Array Values Using a Loop.....	9-24
Exercise 2: Using Loops and Arrays.....	9-26
Tasks	9-26
Two-Dimensional Arrays	9-27
Exercise 3: Two-Dimensional Arrays.....	9-29
Tasks	9-29
Exercise Summary.....	9-30
Check Your Progress	9-31
Think Beyond	9-32
Inheritance.....	10-1
Objectives	10-1
Relevance.....	10-2
Inheritance Overview	10-3
Inheritance Definition.....	10-5
Syntax and Examples	10-6
Adding Abstraction	10-8
Testing Inheritance	10-12
Using Code From the Java API.....	10-15
Implicitly Available Classes	10-15
Classes You Must Import or Fully Qualify	10-16
Check Your Progress	10-18
Think Beyond	10-19

Advanced Object-Oriented Concepts	A-1
Objectives	A-1
Relevance.....	A-2
Containment	A-3
Containment Examples	A-4
Polymorphism	A-6
Referring to Multiple Possible Objects.....	A-9
Polymorphic Methods.....	A-11
Overriding Methods.....	A-12
Polymorphism and Overriding.....	A-13
Arrays With Multiple Types.....	A-16
Abstract Methods.....	A-18
Interfaces	A-20
Uses of Interfaces	A-22
Interface Example	A-23
The this Reference.....	A-26
Check Your Progress	A-28
Think Beyond	A-29
Graphical User Interface Development.....	B-1
Objectives	B-1
Relevance.....	B-2
Extent of This Module	B-3
The AWT	B-4
Viewing Package Information.....	B-5
The java.awt Package Class Hierarchy	B-7
GUI Project.....	B-9
Frames.....	B-10
Adding a Button.....	B-13
FlowLayout Manager.....	B-16
BorderLayout Manager	B-19
Creating Panels and Complex Layouts	B-24
Event Handling	B-28
Listening to a Button	B-30
Listening to Multiple Buttons	B-33
Closing the Window	B-36
More Components and Events.....	B-42
Exercise: Graphical User Interface Development.....	B-44
Tasks	B-44
Exercise Summary.....	B-45
Check Your Progress	B-46
Think Beyond	B-47

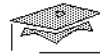
Where Do You Go From Here?	C-1
Getting Ready to Program	C-2
Downloading Java Technology	C-2
Downloading the Java Technology API Specification.....	C-3
Setting up Your Computer to Run Java Programs.....	C-3
Downloading a Development Environment or Debugger	C-4
References	C-6
Basic Java Technology	C-6
Using Applets	C-7
Online Tutorial	C-7
Technical White Papers	C-8
Java Keywords	D-1
Java Naming Conventions	E-1
Class, Method, and Variable Identifiers	E-2
Constant Identifiers	E-3
Navigating the Operating Environment	F-1

About This Course

Course Goal

Java™ Programming Language for Non-Programmers provides first-time programmers an excellent choice for learning programming using the Java™ programming language. Key areas are the significance of the Java programming language, principles of object orientation (OO), and applying those concepts when writing Java technology code encompassing the essential Java programming functions.

At the end of the course, you will be able to write simple Java programs, but will not have extensive programming ability. *Java™ Programming Language for Non-Programmers* provides a solid basis in the Java programming language on which you can base your continued work and training.



Sun Educational Services

Course Overview

- Java technology history
- Object-oriented analysis and design (OOAD)
- Basic Java applications
- Advanced OO concepts
- Graphical user interface (GUI) basics

Course Overview

The course covers a broad spectrum of basic Java programming language information, from the history of how it began to advanced object-oriented programming approaches. It also covers object-oriented analysis and design and how to translate those designs into object-oriented Java technology programs. Appendixes for optional coverage of advanced object-oriented concepts and graphical user interface (GUI) development are also included.



Module-by-Module Overview

- Module 1 – “Java Programming Language Overview”
- Module 2 – “Object-Oriented Analysis and Design”
- Module 3 – “Getting Started With Java Technology Programming”
- Module 4 – “Objects and Programming Structure”
- Module 5 – “Operators, Casting, and Decision Constructs”
- Module 6 – “Loop Constructs”
- Module 7 – “Using Methods”

Module-by-Module Overview

- Module 1 – “Java Programming Language Overview”

This module gives you a background in Java technology, a list of tools that you can use to continue your programming after this class, and a context for future learning in this course.

- Module 2 – “Object-Oriented Analysis and Design”

This module teaches you how to analyze a “problem domain” (a system to be programmed) and identify the items to include when designing the program.

- Module 3 – “Getting Started With Java Technology Programming”

This module describes how data is stored in programs.

- Module 4 – “Objects and Programming Structure”

This module describes to use objects in programs, and the structure in which to write basic Java code.



Module-by-Module Overview

- Module 8 – “Object-Oriented Java Programs”
- Module 9 – “Arrays”
- Module 10 – “Inheritance”
- Appendix A– “Advanced Object-Oriented Concepts”
- Appendix B – “Graphical User Interface Development”
- Appendix C – “Where Do You Go From Here?”

- Module 5 – “Operators, Casting, and Decision Constructs”

This module teaches you how to use operators in expressions in your code, manage mismatched data types in statements, and use those expressions in decision-making constructs.

- Module 6 – “Loop Constructs”

This module covers the different types of loop constructs, which let you control repetition of a set of statements.

- Module 7 – “Using Methods”

This module describes how to extend your knowledge of the main method to declare and call other methods.

- Module 8 – “Object-Oriented Java Programs”

This module describes how to bring together the OO and Java programming language concepts presented so far in this course to write encapsulated, object-oriented programs using good OO design and implementation.

- Module 9 – “Arrays”

This module describes how to use arrays to manage multiple values in the same variable.

- Module 10 – “Inheritance”

This module describes the object-oriented concept of inheritance, and how to use it in Java applications.

- Appendix A – “Advanced Object-Oriented Concepts”

This appendix describes how to implement more advanced object-oriented concepts in the Java programming language.

- Appendix B – “Graphical User Interface Development”

This appendix is an optional part of this course. It provides basic information about graphical user interfaces and how to create a simple window and corresponding functionality. This module is provided so that you can get an idea of Java’s capabilities for graphical user interfaces, and understand a few of the main tools you can use to create them.

- Appendix C – “Where Do You Go From Here?”

This appendix provides guidance for you to continue your Java technology training and practice.

Course Objectives

When you complete this course, you should be able to:

- Describe the history and significant features of the Java programming language
- Download the tools necessary for Java programming language development
- Analyze a programming project using object-oriented analysis and design, and provide a set of classes, attributes, and operations
- Understand a program by reading its source code
- Create and assign values to primitive and reference variables, and use them in programs
- Describe how primitive variables and reference variables are stored in memory
- Write a basic program containing the main method, reference and primitive variables, and a class declaration
- Determine when it is necessary to change variable data types (casting) and write code to do so
- Write programs that implement decision constructs, such as `if/else`
- Write programs that implement loop constructs, such as `while`, `for`, or `do`
- Write programs using multiple methods that call methods, pass arguments, and receive return values
- Write programs that implement good object-oriented concepts, such as encapsulation
- Write programs that create and access a one- or two-dimensional array and its elements
- Describe inheritance and how to implement it in a Java application
- Describe advanced object-oriented concepts like polymorphism
- Write a very simple Java application using a graphical user interface

Skills Gained by Module

The skills for *Java™ Programming Language for Non-Programmers* are shown in column 1 of the matrix below.

Skills Gained	1	2	3	4	5	6	7	8	9	10	A	B	C
Describe the history and significant features of the Java programming language													
Describe how to download the tools necessary for Java programming language development													
Analyze a programming project using object-oriented analysis and design, and provide a set of classes, attributes, and operations													
Understand a program by reading its source code													
Create and assign values to primitive and reference variables, and use them in programs													
Describe how primitive variables and reference variables are stored in memory													
Write a basic program containing the main method, reference and primitive variables, and a class declaration													
Determine when it is necessary to change variable data types (casting) and write code to do so													
Write programs that implement decision constructs, such as if/else													
Write programs that implement loop constructs, such as while, for, or do													
Write programs using multiple methods that call methods, pass arguments, and receive return values													
Write programs that implement good object-oriented concepts, such as encapsulation													
Write programs that create and access a one- or two-dimensional array and its elements													
Describe inheritance and how to implement it													
Describe advanced object-oriented concepts like polymorphism													
Write a simple Java application using a graphical user interface													

Guidelines for Module Pacing

The following table provides a rough estimate of pacing for this course:

Module	Day 1	Day 2	Day 3	Day 4	Day 5
About This Course	A.M.				
Module 1 – “Java Programming Language Overview”	A.M.				
Module 2 – “Object-Oriented Analysis and Design”	P.M.				
Module 3 – “Getting Started With Java Technology Programming”	P.M.				
Module 4 – “Objects and Programming Structure”		A.M.			
Module 5 – “Operators, Casting, and Decision Constructs”		P.M.			
Module 6 – “Loop Constructs”			A.M.		
Module 7 – “Using Methods”			P.M.		
Module 8 – “Object-Oriented Java Programs”				A.M.	
Module 9 – “Arrays”				P.M.	
Module 10 – “Inheritance”				A.M.	
Appendix A – “Advanced Object-Oriented Concepts”					A.M.
Appendix B – “Graphical User Interface Development”					P.M.
Appendix C – “Where Do You Go From Here?”					P.M.



Topics Not Covered

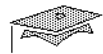
- Advanced Java technology programming – Covered in SL-275: *Java Programming Language*
- Advanced OO analysis and design – Covered in OO-226: *Object-Oriented Application Analysis and Design for Java Technology (UML)*
- Applet programming or web page design

Topics Not Covered

This course does not cover the topics shown on the above overhead. Many of the topics listed on the overhead are covered in other courses offered by Sun Educational Services:

- Advanced Java technology programming – Covered in SL-275: *Java™ Programming Language*
- Advanced OO analysis and design – Covered in OO-226: *Object-Oriented Application Analysis and Design for Java Technology (UML)*
- Applet programming or web page design

Refer to the Sun Educational Services catalog for specific information and registration.



Sun Educational Services

How Prepared Are You?

- Basic computer skills (opening and running applications)

How Prepared Are You?

To be sure you are prepared to take this course, do you have the prerequisites shown on the above overhead?

- Basic computer skills (opening and running applications)



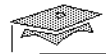
Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Technical background
- Reasons for enrolling in this course
- Expectations for this course

Introductions

Now that you have been introduced to the course, introduce yourself to each other and the instructor, addressing the items listed:

- Name
- Company affiliation
- Title, function, and job responsibility
- Technical background
- Reasons for enrolling in this course
- Expectations for this course



How to Use Course Materials

- Relevance
- Overhead image
- Lecture
- Exercise
- Check Your Progress
- Think Beyond

How to Use Course Materials

To enable you to succeed in this course, these course materials employ a learning model that is composed of the following components:

- **Relevance** – The relevance section for each module provides scenarios or questions that introduce you to the information contained in the module and provoke you to think about how the module content relates to object-oriented design and Java programming.
- **Overhead image** – Reduced overhead images for the course are included in the course materials to help you easily follow where the instructor is at any point in time. Overheads do not appear on every page.
- **Lecture** – The instructor will present information specific to the topic of the module. This information will help you learn the knowledge and skills necessary to succeed with the exercises.
- **Exercise** – Exercises will give you the opportunity to practice your skills and apply the concepts presented in the lecture.

-
- **Check your progress** – Module objectives are restated, sometimes in question format, so that before moving on to the next module you are sure that you can accomplish the objectives of the current module.
 - **Think beyond** – Thought-provoking questions are posed to help you apply the content of the module or predict the content in the next module.

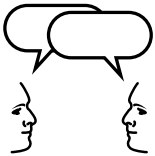
Course Icons and Typographical Conventions

The following icons, typographical conventions, and syntax rules are used in this course to represent various training elements and alternative learning resources.

Icons



Additional resources – Indicates additional reference materials are available.



Discussion – Indicates a small-group or class discussion on the current topic is recommended at this time.



Exercise objective – Indicates the objective for the lab exercises that follow. The exercises are appropriate for the material being discussed.

Note – Additional important, reinforcing, interesting or special information.

Typographical Conventions

Courier is used for the names of commands, files, and directories, as well as on-screen computer output. For example:

```
Use ls -al to list all files.  
system% You have mail.
```

Courier bold is used for characters and numbers that you type. For example:

```
system% su  
Password:
```

Courier italic is used for variables and command-line placeholders that are replaced with a real name or value. For example:

To delete a file, type `rm filename`.

Courier in square brackets is used for variables and command-line placeholders that are optional. For example:

```
[class_modifier] class name {block}
```

Palatino italics is used for book titles, new words or terms, or words that are emphasized. For example:

Read Chapter 6 in *User's Guide*.
These are called *class* options.
You *must* be root to do this.

Syntax Conventions

This course presents the basics of Java programming. Whenever possible, both the syntax of a particular language element and an example are shown. The following example of this approach shows the syntax, then an example of, declaring an integer variable.

type variable_identifier

`int myFirstVariable;`

The first line shown, typically with elements in italics, is the syntax; the second is the example.

Any elements shown in italics are variables, which need to have values substituted for them.

Any elements shown in square brackets, such as the *class_modifier* element in the following class declaration syntax, are optional:

`[class_modifier] class name {block}`

Java Programming Language Overview

1 

Objectives

Upon completion of this module, you should be able to:

- List your expectations for the course, based on the information presented in this module
- Describe when and why Java technology was originally developed
- List at least four items that make Java technology superior to other programming languages
- Describe the difference between an applet and an application
- Describe the processes Java technology uses that make it cross-platform, and how this makes it different from other programming languages
- List two essential components of Java technology
- List the software you need to write Java programs, and that users need to run those programs

This module gives you a background in Java technology, a list of tools that you can use to continue your programming after this class, and a context for future learning in this course.

Relevance



Discussion – Discuss the following questions:

- Why should a new programmer think about using Java instead of another programming language?
- Why are so many companies jumping on the Java technology bandwagon?
- Why did you choose to learn the Java programming language?

Additional Resources

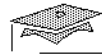


Additional resources – The following references can provide additional details on the topics discussed in this module:

- Java Technology: An Early History
<http://java.sun.com/features/1998/05/birthday.html>

A fifth-anniversary story about the people behind the development of Java technology.

- Java API Specification. Available:
<http://java.sun.com/j2se/1.3/docs/api/index.html>



What You Can Get From This Course

- The Java programming language is complex.
- You will not be an expert programmer.
- You will have a knowledge base.
- This course applies equally to applications and applets.
- You will be able to write simple programs within a few hours.
- Java is a fun programming language to learn.

What You Can Get From This Course

This section is provided to help you get a sense of what you will be able to do by the end of this course, and the extent of becoming a Java technology programmer.

Java Is a Robust Programming Language

Java is an extremely large programming language. It has a reputation for being easy, in part because it corrects many of the problems that C++ still has, and because of its use with applets. However, it is a large, complex programming language like C++ or Fortran, and cannot be mastered in a one-week class.

In addition, its syntax—the rules for how to write what will happen in the program—is very similar to C++, which can be a difficult syntax to learn.

This Course Is a Starting Point

This course will give you the fundamental object-oriented and Java technology knowledge you need to start writing code.

This is not a course about how to write applets. Nor, at the end, will you be able to go home and start writing major applications. This knowledge can be applied equally to applications and to applets. With some practice and additional reading or training, you will be able to write simple applications and applets; with additional work and training, you can become a Java technology programmer.

If your goal is to become conversant in Java technology, rather than to program, then you will have achieved that goal by the end of the week. However, additional practice and training will enhance your knowledge and help you retain what you learn here.

Java Is a Fun Programming Language

It is important that you know what it takes to learn to use a programming language, as presented in the previous parts of this section. But, that said, you can learn a great deal from this class, eventually become an excellent programmer, and enjoy yourself doing it. Java is the latest in a long series of programming languages, so it improves on languages created over the last few decades.

Learning to write Java technology programs (“Java programs”) is well worth the effort, because it is so powerful—it comes with hundreds of pieces of code that let you instantly do things that you would have to spend hours or days coding, in other languages.

You can get started quickly with Java technology; in a few hours, you will be writing simple programs.

Java technology runs on any computer (you will learn more about this later) so you can run the programs you create in class on your computer at work or at home.



Sun Educational Services

The Origin of the Java Programming Language

- Originally named Oak
- Main team members: Bill Joy, Patrick Naughton, Mike Sheridan, James Gosling
- Original goal: use with home appliances
- In 1994, team realized Oak was perfect for Internet
- Announced in May of 1995
- First non-beta release in 1996
- Java technology is free

The Origin of the Java Programming Language

The Java programming language originated as part of a research project to develop advanced software for a wide variety of networked devices and embedded systems. Bill Joy and James Gosling were the main people on the project.

Java technology was created as a programming tool in a small, closed-door project initiated by Patrick Naughton, Mike Sheridan, and James Gosling of Sun™ Microsystems in 1991. Bill Joy and others joined later. At that point, their goal was to create a language that would work on interactive, handheld home-entertainment device controllers for home-entertainment and home-appliance functions.

This was a very early version of the current vision for Java technology and the Internet that Scott McNealy, the CEO of Sun Microsystems, talks about. His vision is full of refrigerators that tell you when your milk has gone bad and send your microwave recipes about what to make from your leftovers, and smart light bulbs that know when they are going to burn out and send an order to the factory just in time.

Team member James Gosling called the new language Oak; the story goes that this is simply because there was an oak tree outside his window.

The team tried to find a market for this type of device, and the TV set-top box and video-on-demand industries seemed to make the most sense. However, no one at that point gave them much encouragement; it did not seem particularly marketable or useful to have those features in appliances.

Luckily for the computing world, in 1994, the Internet was becoming popular. The team realized that Oak fit perfectly with the way applications were written, delivered, and used on the Internet.

Java technology was designed to be platform independent. The Internet is the largest client-server system in the world with all sorts of different clients. Web designers certainly could not expect to write different programs for every possible computer that could access their pages.

Java technology can also move objects in a web page, which HTML cannot do. When the team did a demo of Java that used applets, the audience of programmers and venture capitalists was stunned. This demo was the point at which the team knew that not only did they really have something special, but that the rest of the world was about to recognize it.

Java technology was announced in May of 1995, and the first non-beta version of the language was released in 1996. Java has developed much more quickly than other programming languages.

Java technology is absolutely free; you can download it from the Sun™ web site (see Appendix C, “Where Do You Go From Here?”).



Building a Better Language

- Object-oriented
- Improved on C++
- Distributed
- Multi-threading
- Security
- APIs (pre-written code)
- Easily portable to multiple platforms (Write Once, Run Anywhere™)

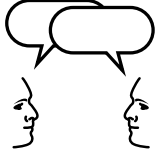
Building a Better Language

When Java technology switched from being a desktop system to a programming language, the primary goals were to develop an operating environment that met certain criteria. The following list includes these criteria as well as other advances that were made.

- Object-oriented – The numerous advantages of using object orientation include code reuse and improved maintenance.
- Improved on C++ – Changed and improved upon the powerful but difficult and potentially dangerous C++ features.
- Distributed – Able to run over a network.
- Multi-threading – Excellent capabilities for running several processes at a time (like opening a web page, and printing a document simultaneously).
- Security – One example of this is applets, which have built-in security controlling whether they can read and write to disk.
- Pre-written code – Java technology has a vast set of code that has already been written for you.

- Platform-independent – Easily modifiable to run on different platforms such as UNIX®, Macintosh, and Microsoft Windows, unlike the programs written in other languages.

Discussion – Why do we need a programming language?





Applets and Applications

- Application – Runs in the operating system
- Applet – Runs in a browser, does smaller tasks
- "Programs" – Applies to applications and applets

Applets and Applications

While Java technology's first use was on the Internet, it has rapidly grown into a complex programming language, and, increasingly, is being chosen over other programming languages such as C++ as the language to use for creating large *applications*.

Applications

An application is a type of program that runs on a single machine or over a network, and interacts with the computers' operating systems.

Applets

Applets are different from applications; they are not self-contained programs and are more like application fragments. Applets have a variety of uses that include displaying animated text and graphics, loading URLs, and doing calculations.

Applets run within the browser, not the operating system.

Applets are typically very small programs. They might need to be run from a computer thousands of miles away, or downloaded to the computer where they are being run.

Applets have very specific properties that make them ideal for running within a browser environment. They also have built-in features for the Internet such as security; for example, they are inspected for viruses.

Referring to Applets and Applications

Both applications and applets come under the common category of programs; however, you cannot use the term “application” to mean applets.

Note – This is not a course on writing applets, or applications, specifically. This course gives you the fundamental knowledge of Java technology, and of object orientation, that will allow you to go on to write both applets and applications.

Exercise 1: Applets and Applications



Exercise objective – See the difference between an applet and an application, and run both.

Tasks

Follow the directions from your instructor to locate the exercises directory for this module.

If you are unfamiliar with the operating system you are using in this class, see Appendix F, “Navigating the Operating Environment.” It provides basic commands as well as tips for using the Solaris Operating System.

Running an Applet

1. Start Netscape.
2. Choose File > Open Page, then click Choose File. Navigate to the location of the `calculator.html` file in your exercises directory, as directed by your instructor.
3. Click the Open in Navigator button.
4. The HTML file will open, with the simplified calculator applet displayed.
5. Choose View > Page Source to view the HTML source code.

Running an Application: Solaris Operating Environment

1. Open a terminal window using either of the following methods:

- ▼ Right-click on the desktop and choose Tools > Terminal.
- ▼ Open File Manager by clicking on the File Manager icon in the navigation bar, then choose File > Open Terminal.



2. Type `cd` followed by the full path to the module 1 exercise directory, as directed by your instructor.

Example: `cd home/export/SL110/SL110_LF/01_javaoverview`

Press Return.

3. Type the following at the terminal window command-line prompt:
`java StickyPad`
4. An application that functions as an on-screen sticky note will appear.

Running an Application: Microsoft Windows

1. Open a DOS terminal window.
2. Type `cd` followed by the full path to the module 1 exercise directory, as directed by your instructor.

Example: `cd C:\SL110\SL110_LF\01_javaoverview`

Press Return.

3. Type the following at the terminal window command-line prompt:
`java StickyPad`
4. An application that functions as an on-screen sticky note will appear.



Sun Educational Services

Basic Components of a Computer

- Hardware
- Operating system
- Applications

The Computer Communication Problem

As powerful and intelligent as computers are, it is not possible to simply put an application on any computer and have it run. This section covers the limitations placed on computer communication; the next section covers how Java technology solves the problem.

Basic Components of a Computer

The essential parts of a computer and what runs on it are:

- Hardware – The box that sits on or under your desk.
- Operating system – The software that is installed on the hardware. This is the system of commands that lets applications communicate with the hardware.
- Applications – All the things you use on a daily basis, such as your browser, email, or word-processing application. These are written in various types of programming languages, such as Fortran, COBOL, C++, Java, and so on.

Figure 1-1 shows an example of this basic communication.

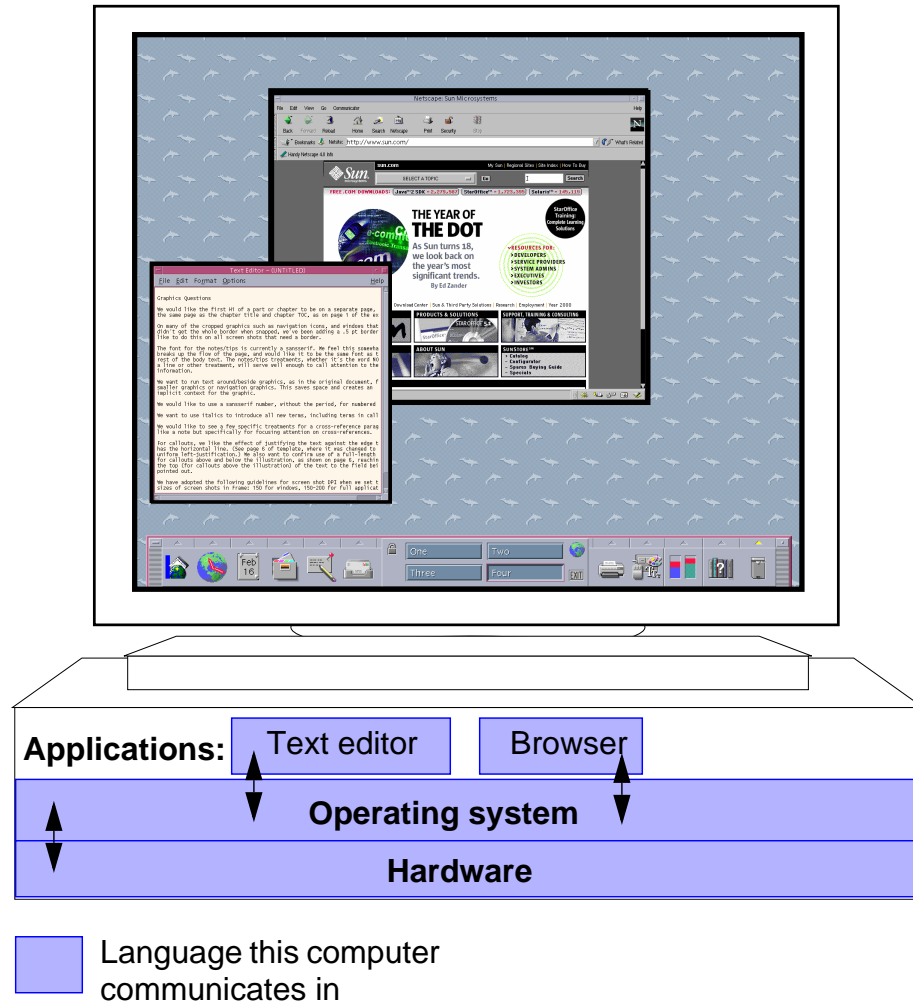


Figure 1-1 How Applications Interact With Computers

What Programs Need in Order to Communicate

Each of the main computer categories—Macintosh, Microsoft Windows, and so on—understands a different type of language. This makes it a little complicated for a program to talk to more than one type of computer.

In addition, the components for each type of computer speak a different language. You need to convert the software so it talks to the hardware, and you need to make sure you convert it to the specific kind of hardware that you want it to run on.

As a result of this requirement of speaking the same kind of language, you cannot simply write a program, in Java or any programming language, copy it onto a computer, and run it. Program code files need conversion or processing of some sort in order to talk to hardware.

Each type of hardware speaks a different kind of machine language, so this usually means that the programs are converted so that they only run on one kind of computer.

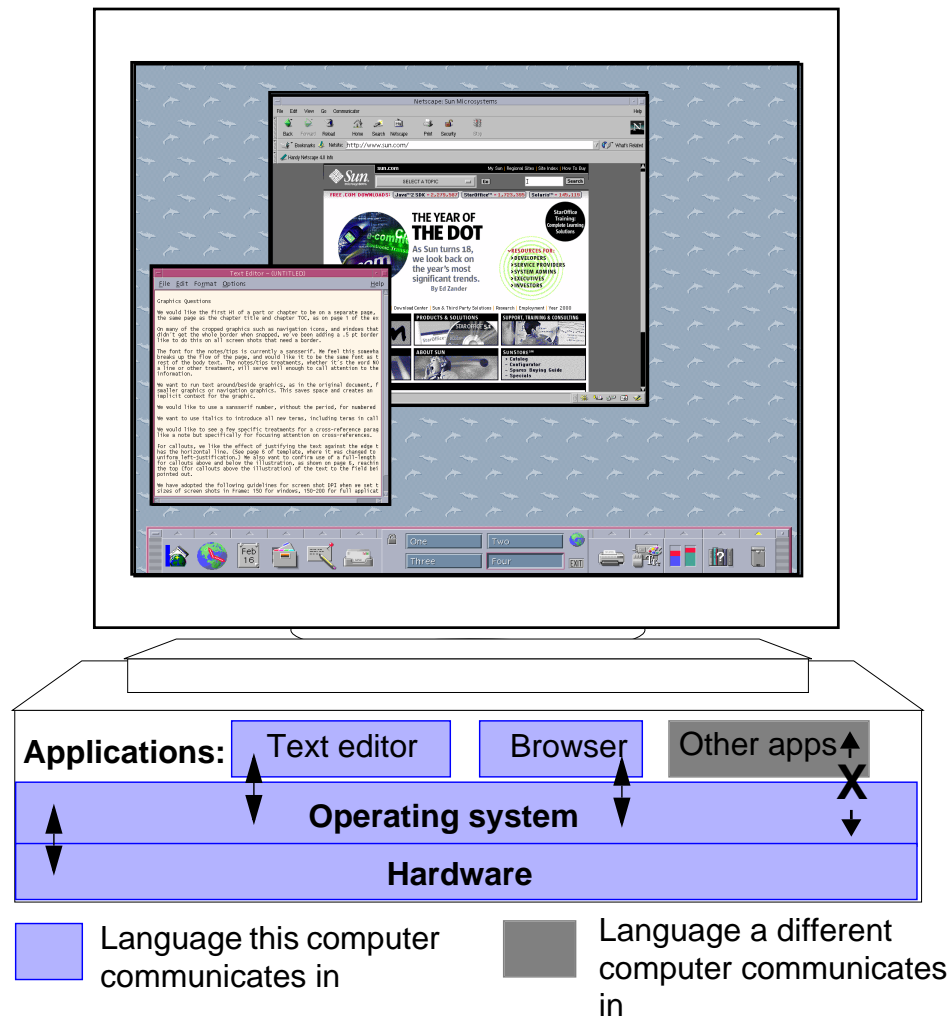


Figure 1-2 Computer Communication Problem

For example, making a program so that it can only talk to Microsoft Windows means it cannot run on Macintosh computers.



Two Ways of Solving the Communication Problem

- Compiled – Communicates with only one kind of computer
- Interpreted – Communicates with any kind of computer

Two Ways of Solving the Communication Problem

Most code development uses the following process: You type your program into a file, using a text editor or other development environment. Then you need to find a way to process the program so that the hardware and operating system understand it. Most programs use one of the following types of processing:

- Compiled
- Interpreted

Compiled Programs

Most programs are compiled, meaning the source code, the program you write, is run through a program that translates it into 0s and 1s. Computers can only use 0s and 1s to figure out what they need to do. However, since each computer communicates differently, the 0s and 1s are different for each type of computer: Macintosh, the Solaris Operating Environment, Microsoft Windows, and so on. (The 0s and 1s are also referred to as a *binary file*.)

The flowchart in Figure 1-3 demonstrates this process.

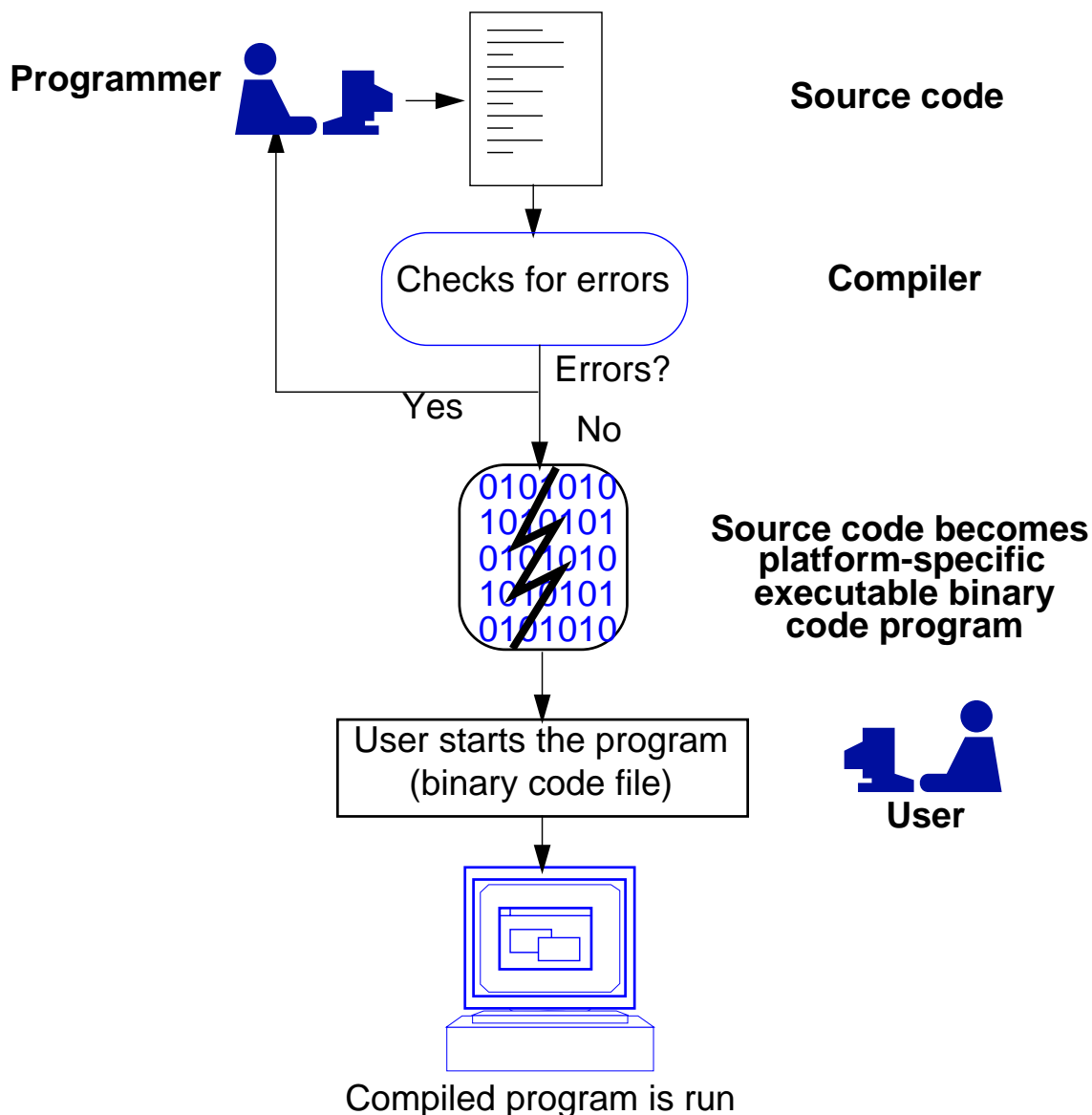


Figure 1-3 Creating a Compiled Program

Compiled programs are reasonably fast to run, because the program talks directly to the computer. Also, the programmer checks for errors in the program by compiling and fixing the program, so it is done by the time you run the program.

However, the file that the source code is compiled to is platform-specific. Programmers creating the program must produce a different version for each platform they want to run on.

Interpreted Programs

Some programs are interpreted. This means that instead of a compiler making the source code readable by the computer, an interpreter does it when you run the program. The flowchart in Figure 1-4 demonstrates this process.

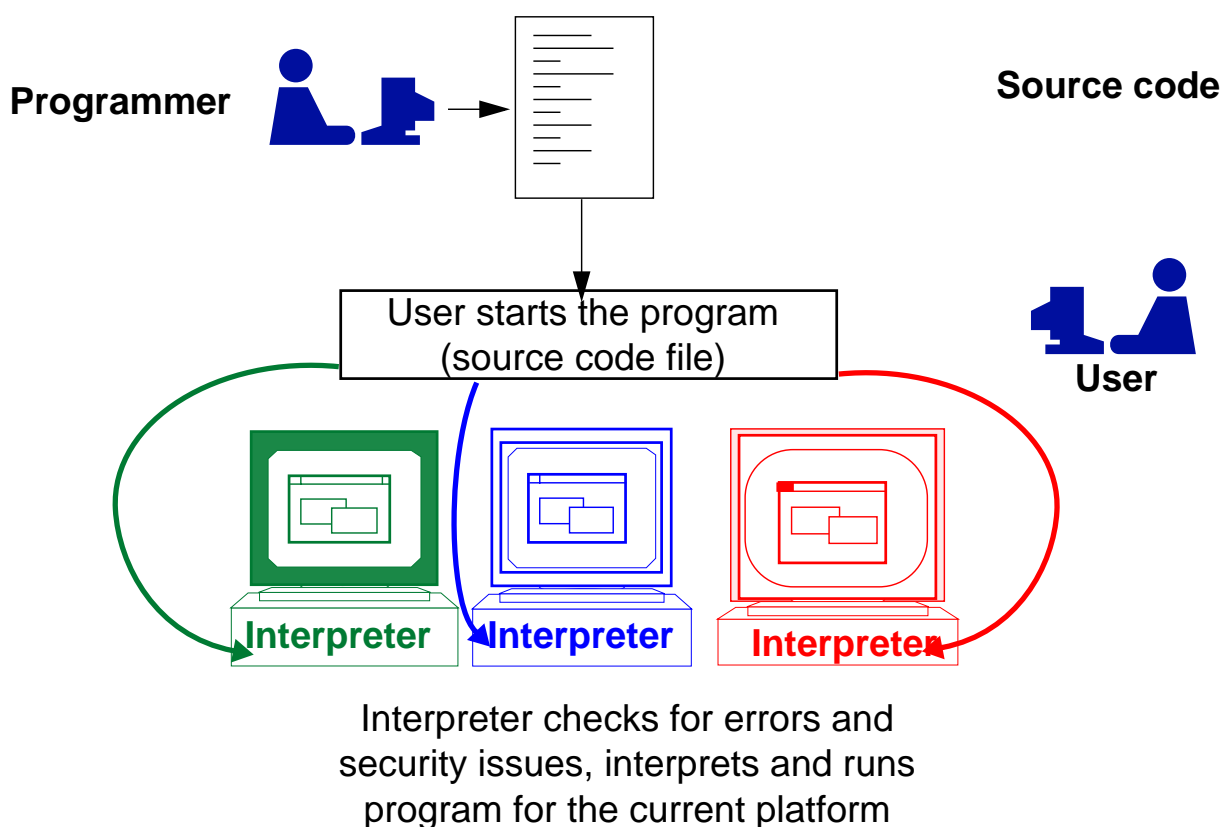


Figure 1-4 Creating an Interpreted Program

Interpreted programs are cross-platform, but run slower because the error checking and interpreting are done when you run the program.



Sun Educational Services

How Java Technology Solves the Communication Problem

- Uses compiling and interpretation
- A little slower than compiled programs, but runs on any operating system
- Compiles source code to *bytecode*
- Uses Java virtual machine (JVM™), interprets bytecode
- Uses a different JVM for every operating system

How Java Technology Solves the Communication Problem

Java technology solves the problem of the tradeoff between speed and platform compatibility by being both compiled and interpreted. Instead of being compiled to machine code, however, Java source code is compiled to bytecode. This bytecode file is platform-generic; it is not tied to any one operating system. When you run a Java program the bytecode file is then interpreted by the Java technology interpreter, the Java virtual machine (JVM).

There is a different JVM for every platform Java technology runs on, and must be included on any computer on which you want to run a Java program. A JVM is included in browsers, if they run applets.

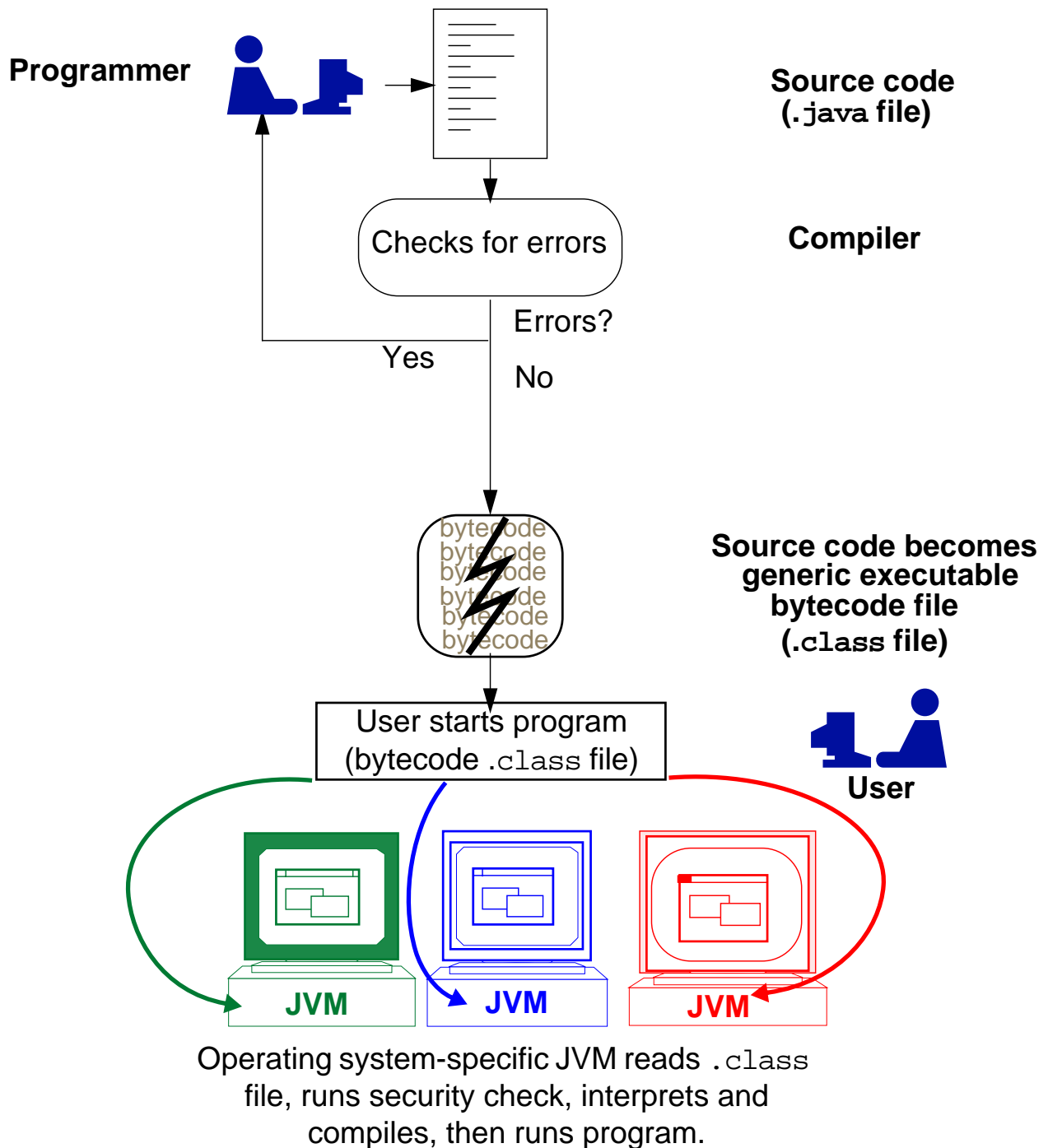


Figure 1-5 Operating System, Hardware, and Application Interaction

How Applications and Applets Run

A Java application can run on any computer, as long as there is a JVM for that type of computer installed. Figure 1-6 shows a Java application running on the computer first seen in Figure 1-5 on page 1-23.

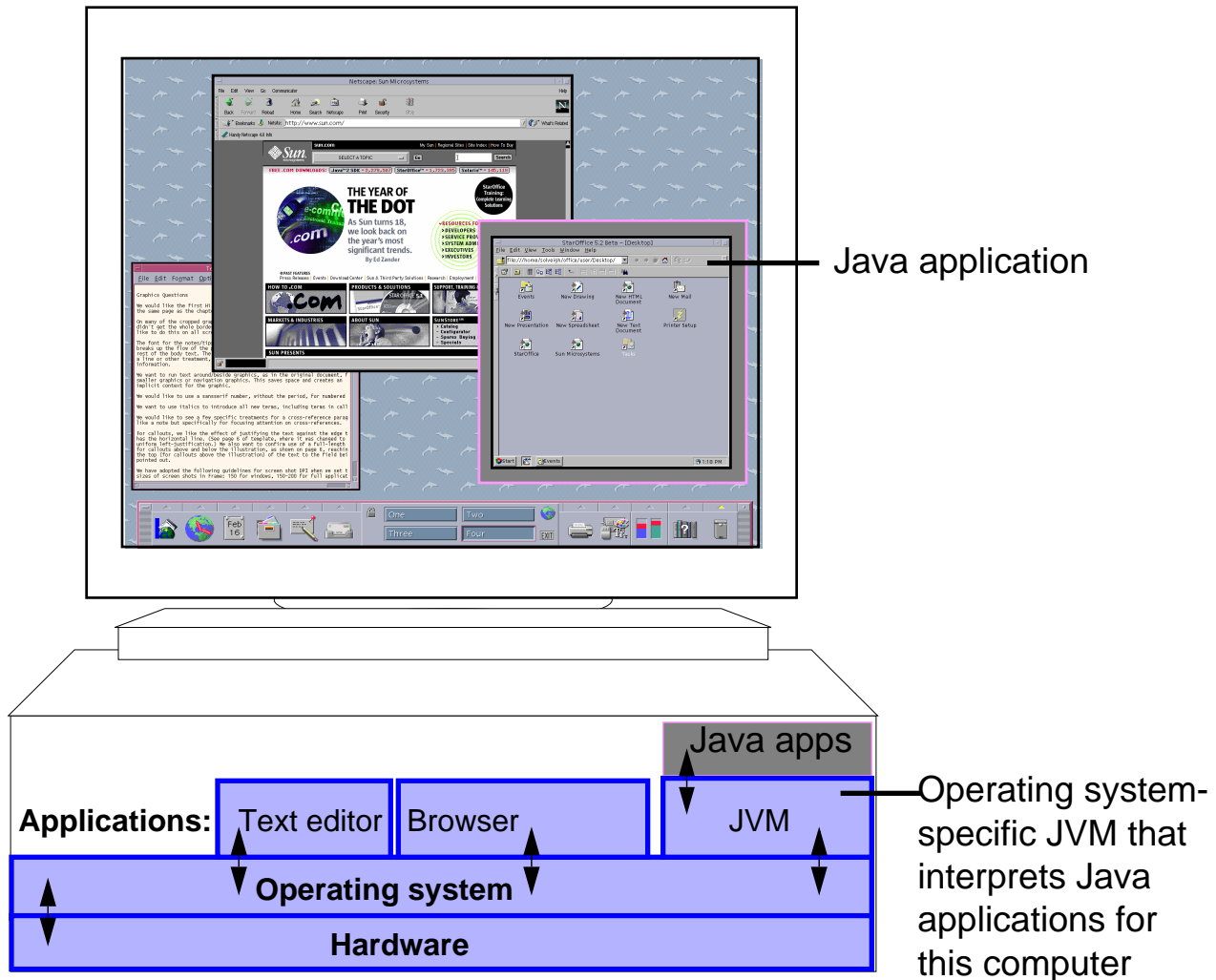


Figure 1-6 How the JVM Interacts With the Operating System and Java Applications

Java applets also are interpreted by the JVM; however, they are interpreted by a separate JVM, which runs inside the browser.

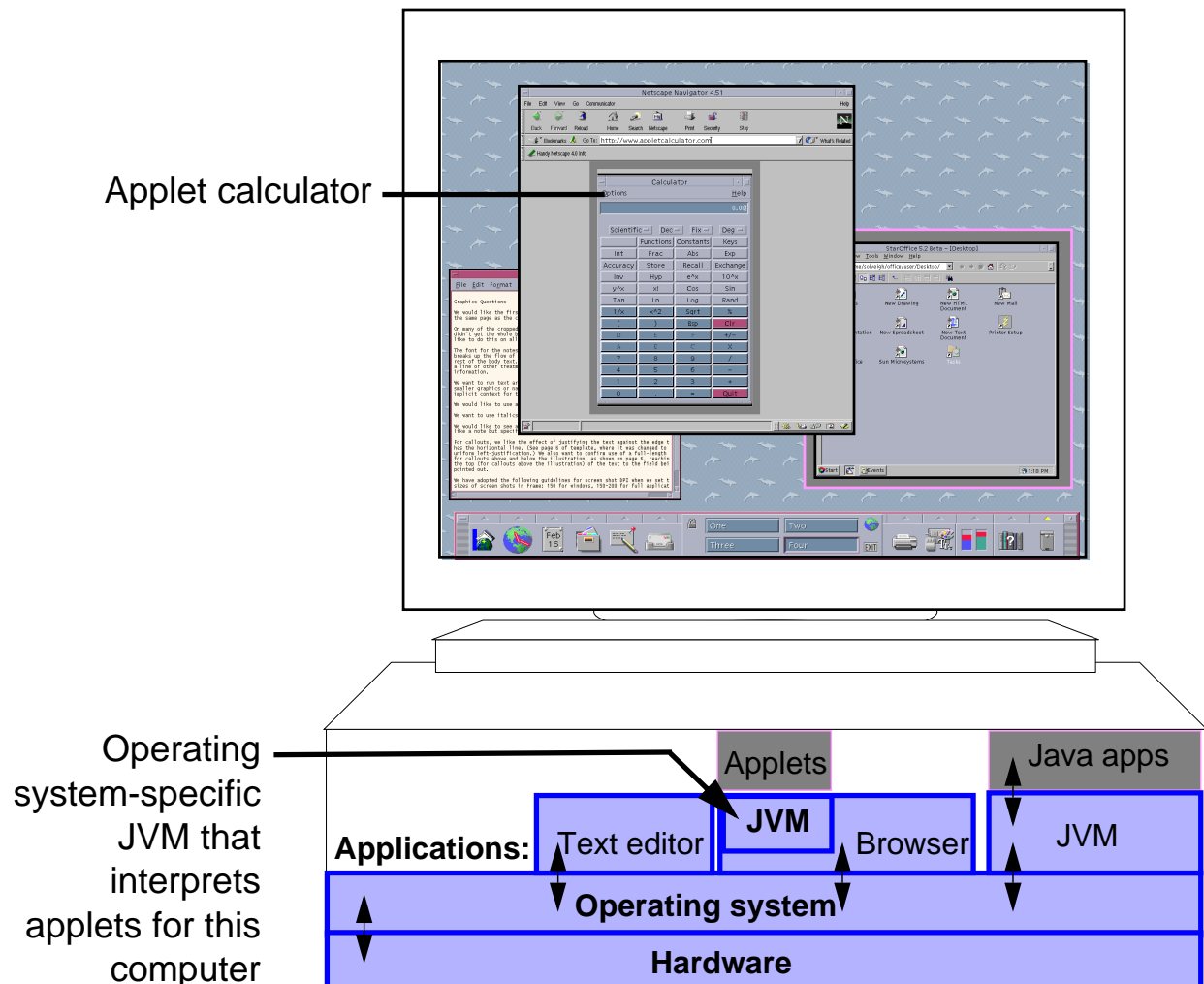
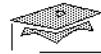


Figure 1-7 How the JVM Interacts With the Operating System and Java Applets



Sun Educational Services

Java Technology Products and Terminology

- Java virtual machine (JVM)
- Java runtime environment (JRE)
- Application programming interface (API)
- The JDK™ (Java developer's kit) – Old name for Java technology
- Java 2, Java 2 Platform, J2 SDK – Current name

Java Technology Products and Terminology

This section is provided to help you get a more concrete idea of what Java technology is, including:

- What the different products are
- Which products you need
- How to find out about each function in the Java programming language

Note – This section lets you know what each term means and what you need; Appendix C, “Where Do You Go From Here?” provides detailed information on how to download the software and documentation.

Table 1-1 differentiates among Java technology terms and products.

Table 1-1 Java Technology Terminology

Term	Description
Java virtual machine (JVM)	Interprets Java technology bytecodes so that the computer can understand them. The JVM has no other associated files.
Java runtime environment (JRE)	<p>The JVM itself is not enough for a Java application or applet to run (though it was presented that way in the previous sections for simplicity's sake.)</p> <p>The JRE is what you include with your programs so that your users can run them.</p> <p>It consists of the JVM plus Java platform core classes, and supporting files.</p>
API	Application programming interface. The API is the rules (syntax) for how to program in Java technology. It includes hundreds of <i>class libraries</i> , pre-written code that you can take advantage of when writing your programs.
The JDK™ (Java developer's kit)	<p>The <i>old</i> name for Java technology, up to November of 1999.</p> <p>JDK 1.1 was the last version. (If you go to http://java.sun.com/jdk/ you will see information about Java 2.)</p> <p>It included the JVM, a compiler and other tools, and all APIs and associated files.</p> <p>An application programming interface, or <i>API</i>, is a set of rules for programming, so that your code will work the way it should. This is the programming language and development tool part of Java technology.</p>
Java 2, Java 2 Platform, J2 SDK (software developer's kit)	<p>The <i>new</i> name for Java technology, as of November 1999.</p> <p>In old terminology, it would be equivalent to JDK 1.2.</p> <p>Important: The name changed, but the version numbering did not. Currently at http://java.sun.com/jdk/ the link to download the current Java 2 Platform is "Java 2 SDK, Standard Edition, v 1.2.2".</p> <p>Java 2, like the JDK, includes the JVM, a compiler and other tools, and Java APIs and associated files.</p>

You can think of the three main Java technology products as a series of concentric circles, shown in Figure 1-8.

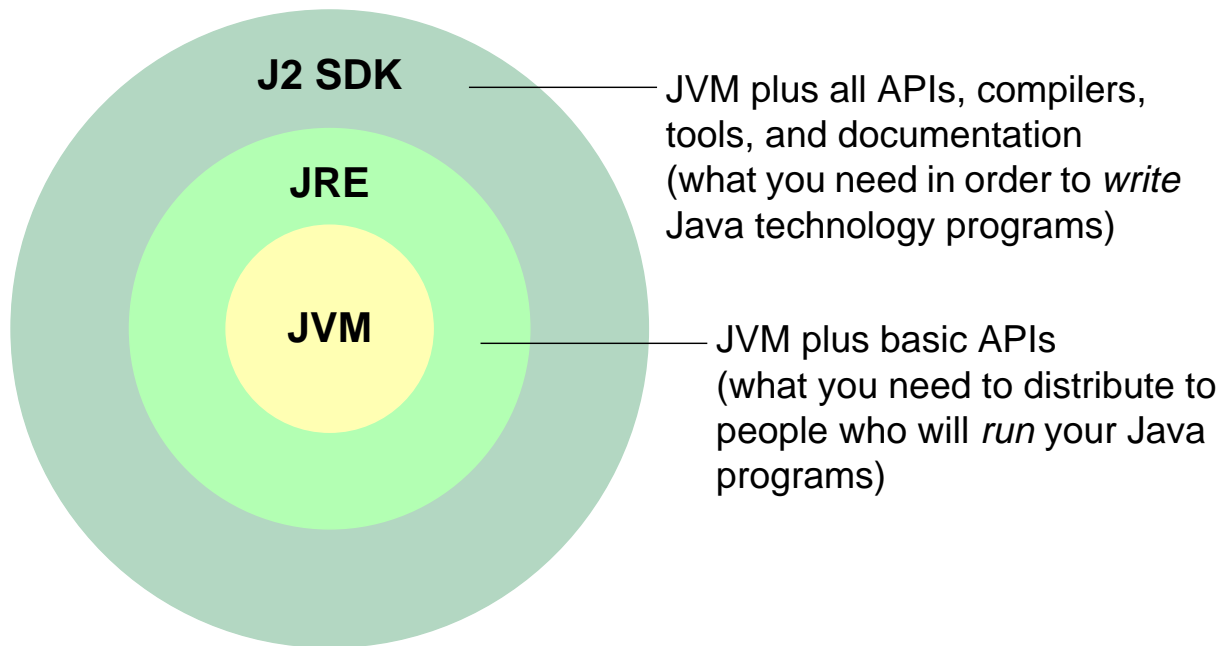


Figure 1-8 Relationship Among Three Main Java Technology Products

Note – Throughout this course, you will see references to “Java technology” and “the Java programming language.” They are not official Java technology terms, and are interchangeable; they are simply required for trademark purposes.



Sun Educational Services

Java API Documentation

- Detailed information API
- Very valuable resource: download, or view online at:

<http://java.sun.com/j2se/1.3/docs/api/index.html>

Java API Documentation

A Java application programming interface, or API, also includes very detailed documentation, referred to as the API specification or documentation. Every API includes documentation, which describes what the classes do and how to use them in your programs.

This is the best source for information about what ready-made code there is in the Java API, when you are looking for a way to perform a certain set of tasks.

To view the specification for the standard edition API online, go to the following site.

<http://java.sun.com/j2se/1.3/docs/api/index.html>

(To download it, see the instructions in the “Technical White Papers” section of Appendix C, “Where Do You Go From Here?”)

Figure 1-9 shows a common data-storage class.

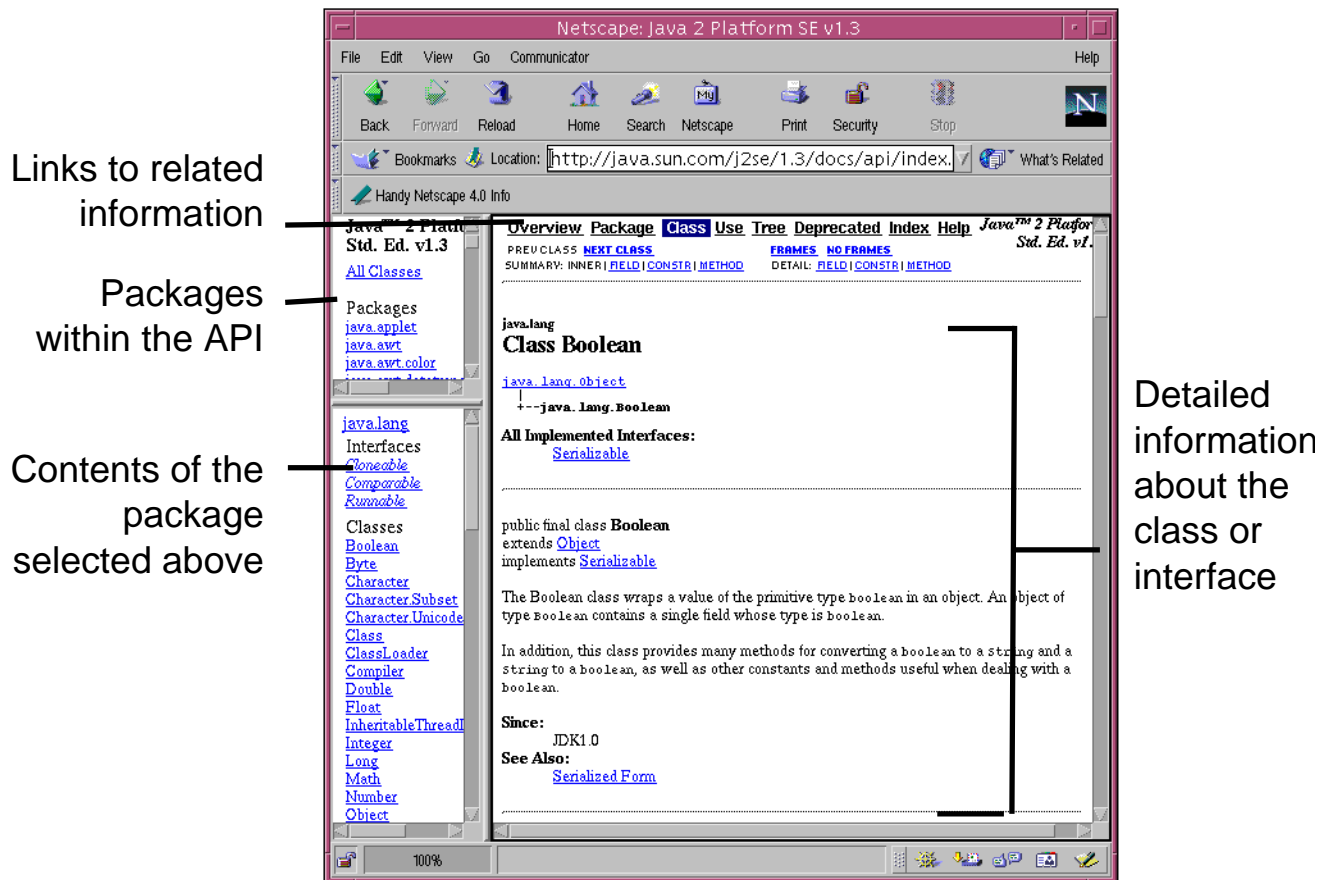


Figure 1-9 Java 2 Platform Specification, java.lang Package, Boolean Class

Note – The specification might look a little complicated and technical at this point. However, by the end of the class you will use this information.

Exercise 2: Using the API Specification



Exercise objective – Become familiar with the specification.

Tasks

Go to the following URL, or to one specified by your instructor:

<http://java.sun.com/j2se/1.3/docs/>

1. From the site listed, find the API specification.
2. Using the API specification, complete the following:
 - a. Find the `Math` class under `java.lang`. How many methods are there in this class?
 - b. What class does every class refer to at the top of the page?
 - c. Find the `String` class. What method is there to change the value of a `String`?
3. List five items at the original URL that you will use as resources when you leave class.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ List your expectations for the course, based on the information presented in this module
- ☐ Describe when and why Java technology was originally developed
- ☐ List at least four items that make Java technology superior to other programming languages
- ☐ Describe the difference between an applet and an application
- ☐ Describe the processes Java technology uses that make it cross-platform, and how this makes it different from other programming languages
- ☐ List two essential components of Java technology
- ☐ List the software you need to write Java programs, and that users need to run those programs

Think Beyond

How would you begin to analyze a system that you want to automate?
Think in terms of objects doing all the work.

Objectives

Upon completion of this module, you should be able to:

- Identify objects in a system
- Identify object attributes and operations
- Test the validity of an object
- Differentiate between an object and a class
- Describe the most significant difference between object-oriented and procedural programming
- Explain the main benefits of encapsulation
- List the main steps of software development

This module teaches you how to analyze a scenario and come up with a design on which to base an application.

Relevance



Discussion – How do you decide what the objects are in the system you are analyzing?

Could there be more than one design for the same system?

One of the chief advantages of OO is being able to reuse existing code. How could parts of an object definition be reused?

Additional Resources



Additional resources – The following references can provide additional details on the topics discussed in this course. They are relatively advanced sources; however, they contain valuable information.

- Larman, Craig. *Applying UML and Patterns – An Introduction to Object Oriented Analysis and Design*. Prentice-Hall. 1998.

This book covers the entire OO analysis and design process, in detail but without excess complexity.

- Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc. 1995.

This is considered to be the most widely read book on design patterns.



Sun Educational Services

Case Study

- Use this example to apply OOAD principles in this course.
 - ▼ Name of company: DirectClothing Inc.
 - ▼ Catalog produced monthly.
 - ▼ Customers can call, mail, or fax.
 - ▼ Inventory levels must be checked for each item.
 - ▼ DirectClothing accepts checks and credit cards.

Case Study

The following case study is used in this course to illustrate object-oriented principles. You will analyze this example throughout the course and through it learn to apply object-oriented analysis and design to systems you want to write programs for.

Note – Do not be intimidated if this seems complex; you will walk through the analysis in class step by step.

DirectClothing Inc. sells clothing through catalogs. Business is growing 30 percent a year and they need a new order entry system. You have been contracted by DirectClothing to design the new system.

A catalog of clothing is produced each month and mailed to subscribers. The catalog has a section for close-out items, monthly specials, and the normal prices. A piece of clothing may be priced differently each month, so when a customer orders from the catalog you must know the catalog used to price the item correctly. If the customers catalog is older than six months, then the items are priced according to the current catalog.

To place an order, customers call a customer service representative (CSR), mail the order form, or fax the order form.

Orders mailed or faxed are entered by an order entry clerk.

DirectClothing, Inc. wants to expand and would like to give the customer the option of entering an order on-line through the Internet. The items available on-line will be priced according to the current catalog.

After the order is entered in the system, inventory levels must be checked for each item and, if possible, assigned to the order. If the items are back-ordered from the supplier, then the order will be placed on hold until the items arrive from the supplier. Once the order has all items assigned, the payment must be verified; after verification, the order is sent to the warehouse to be assembled.

DirectClothing accepts checks and all major credit cards.



Overview of Object Orientation

- A technique for system modeling
- Attempts to describe a system as it exists in real life
- Identifies a system's:
 - ▼ Objects – A program asks objects to do work
 - ▼ Object attributes – Characteristics, what the object knows
 - ▼ Object operations – Tasks it can perform with what it knows

Overview of Object Orientation

The Java programming language is object oriented. Object orientation is a technique for system modeling, *system* being a situation that you need to write a program for, such as an online shopping system or all the processes and rules involved in doing accounting for a company.

This system is also referred to as a *problem domain*.

Object oriented analysis and design, or OOAD, attempts to describe a system as it exists in real life.

- The system is modeled as a number of related *objects* that interact. Any system needs objects to do work. It “asks” its objects to do its work, just as you might ask objects in an online store to deliver goods to you and charge the cost to your credit card.
- All objects and classes have *attributes* (characteristics), such as size, name, shape, and so on. This is sometimes referred to as *state*.
- They also have *operations*, the things they can do, such as start, show a blank screen, restart, and so on. This is sometimes referred to as *behavior*.

In the DirectClothing example, objects could include orders, customers, and so on. If the company sells a lot of shirts (or other types of garments), you might want to include a Shirt object in your design. If, on the other hand, shirts are not a significant factor in your business and you only have one kind of shirt, you might want to simply make a Clothing object.

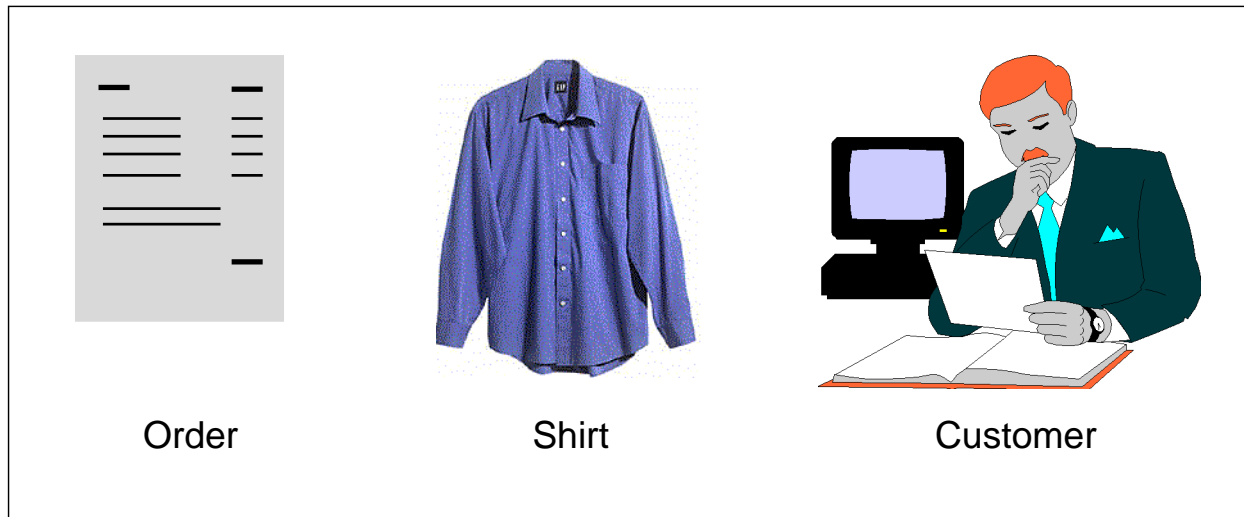
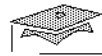


Figure 2-1 Possible Objects in DirectClothing Case Study

Note – Identifying objects in a problem domain is an art, not a science. Objects depend on context and on the viewpoint of the person doing the modeling and on the users.



Identifying Objects

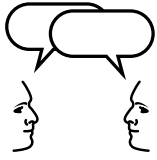
- Look for the main nouns in the problem domain description.
- Objects can be:
 - ▼ Simple or complex (shirt, bank)
 - ▼ Real or conceptual (bank teller, bank account)
- Objects have:
 - ▼ Attributes
 - ▼ Operations

Identifying Objects

Nouns are a good place to start to determine the objects in a system, but keep in mind that some nouns are variations of other nouns, express actions or events, or are attributes like color. The primary problem and challenge in object-oriented development is finding the right objects to model the system being represented.

Objects have the following characteristics:

- Objects can be simple or complex – A shirt and a bank can both be objects.
- Objects can be real or conceptual – A customer's account is a concept rather than a physical entity (it has related items that are physical objects such as a number and transactions). A bank teller could also be an object.
- Objects have attributes and operations – Attributes are characteristics like color or size; operations are what the object can do, like place or cancel orders.



Discussion – What are possible objects for DirectClothing, Inc.?



Identifying Object Attributes and Operations

- Attributes: Characteristics of an object
 - ▼ Can be data or an object
 - ▼ For Order, can include orderID, items
- Operations: Actions that the object performs
 - ▼ Can be something done by or to the object
 - ▼ For Order, can include place, cancel

Identifying Object Attributes and Operations

Now that you have learned to define objects, you need to specify their attributes and operations.

As described previously, attributes are characteristics, and operations are actions. You can think of attributes and operations this way:

- Attributes know something
- Operations do something with what the attributes know

Identify the things you need to have done, and the data needed to do it. Then figure out which objects you want to do the tasks, and which objects should have the data.

Attributes are often data, like order ID and customer ID for an Order object. Attributes can also be another object, such as the entire Customer object rather than just the customer ID.

Operations can be actions that the object does, often affecting its attributes. Operations can be done by the object, or to the object.

For example, an order can be placed, printed, have an item changed, and so on. (The customer or customer service representative would be initiating those actions, in real life, but the operations belong to the Order object.)

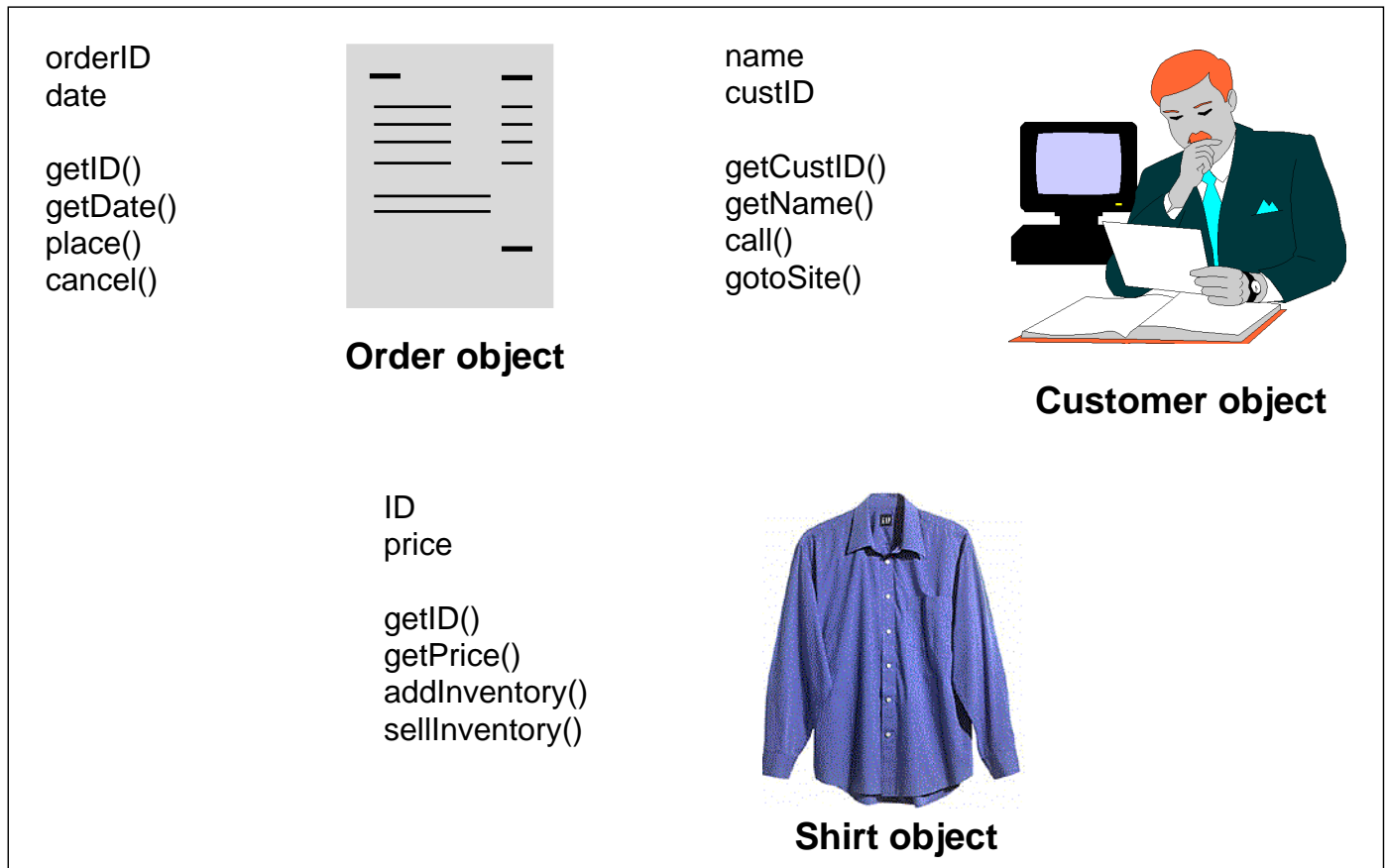


Figure 2-2 Possible Attributes and Operations for Objects in DirectClothing Case Study

Modeling Objects

You can *model* (visually organize) objects like Order using the example shown in Figure 2-3:

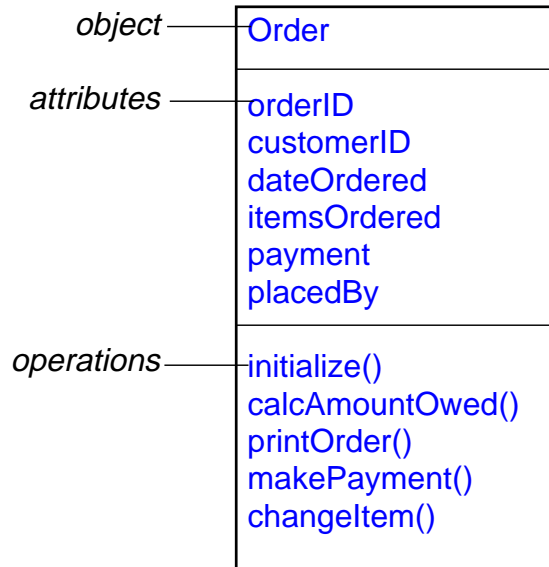


Figure 2-3 Basic Format for Modeling Objects

Note – These are possible attributes and operations; there are several possible workable designs for most problem domains.

The convention of a set of closed parentheses is used to indicate an operation when you model objects. The closed parentheses appear in Java programs, though not necessarily in referring to them. (In subsequent modules, you will see statements in the text about using `main` rather than `main()`).

Types of Attributes

An attribute can be a reference to another object. For instance, the `customerID` attribute for `Order` contains only the ID. If you changed it so that the `Customer` object is now an *attribute* of the `Order` object, the `Order` object now has access to all of the information in the `Customer` object.

Figure 2-4 models this example, showing the `Order` and `Customer` objects.

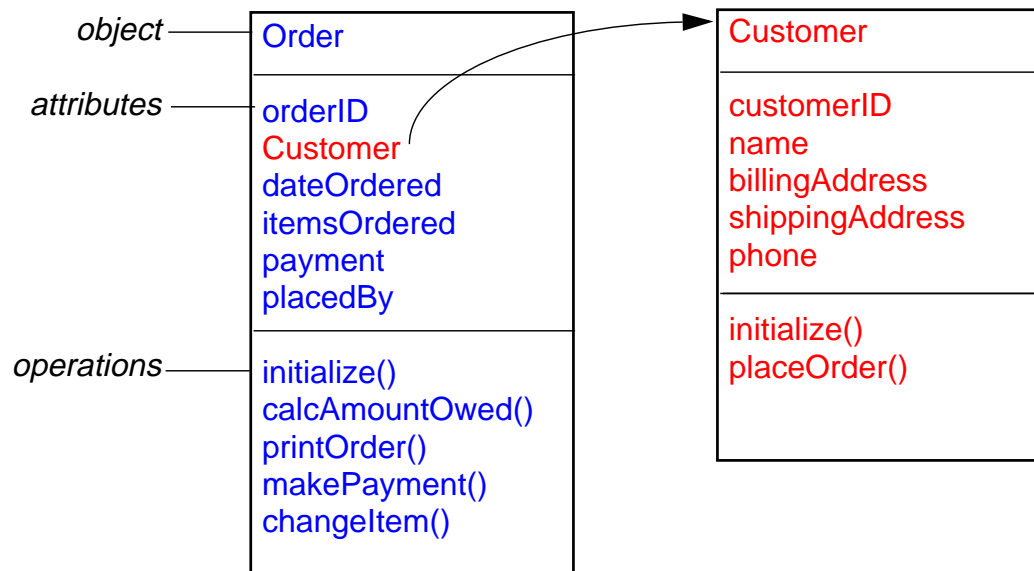
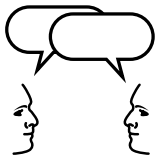


Figure 2-4 Object With Another Object as an Attribute

Discussion – What are possible attributes and operations for the objects in DirectClothing, Inc.?



Notes



Testing an Object

- Relevance to the problem domain
 - ▼ Does it exist within the problem statement?
 - ▼ Is it required, for the system to do what it needs to?
 - ▼ Is it required, for user and system to interact?
- Independent existence
- Attributes and operations

Testing an Object

Once you have selected objects in the problem domain, evaluate them to be sure each is a valid object. Use the following criteria to test whether something should be an object:

- Relevance to the problem domain
- Independent existence
- Attributes and operations

Relevance to the Problem Domain

To determine if the object is relevant to the problem domain, ask yourself the following:

- Does it exist within the boundaries of the problem statement? (The problem statement is the description of the problem domain; the case study on page 2-4 is a problem statement.)
- Is it required in order for the system to fulfill its responsibility?
- Is it required as part of an interaction between a user and the system?

Note – Be aware of context; objects can sometimes be a characteristic of other objects or objects themselves. Temperature might be just an attribute of a travel agency system, for example, but might be a central object in a scientific system tracking weather patterns.

Independent Existence

In order for an object to be an object and not a characteristic of another object, it must exist independently in the context of the problem domain. Objects can be connected and still have independent existence. A customer and an order are connected, but are independent of each other, so both would be objects for the DirectClothing case study.

When evaluating potential objects for this characteristic, ask if the object needs to exist independently, rather than being an attribute of another object.

In the case study, can an Order object exist without any of the other objects? It can, but in use, it must have an associated Customer object.

Address could be an attribute of Customer, but in this case study it is advantageous for Address to be a separate object.

Attributes and Operations

An object must have attributes and operations. If you cannot define attributes and operations for an object, then it probably is not an object but an attribute or operation of another object.



Discussion – Do the objects in the DirectClothing scenario pass the tests described in this section?

Exercise: Analyzing a Problem Domain



Exercise objective – Find objects, attributes, and operations in a sample problem domain.

Tasks

Follow the directions from your instructor to locate the exercises directory for this module.

Complete the exercise that your instructor indicates: either the soccer statistics case study or the dog pound case study.

To create the design, do the following:

1. Create a list of the potential objects (nouns).
2. Create a list of attributes and operations to your objects.
3. Assign the attributes and operations to the appropriate objects.
4. Apply the three tests used in this chapter to decide if all your objects are valid.

Modeling a Dog Pound

The dog pound in Fargo, North Dakota needs an object-oriented Java application to track its activities. The city commissioner presented you with the following information.

The dog pound collects stray dogs and has a capacity of 50 dogs. Three trucks collect the dogs, each with a capacity of 5 dogs. If the pound is over capacity, it reduces the fee for picking up a stray dog or adopting one, and issues tickets to owners rather than picking up dogs.

When a dog is checked into the pound, the pound uses the tags, if present, to find out its name, breed, owner, and any other information. The dog is assigned a name if one cannot be determined, and an ID. The dog is then examined to determine whether it has any infectious diseases or needs shots.

The pound is in charge of feeding the dogs, and needs to keep enough food on hand to feed all the dogs. The amount of food each dog needs depends on its size and age.

After a dog has been at the pound a month, it changes status from waiting to be picked up, to ready for adoption. There is a \$40 adoption fee. People come to the pound, either to retrieve their dog (in which case they must pay a fine and for any medical care the dog has received) or to pick out a new dog and pay the adoption fee.

Tracking Soccer Statistics

Read the following case study, then model the system by choosing objects, attributes, and operations.

Your task is to produce an object-oriented design for a Java application that tracks soccer scores. The program should track:

- The number of goals each player scores in each game
- What teams the players play for and what season the games were played in

The application should be able to generate statistics for teams, players, and seasons.

Notes

Notes

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences

- Interpretations

- Conclusions

- Applications



Sun Educational Services

Classes

- An object is an instance of a class.
- A class:
 - ▼ Is a category or template
 - ▼ Can be compared to a cookie cutter or house blueprint

Classes

Up to now, for simplicity, this module has described both objects and classes as objects. The difference between them is as follows:

- A class is how you define an object. Classes are descriptive categories or templates. Shirt could be a class.
- Objects are unique *instances* of classes. The large blue polo shirt that costs \$29.99 with item ID 62467-B is an *object* of the *Shirt class*, as is the small green shirt with the same price and item ID 66889-C, or the patterned shirt for \$39.99, ID 09988-A.

Classes and objects are illustrated in Figure 2-5 on page 2-26.

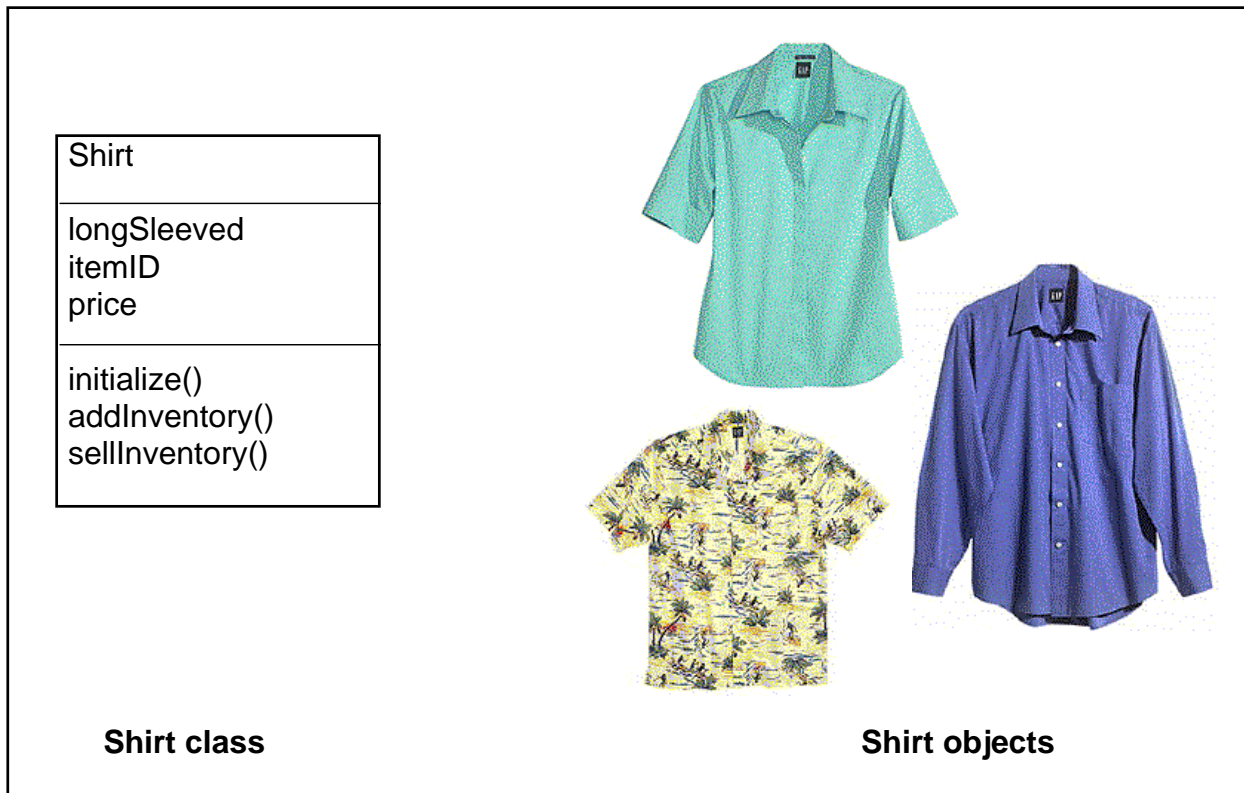
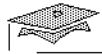


Figure 2-5 Classes and Objects

The attributes and operations defined by a class are for its objects, not for itself. There is no concrete realization of the class `Shirt`, but there are `Shirt` objects.

A class can be compared to a blueprint. Imagine you are in charge of building a housing development, with one housing blueprint. Each house in a development is shaped the same way, but some have brick or aluminum siding, some have custom paint colors inside and some are just white, and so on.



Object Orientation and Encapsulation

- Java technology uses OO; older programming languages use a procedural approach.
- OO unifies a class with its attributes and operations; procedural programming separates operations.

Object Orientation and Encapsulation

Programmers use object orientation to write Java applications; other older programming languages such as Fortran and C use a procedural approach instead. This section contrasts the two approaches to show the advantages of object orientation.

Object orientation is how the Java programming language looks at a system; procedural programming is how older languages such as Fortran and C look at problems.

The key difference between OO and procedural programming is that in OO, each class has its own attributes and operations. The class's operations can therefore apply programming logic to make sure the operations work appropriately with the attributes they are used for, and that inappropriate values are not applied to the attributes. Each object has access to its own set of attributes and operations.

Procedural programs do not link operations with classes; classes have specific identified attributes, but the operations are not associated with any particular class.

The example on the following pages shows the disadvantage of procedural programming and advantages of OO programming.



Sun Educational Services

Example: Comparing Procedural and OO Approaches

- Several teams of programmers are writing an accounting system.
- One team writes code for items and inventory.
- One team writes code for orders.

Example: Comparing Procedural and OO Approaches

Unifying a class's attributes and operations, using OO, enables programmers to create consistent code with reduced errors that is easy to maintain. Imagine a programming team, working on the DirectClothing order-entry and delivery system. There is a team working on how the order is processed, and a separate team working on how inventory is checked and items are processed.

Procedural Problems

This section shows the mistakes that can be made when programmers do not use OO.

If the programming teams take the procedural approach, they will have coded objects with corresponding attributes, as well as separate operations (shown in Figure 2-6).

Assume a programmer on the Order team needs to write code to make sure an item is in stock before an order can be processed. He or she might do so by directly changing the value of the quantityInStock attribute for the item, setting it to 56, for example.

The code to verify whether that many items are actually available was created by the Inventory team. In the procedural approach, the Order programmer is not required to use the verifying code in order to do his or her work.

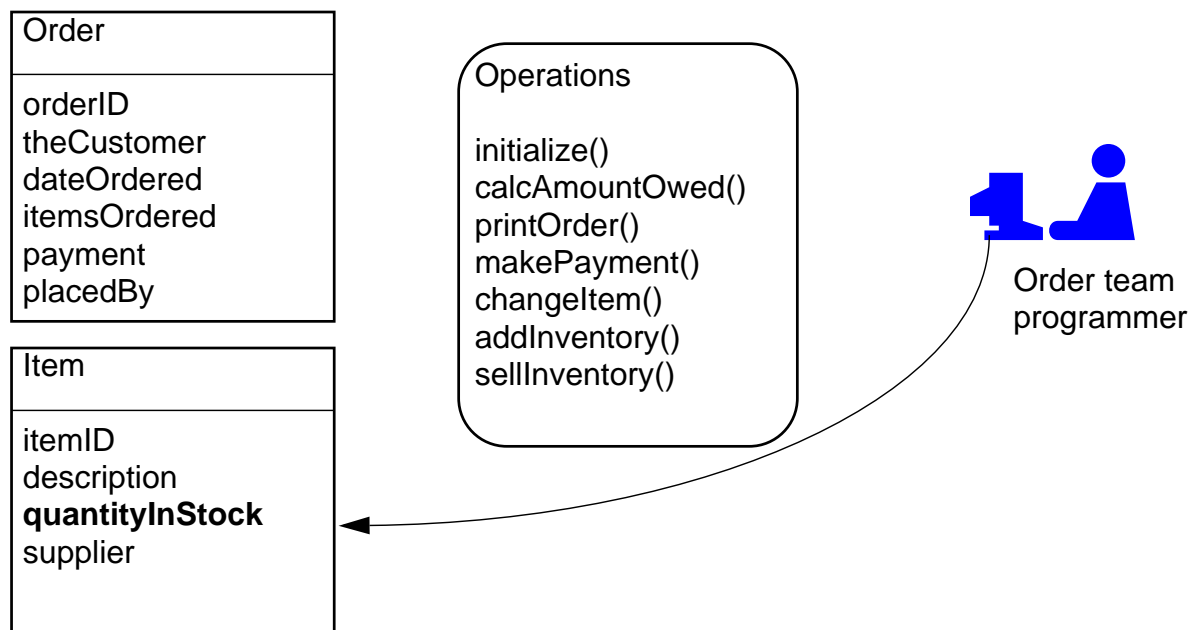


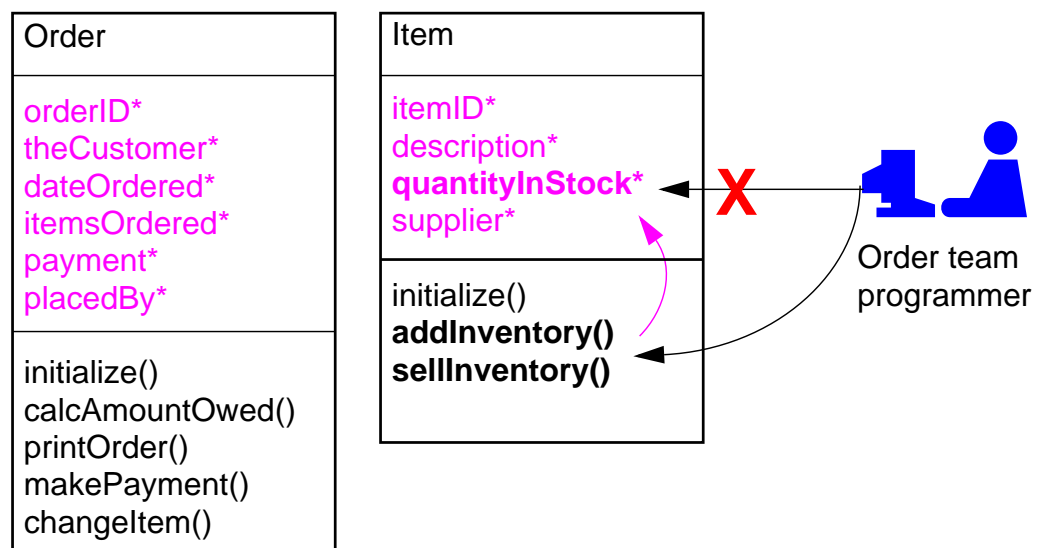
Figure 2-6 Separate Operations

If this happens, an order could be sent out without an item the customer ordered, but the customer would be charged for it.

Object-Oriented Solutions

These mistakes could not happen in a well designed, well programmed OO application. In OO, the programmer on the Order team would not be able to directly change the data of the Item object. Its attributes would be hidden, or *private*, only accessible by methods of that object's class. The methods are not hidden; they are *public*. The Order programmer can use the `addInventory()` and `sellInventory()` operations from the Item class.

However, the operation is controlled by the Inventory team and includes error-checking to make sure enough items are in stock, and the item would only be included on the order, and charged to the customer, if it were actually in stock.



*private attributes

Figure 2-7 Including Operations and Attributes in Each Class

This solution is called *encapsulation*. Encapsulation separates the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects.



Sun Educational Services

Advantages of Encapsulation

- Objects protect their own data.
- Implementation is controlled by the person who programs the class.
- Objects are managed through their operations.

Advantages of Encapsulation

The beauty of object orientation is that it enables you to look at the problem domain in a simple way which ultimately leads to simpler coding, maintenance and re-use of common code.

Encapsulation provides the following advantages:

- Objects protect their own data.
- Implementation is controlled by the person who programs the class, so anyone else using that code does not need to know how all the details of how it works. (They need to know information like the data, if any, to pass to a method, for example, but not more detailed information.)

In the example, the Order team uses Item objects, and does not write code to control how the Item object manages its state.

- Item objects are managed through their operations.



Interface and Implementation

- Make changes to one part of a program without affecting links to other parts.
- Other parts of the program interact with the interface (the "outside") of a class.
- Implementation ("the inside") can change without affecting interface.
- A car is encapsulated:
 - ▼ Implementation details are under the hood; driving does not require knowing how the car works.
 - ▼ Parts of car can be changed or replaced without changing how the car needs to be driven.

Interface and Implementation

Another advantage of encapsulation is that you can make changes to how things work and make changes to only a few places in your code, rather than updating an entire application.

You can change the way a piece of code works, and the user of that code need not know about it as long as the interface remains the same. You can liken it to using a class in the API—you don't know at this level how any of the classes you have looked at so far work "under the hood." The fundamental workings may change in the next release of the language but you do not care as long as the API publishes the interface for using the code.

For example, assume the Inventory team wanted to change the `isInStock` attribute from a simple attribute that says yes or no, to an attribute called `stockStatus` that returns whether the item is in stock, back ordered, or discontinued. The Order team does not need to know even that it has changed, much less change their code. The Item team makes the change in the code defining the Item class, and the code in the rest of the application stays the same.

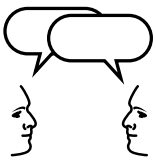
Changes to the Order operation `isInStock()` might return true or false according to a whole new set of internal tests, which the person calling the operation knows nothing about. Thus the internal workings are hidden, or encapsulated.

This separation of the outside and the inside, essentially, of a class is called the *interface* and the *implementation*. The interface is how code outside the class interacts, or interfaces, with the class. The implementation is what goes on inside. Encapsulation lets programmers change their implementation easily, as long as the interface stays the same.

Consider the example of a car. In the real world (not programming), cars are very complex pieces of machinery. Most of us do not know all the details of the implementation of how a car works. Luckily, all the complex implementation details of a car are hidden under the hood of the car (private.) We are generally encouraged not to touch the private implementation under the hood. Instead, we are given a very easy “public” interface made up of a steering wheel, pedals, and a few switches. This “public” interface is an easy way for us to use the car without having to know all of the implementation details of the car. Thus, encapsulating cars provides the following advantages:

- The other class (driver) does not have to know the implementation details of the class (car) to be able to use it. Thus, it is easier to use the class.
- The other class cannot improperly use this class, it is only allowed to do what the interface allows it to do.
- You can change the implementation of the class without changing the interface, and the other class doesn’t need to be rewritten. For the car analogy: You could replace my gasoline burning engine with a more efficient electrical battery. If you do not change what the steering wheel, pedals, and switches do (interface) the driver does not have to be retrained even though the implementation changed.

Discussion – What are some real-life examples of encapsulation, when you interact with an interface and do not need to know how the implementation works? Discuss non-programming examples.





Re-Use

- Pre-written operations can be reused throughout project.
- Easier to re-use than to re-write.
- Write small, simple operations: `pay()`, `bonus()`, `tax()`, and `deduct()`, rather than `payrollFunctions()`.

Re-Use

Re-use, a primary advantage of object orientation, is evident in encapsulation. The operations that are written ahead of time include error checking and other features, so it is easier and more productive to re-use that code rather than writing new code. In addition, it forces programmers to re-use the limited number of operations in the interface, because they cannot access the data any other way.

To re-use code as much as possible, it is a good idea to write operations with as little, and as simple, functionality as possible. This makes it more likely that the operations can be re-used for multiple parts of a program.

For example, you might write a single operation in a payroll system that runs all the functions necessary each payday: adding bonuses and salary, taxing, deducting for insurance, and so on. However, it would be re-usable to write separate operations, which you could use separately, or together in any groupings or order you choose:

- `pay()`
- `bonus()`
- `tax()`
- `deduct()`



Software Development Stages

1. Gather and analyze requirements.
2. Design the system.
3. Write the system.
4. Test all aspects of the system.
5. Have customers and users test the system.
6. Maintain the system.

Software Development Stages

The object-oriented analysis is just one part of any programming project. The following are the basic six steps you should take in most projects, regardless of the size. In this module, you have already learned steps 1 and 2.

All six steps are important. If you simply begin coding without the analysis and design, you will encounter reasons to change how the system works and waste time rewriting code. If the programming team is large, these problems are increased dramatically.

Each step includes documentation: documenting the requirements, the design, how to use the program, test cases and results, and maintenance documents so that other team members can more easily know what the code does and how to update it or fix it.

1. Gather and analyze requirements.

This is the process of compiling and reviewing the problem domain statement, like the one on page 2-4.

2. Design the system.

This is a lengthy process, primarily composed of the OOAD shown in this module.

The following Sun Microsystems course concentrates on this phase: OO-226 – *Object-Oriented Application Analysis and Design for Java Technology (UML)*

3. Write the code for the system.

Once the design is in place, the actual coding begins.

In order to break down this step and make it more manageable, two approaches are used.

- ▼ Break the problems down into even smaller chunks to make the coding simpler.
- ▼ “Pseudocode” is also used in some development projects: writing in English the exact steps that need to take place in the program. These steps could include accessing a piece of data, retrieving it, passing it to an operation, and so on. The programmers would then translate that to the Java programming language.

4. Test all aspects of the system.

Testing is a crucial part of development. If you have used a program with many bugs or one that did not do what it was supposed to do, you know how important good testing is.

Testing time can be reduced if OOAD is effectively implemented in the design and code phases.

5. Have customers and users test the system.

The team delivers the system to a small group for “real world” usage; it is sometimes referred to as a beta test. This subjects the system to being used, not only as the team planned for it to be used, but how it really is used.

6. Maintain the system.

The same programmers, or sometimes a group of different programmers, add features and fix any additional problems in the system or documentation. OOAD can also make this stage of the process simpler and quicker.

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Identify objects in a system
- ☐ Identify object attributes and operations
- ☐ Test the validity of an object
- ☐ Differentiate between an object and a class
- ☐ Describe the most significant difference between object-oriented and procedural programming
- ☐ Explain the main benefits of encapsulation
- ☐ List the main steps of software development

Think Beyond

The next stage in software development is step 3, coding the project. What are some of the basic elements of code you would need to write? For example, you would need the ability to write code that assigns a value to an attribute, such as assigning \$23.15 to the Shirt attribute price.

Objectives

Upon completion of this module, you should be able to:

- Compile and run a basic Java program
- Describe how computers store data using the binary system
- List the eight Java technology primitive types
- State the storage values for the primitive types
- Create identifiers according to Java technology rules and coding standards
- Write programs that follow Java technology rules and recommended coding standards
- Write code to declare and assign literal values, expressions, and variables
- Write code to declare and assign data to a constant
- Describe how variables and constants are stored in memory

This module describes how data is stored in programs, and the structure in which to write basic Java technology code (“Java code”).

Relevance



Discussion – Discuss the following questions:

- What can humans do better than computers?
- What can computers do better than humans?
- Is a computer or a human more intelligent? Why?

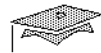
Additional Resources



Additional resources – The following references can provide additional details on the topics discussed in this module:

- Farrell, Joyce. 1999. *Java Programming: Comprehensive*.

This is an excellent book for non-programmers; it explains concepts that are glossed over in more advanced books.



Sun Educational Services

New Terminology

- In Java technology:
 - ▼ Variable = attribute
 - ▼ Method = operation
- Classes are still called classes.

New Terminology

The object-oriented analysis and design field uses the terms attributes and operations for an object's characteristics and its actions. In programming, however, those terms are different:

- Attributes are called *variables*.
- Operations are called *methods*.

Classes are still called classes.



The First Step in Writing Java Applications

- OOAD module showed OO design, with classes represented as diagrams.
- The rest of this course teaches how to represent the same classes, using the Java programming language.
- This module uses *procedural* approach for simplicity:
 - ▼ Classes and variables are still included in each program.
 - ▼ Instead of class-specific methods like `order`, one "all-purpose" method controls all actions.

The First Step in Writing Java Applications

In Module 2, "Object-Oriented Analysis and Design," you spent a lot of time analyzing a problem domain from an object-oriented point of view. You represented problem domains using diagrams such as the ones you created for the online clothing store.

In this and subsequent modules, you will represent similar objects, but using the Java programming language instead of diagrams. You will write code to represent the class and variables. However, for now you will not use class-specific methods such as `order()`. To simplify the process of learning to program, class-specific methods are included later in this course.

You will use the "all-purpose" Java technology method ("Java method") called `main` to initiate all actions in your programs. Using this approach of putting *all* actions inside one method is sometimes referred to as procedural programming.

Once you get comfortable with writing basic Java code, then you will learn how to use different methods and classes in your programs, which will make them fully object oriented.

Basic Java Application

The following example shows a basic application that sets the values for a shirt's catalog ID and price, and prints the information to the screen.

The same types of information that you specified when you designed the problem domain now appears in a different structure, the program source code.

- Class name, `Shirt` (line 1)
- Variables such as `price` and `ID` (lines 3 and 4)
- Methods; the method used here is the "all-purpose" Java method, `main` (lines 5 through 9)

By viewing this example, you will see the structure and components of the program.

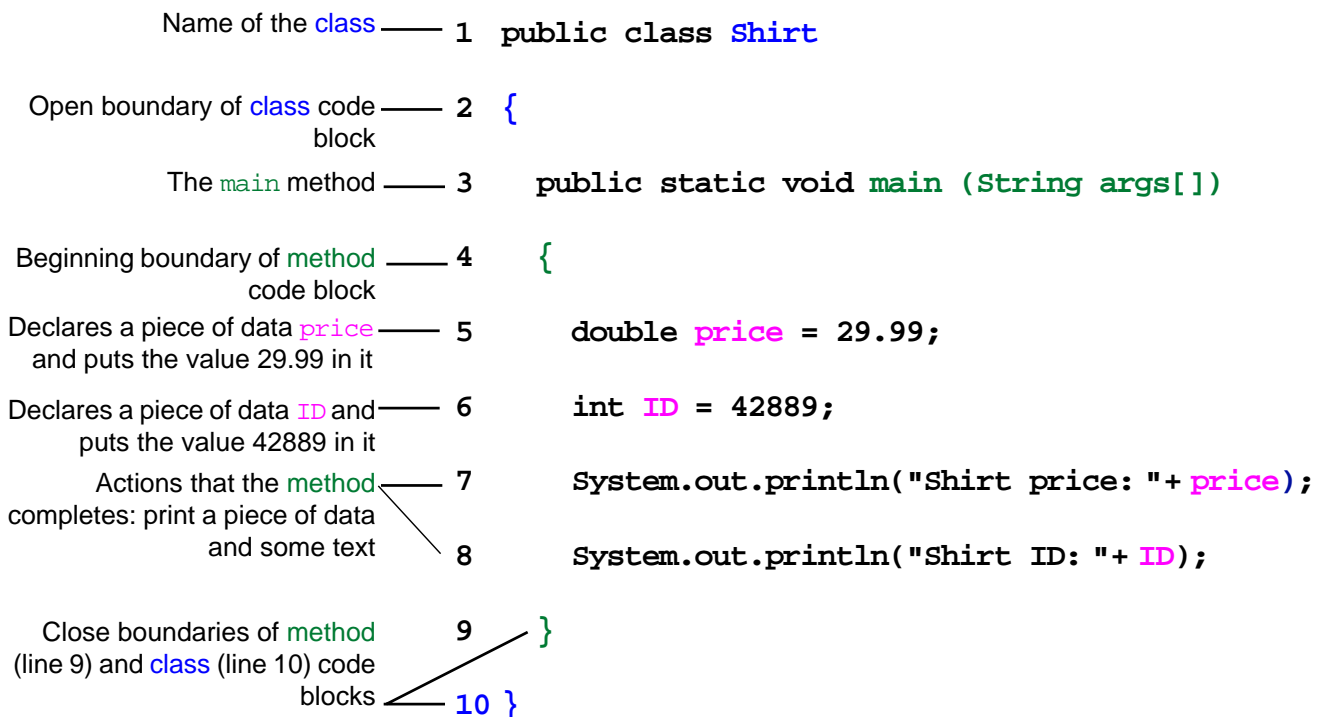


Figure 3-1 Basic Procedural Java Program



Primary Application Components

- The class code block is the primary structure of the program.
- The variables are the data the program works with.
- The method code block (`main` in procedural programs) is the structure for the program's actions.
- Use braces (`{` and `}`) to define code blocks.
- Use semicolon at the end of an action.

Primary Application Components

The most important parts of most Java applications are:

- The class code block
- The data (variables)
- The method (`main`)

You will learn a lot more about each category as you continue through this course; however, this is the essential information about the building blocks of a Java program.

Class Code Block

All programs need to have a class in them, as the primary structure of the program. Inside the class code block (marked by the beginning and ending braces, shown in blue in lines 2 and 10), you declare the variables (or attributes, from OOAD) and the methods.

Data (Variables)

These are the attributes, such as an item's ID or price, that the program needs to know in order to do its work. You need to declare, or set up, the names of the variables (like `price`) somewhere within the class code block or method code block. (With `main`, for now, put them within the method code block, as shown.)

Method Code Block (main in example program)

Methods are the same items you identified in Module 2, "Object-Oriented Analysis and Design" and called operations. For now, however, to keep things simple, instead of different methods for each class, all the programs you write will use the "all-purpose" method that is available to all Java programs, `main`. All Java classes can have a `main` method, declared exactly in this way. It is the starting point of the program, the first method that will be run (executed by the JVM). The JVM starts any program with `main`.

Procedural programs put all procedures in the `main` method.

Using Semicolons and Braces

In addition to these three main parts, note that the lines in the program that do something, like printing or declaring the value of variables, have a semicolon at the end of each line. In general, there is a brace (`{` or `}`) or a semicolon at the end of each line in a Java program line that is not the beginning of a code block. The syntax and examples throughout this course show you when to use each.



Sun Educational Services

Compiling and Running a Program

- Source file requirements: only one public class per source code file
- Compiling: `javac filename.java`
- Results: `public_class_name.class`
- Running: `java filename` `java MyClass`

Compiling and Running a Program

You have seen a basic Java program and the main parts; once you learn how to compile and run it, you will be able to start writing programs.

Requirements for Your Source File

The source file name *must* match the public class name in the source file, and must have a `.java` extension. For example, assume there is a source file named `Order.java`. It must contain only one public class statement, such as:

```
public class Order
{
    //class statement code
}
```

You can have only one public class in a source file.

Compiling

Compiling, as you learned in “How Java Technology Solves the Communication Problem” on page 1-22, turns the program you write into runnable bytecode.

How to Compile

At the command line, go to the directory where your source files are. Type the following:

```
javac filename.java
```

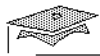
Results

When compiled, the result will be *class_name.class*.

Running the Program

At the command line, go to the directory where your source files are. Type the following (only the name up to *.class*, not the *.class* extension):

```
java filename
```

Debugging

- Check line referenced in error message
- Check for semicolons
- Check for even number of braces

Debugging

You will almost always have at least one error in any code you write. Use the following tips when debugging.

- Error messages state the line number where the error occurs. That line might not always be the actual source of the error. Also, keep in mind that fixing one error can cause others to show up or occur.
- Be sure you have a semicolon at the end of every line where one is required, and no others.
- Be sure you have an even number of braces: a closing brace for each opening brace.
- Be sure that you have used consistent indenting in your program, as shown in examples in this course. Your programs will run without indenting different types of code. However, indenting makes the code easier to read and debug, and makes it easier to avoid mistakes while you are writing.

Exercise 1: Running, Compiling, and Modifying a Basic Program



Exercise objective – Become familiar with the structure and parts of a basic (procedural) Java program. Use the instructions in this exercise and the information in “Compiling and Running a Program” on page 3-9 of your student guide; also refer to the example program in “Basic Java Application” on page 3-6.

Tasks

Follow the directions from your instructor to locate the exercises directory for this module.

Run and Modify the Shirt Program

1. In File Manager, go to the exercises directory for this module, as directed by your instructor. The name of the directory is 03_getstarted.
2. Open the `Shirt.java` file (the program source code).
3. Look at the code; it is the same as the example in this module.
4. Close the file.
5. Open a terminal window.
6. Navigate to the directory for this module.
7. Type the following on the command line to compile the program into an executable bytecode file. The file `Shirt.class` is created.

```
javac Shirt.java
```
8. When the prompt reappears, type the following to run the program:

```
java Shirt
```
9. Open the `Shirt.java` file again.
10. Change the value of the price to 14.99 and the ID to 112244.

11. Close the file, saving changes.

12. Compile the file again:

```
javac Shirt.java
```

13. Run the file again:

```
java Shirt
```

Write a New Program

Exercise objective – Write a basic program.

1. Create a new directory in the directory for this module, called work.
2. Open a text editor and save it, naming it `Quotation.java`. Save it in the work directory for this module.
3. Type the following in the file.

```
public class Quotation
{
    public static void main (String args[])
    {
        String quote = "Welcome to Sun!";
        System.out.println(quote);
    }
}
```

Note – Be sure to indent as shown; it makes the program much easier to read, and easier to debug.

4. Close the file and save it.

5. Open a terminal window and navigate to the work directory where your program is saved.

6. Type the following in the terminal window to compile the program:

```
javac Quotation.java
```

7. If error messages appear, open the file and verify that you copied the program exactly.

8. If no error messages appear, type the following in the terminal window to run the program:

```
java Quotation
```

9. Open the `Quotation.java` file again and change the "Welcome to Sun!" text to your own favorite quotation. Be sure to leave the quotation marks at the beginning and the end. Then compile and run the program again.



Sun Educational Services

Computer Data Storage

- Computers use 0s and 1s (binary) to store data.
- Humans use base 10.
- 0s and 1s are stored in bits (8 bits make a byte).
- A bit can only be on (1) or off (0).
- Values are evaluated by multiplying 2 to a power.

Computer Data Storage

This section covers how data is stored in variables.

All software and data are stored and manipulated within the computer in *bits*. In early computers a bit was represented by a valve that was either on or off. Modern computers use chip-based bits but the effect is the same—a bit is either on or off. Values are stored in data types of 8 bits, 16 bits, and so on. A byte is 8 bits.

Because a bit can store one of two values the computer counts in base 2 or *binary*. Computers think in terms of 2s and powers of 2.

Values are calculated by multiplying 2 to the power of the number of bits: 2^8 , 2^{16} , and so on.

- $2^1 = 1$
- $2^2 = (2 * 2) = 4$, and
- $2^4 = (2 * 2 * 2 * 2) = 16$



Computer Data Storage

- Java technology uses different data types that store different number of bits.
- Range of values: $(-2^{x-1} \text{ to } 2^{x-1} - 1)$ where x is the number of bits.
- A data type storing 8 bits would have:
 - ▼ 2^8 possible values
 - ▼ Highest and lowest values of (-2^7) and $(2^7 - 1)$
- Zero is positive.
- Leftmost bit is reserved for the sign: 0 for positive and 1 for negative.

The Java programming language uses various data types to store a different number of bits. Each type is based on 2 multiplied to a power equalling the number of bits. An 8-bit number has 2^8 possible values, or 256.

All but one of the bits are used as multipliers to determine the value. The farthest left, or highest, is reserved to indicate whether the number is positive or negative (0 for positive and 1 for negative). Therefore, the lowest possible value for a number is:

$-(2 \text{ to the power of } [\text{number of bits} - 1])$

An 8-bit number's lowest possible value, for example, is -2^7 .

The highest possible value for a number is:

$(2 \text{ to the power of } [\text{number of bits} - 1]) - 1$

1 is subtracted at the end because zero is counted as positive. (You begin counting positive numbers at 0, so counting 4 numbers takes you through 0, 1, and 2, to 3.)

The full range of the possible values for a number is $(-2^{x-1} \text{ to } 2^{x-1} - 1)$, where x is the number of bits.

Figure 3-2 visually represents the calculations to find the range of possible values for a data type.

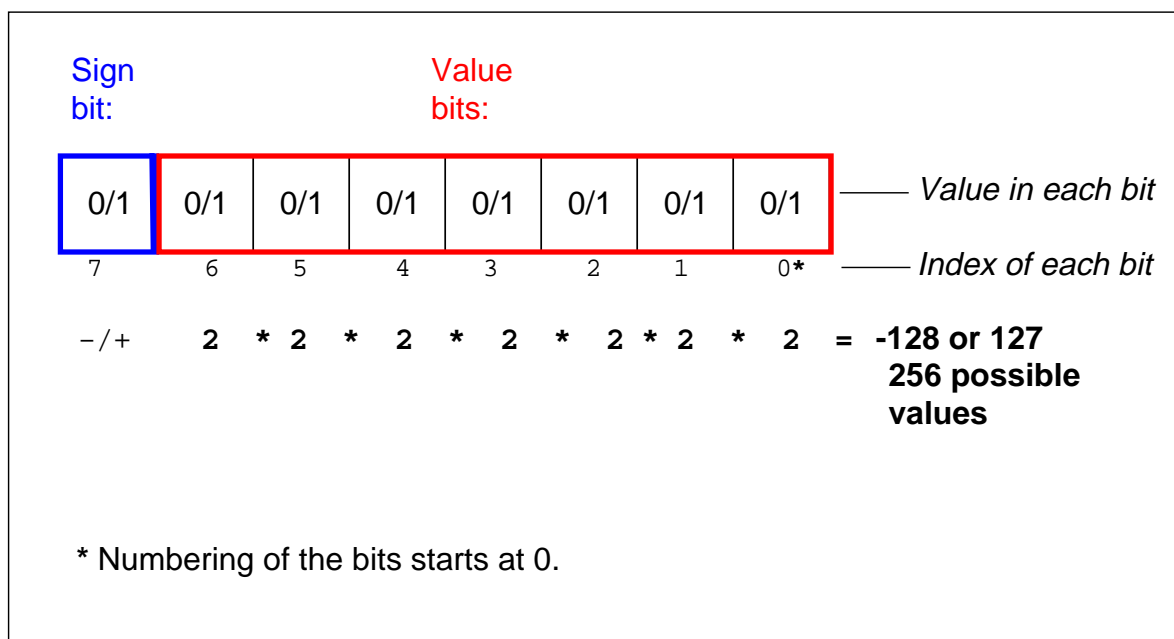


Figure 3-2 Determining Number of Possible Values, and Range of Values, for a Value

Figure 3-3 shows how you can calculate the numeric value of a number, converting it from binary to base 10.

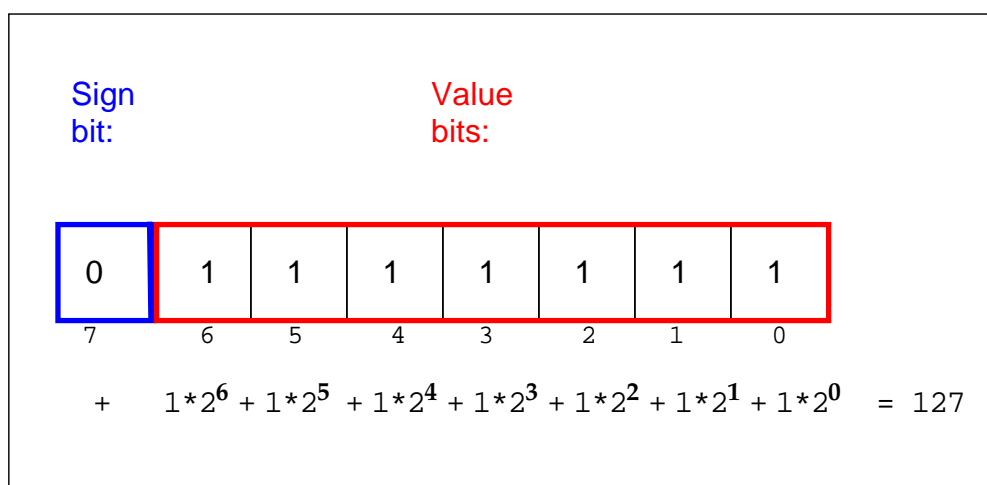


Figure 3-3 Calculating the Base 10 Value of Binary Data



Sun Educational Services

Storing Data in Variables Using Primitive Data Types

- Java technology uses *data types* with predefined:
 - ▼ Storage sizes
 - ▼ Kinds of data they can store
- Types are:
 - ▼ Integral types – byte, short, int, and long
 - ▼ Floating point types – float and double (default)
 - ▼ Textual type – char
 - ▼ Logical type – boolean

Storing Data in Variables Using Primitive Data Types

The variables, as seen in previous modules, contain different types of information. Some are names, some are numbers, some are “toggles” that indicate which one of two possible conditions are true (such as on or off). Each major type of information needs a different data type.

To define what type of information they hold, many of the variables in Java programs are stored as *primitive data types*, or *primitives*.

Primitive Data Types

These are the eight primitive types built into the Java programming language:

- Integral types – byte, short, int, and long

These types store numbers that do not have decimal points. Values such as a person’s age, the number of stars in the galaxy, the result of (4 – 7), or the number of years Westley was gone with the Dread Pirate Roberts can all be expressed as integral types.

- Floating point types – float and double (default)

These types store numbers that have decimal points. Values such as the price of a candy bar and the result of 937,713,499 divided by 4 (234,428,374.75) must be stored as floating point types.

- Textual type – char

This type can store any single character, such as e, 7, or ÿ.

(Storing a series of characters, such as your name, requires a different type that is not a primitive type. This is covered later in this module.)

- Logical type – boolean

This type stores either/or values, like whether an item is on sale, or in stock. They store 'true' or 'false' or a boolean expression such as $x > 4$.



Integral Primitive Types

- Store numbers without decimal places

Type	Integer Length	Range
byte	8 bits	$-2^7 - 2^7 - 1$ (-128 – 127, or 256 possible values)
short	16 bits	$-2^{15} - 2^{15} - 1$ (-16,384 – 16,383, or 32,768 possible values)
int	32 bits	$-2^{31} - 2^{31} - 1$ (-2,147,483,648 – 2,147,483,647 or 4,294,967,296 possible values)
long	64 bits	$-2^{63} - 2^{63} - 1$ (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, or 18,446,744,073,709,551,616 possible values)

Integral Primitive Types

There are four integral primitive types in the Java programming language, identified by the keywords `byte`, `short`, `int`, and `long`.

If you need to store people's ages, for example, a variable of type `byte` would suffice because you know from Table 3-1 that it can cope with values in that range. However, many values require more storage space to store their values, which other types define.

Table 3-1 lists all the integral types, their sizes, and the range of possible values.

Table 3-1 Integral Types

Type	Length	Range	Examples of Allowed Literal Values
byte	8 bits	-2^7 to $2^7 - 1$ (-128 to 127, or 256 possible values)	2 -114
short	16 bits	-2^{15} to $2^{15} - 1$ (-32,768 to 32,767, or 65,535 possible values)	2 -62699
int	32 bits	-2^{31} to $2^{31} - 1$ (-2,147,483,648 to 2,147,483,647 or 4,294,967,296 possible values)	2 147,334,778
long	64 bits	-2^{63} to $2^{63} - 1$ (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, or 18,446,744,073,709,551,616 possible values)	2 -2,036,854,775,808L 1L

When you specify a literal value for a `long`, put a capital L to the right of the value to explicitly state that it is a `long`. Literal values for integral types are sometimes assumed, by the compiler, to be `ints` unless you specify otherwise, using the L.



Floating Point Primitive Types

- Floating point types store numbers with decimal places.
- Highest or lowest values cannot be determined.

Type	Float Length	Examples of Allowed Literal Values
float	32 bits	99F -327,456,99.01F 4.2FE6 (engineering notation for 4.2 * 10 ⁶)
double	64 bits	-1111 2.1E12 999,701,327,456,99.999

Floating Point Primitive Types

There are two types for floating point numbers, `float` and `double`. They are used to store numbers with values to the right of the decimal point. As with the different integer types, these share functionality but differ in size.

It is not possible to state the largest or smallest value that each of these can hold because they allow variable accuracy depending on the magnitude of the number. (That is, the number can hold X number of digits, but the placement of the decimal point determines whether it is an extremely large number, or whether it is a small number like pi.)

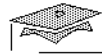
The largest values cannot be specified; however, the largest possible number of digits can be.

Table 3-2 shows information about the two floating point types.

Table 3-2 Floating Point Types

Type	Float Length	Examples of Allowed Literal Values
float	32 bits	99F -327,456,99.01F 4.2FE6 (engineering notation for $4.2 * 10^6$)
double	64 bits	-1111 2.1E12 999,701,327,456,99.999

When you specify a literal value for a `float`, put a capital F (float) to the right of the value to explicitly state that it is a `float`, not a `double`. Literal values for floating point types are assumed to be `doubles` unless you specify otherwise, using the F.



Sun Educational Services

Textual Primitive Type: `char`

- `char` stores any single character.
- Use single quotes around any literal value that you assign to a `char` variable: `'t'`
- `char` represents each character as a series of 16 bits.
- `char` is based on Unicode: 65,535 characters (contains first 128 characters of ASCII).

Textual Primitive Type: `char`

Another data type you need to store and manipulate is single-character information. The primitive type used for storing a single character is `char`.

Note – You can store only one character in a `char` variable. If you want to store whole words or phrases you use an object type (not a primitive type) called `String`. This will be discussed later.

When you assign a literal value to a `char` variable, such as `t`, you need to use single quotation marks around the character: `'t'`

This clarifies for the compiler that the `t` is just the literal value `t`, rather than a variable `t` that represents another value.

The `char` does not store the actual character you type, such as the `t` shown. The `char` representation is reduced to a series of bits that corresponds to a character. The number-character mappings are set up in the character set the programming language uses.

Most computer languages use the American Standard Code for Information Interchange (ASCII), an 8-bit character set that has an entry for every English character and punctuation mark, numbers, and so on.

The Java programming language uses a 16-bit character set called Unicode that can store all the necessary displayable characters from the vast majority of languages used in the modern world. Your programs can therefore be written so they will work correctly and display the correct language for most countries.

Unicode contains a subset of ASCII (the first 128 characters).



Sun Educational Services

Logical Primitive Type: `boolean`

- Represent a value of true or false
- Examples:
 - ▼ Assign a literal value to a `boolean` variable (Java technology keywords `true` or `false`)
 - ▼ Boolean expression: `answer > 42`

Logical Primitive Type: `boolean`

One activity that computer programs do frequently is to make decisions. The result of a decision—whether the statement in the program is true or false—is saved in `boolean` variables. `boolean` variables can store only the results of expressions or Java programming language keywords:

- The Java programming language keywords `true` or `false`
- An expression that may only evaluate to true or false, like `answer < 42` (the variable `answer` is less than 42)

The following are both read as true by the compiler: `true` and `4 > 3`)

Variable Identifier Conventions and Rules

This section tells you more about another variable characteristic, the name. In Java technology, the name is also referred to as an *identifier*.

You now know two attributes of a variable: the name and the type. The rectangle in Figure 3-4 represents a variable.

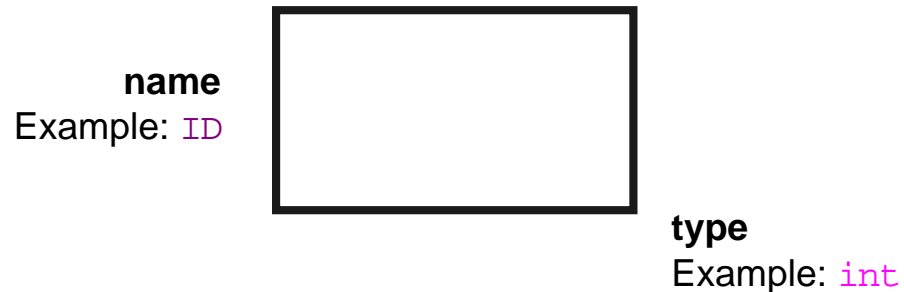
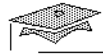


Figure 3-4 Variable Name and Type

Identifiers are the names you assign to classes, variables, methods, and so on. Every class has an identifier, as does each method and variable.



Picking a Variable Identifier

- Identifier should be simple but descriptive.
- An item ID stored as an integer could have the identifier ID, myID, itemID, itemNumber, and so on.
- Common convention is use of one-letter identifier:
 - ▼ Quick
 - ▼ Not recommended for long applications

Picking a Variable Identifier

The purpose of the variable, of course, is so that you can use it to refer to a value, and so that you and others reading your code can figure out what you are referring to. Therefore, it is a good idea to make the identifiers simple but descriptive.

If you store the value of an item ID, you can use the name ID to refer to the value 488990 or whatever the item's ID is. You also can name the variable myInt or myID, itemID, itemNumber, or anything else that makes it clear that this is the identifier for a value that stores an item ID.

Many programmers use the convention of designating the first letter of the type as the identifier: `int i`, `float f`, and so on. This is all right for small programs that are easy to decipher, but generally you should use more descriptive identifiers.



Variable Identifier Naming Rules

- First character can be A-Z, a-z, _ or \$
- Subsequent characters can be any of the above and numeric
- `order` is different than `Order`
- Cannot be a Java technology keyword

Variable Identifier Naming Rules

The following rules dictate the content and structure of identifiers:

- The first character of an identifier should be a lowercase letter. However, it can be any of the following:
 - ▼ A letter (A-Z or a-z)
 - ▼ The underscore or dollar character (_ or \$)
- The second and subsequent characters of an identifier must be any of the following:
 - ▼ Any character from the previous list
 - ▼ Numeric characters (0-9)
- Good examples are `isFull` (for a boolean value), `price` (for a float), `midInitial` (for a char), and so on. Code examples in other books or on the Internet often use names like `myInt` and `myDouble`.

- Java is a *case-sensitive* programming language. Case sensitivity means distinguishing between the upper- and lowercase representations of each alphabetical character. The Java programming language considers two characters in your code to be different if only their capitalization differs. A variable called `order` is different from `Order`.
- You must not use a Java technology keyword (shown in Table 3-3) as an identifier.

Table 3-3 Java Technology Keywords

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



Sun Educational Services

Variable Identifier Naming Conventions

- Example: `isFull` (for a boolean value), `price`, `itemID`, `myInt`
- Start with a lowercase letter, no separating character between words
- Start second and subsequent words with uppercase letter

Variable Identifier Naming Conventions

Follow these conventions to make your code more readable, both for yourself and others.

- Begin each variable with a lowercase letter.
- If you use two or more words in the identifier, begin each subsequent word with an uppercase letter. Do not separate words with spaces, underscores, dashes, or other characters (for example, name a variable `thisIsMyVariable`).

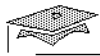
Exercise 2: Selecting Primitive Types

Exercise objective – Learn how to assign primitive types to attributes.



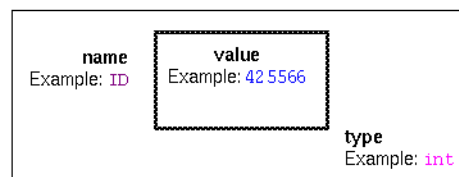
Tasks

Refer to the system that you modeled in the exercise for Module 2, “Object-Oriented Analysis and Design.” Select the appropriate primitive data type for each attribute.



Creating Primitive Type Variables

- With this section, you now know how to specify the type, value, and name.
- You need to:
 1. Declare the variable (give its type and name).
 2. Assign a value to the variable.



Creating Primitive Type Variables

This section tells you more about the name attribute of a variable: how to specify what a variable's name is. In addition, it describes how to specify the third piece of information each variable has; its value. Each primitive type variable or constant has several different pieces of information. You now know how to specify the type, value, and name.

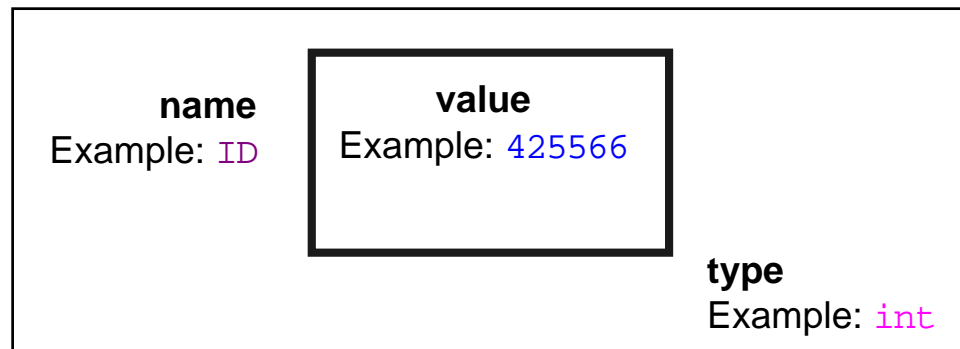
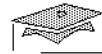


Figure 3-5 Variable Name, Value, and Type

You need to complete two steps:

1. Declare the variable (state its type and name).
2. Set the variable value.



Sun Educational Services

Creating Primitive Type Variables

- Step 1 – Declaring

```
type identifier [, identifier];
int ID;
```

- Step 2 – Assigning

```
identifier = value;
ID = 428890;
```

The equals operator (=) assigns values.

Declaring and assigning

```
type identifier = value [, identifier = value];
int ID = 428890;
```

Step 1 – Declaring a Variable

Declare a variable by assigning a type to a variable name. Declaring variables is done within either a class or method code block.

Syntax:

```
type identifier [, identifier];
```

Note – Square brackets ([]) indicate that the enclosed item is optional, and should not be used within the syntax.

You can declare several variables on the same line, but only if they are all of the same type.

Examples:

```
int ID;
float price, wholesalePrice;
char myChar;
boolean isOpen;
```


Step 2 – Assigning a Value

The variables then need to be assigned values.

Syntax:

```
identifier = value;
```

Examples:

```
ID = 428890;  
price = 2.35F;  
myChar = 't';  
isOpen = false;
```

Note – The = operator assigns the value on the right side to the item on the left side. The = operator should be read as *is assigned to*. In this example, “428890 *is assigned to* ID.”

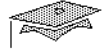
You can declare several variables on the same line, but only if they are all of the same type.

Syntax:

```
type identifier = value [, identifier = value];
```

Examples:

```
int ID = 428890;  
float price = 2.35F, wholesalePrice = 1.95F ;  
char myChar = 't';  
boolean isOpen = false;
```



Sun Educational Services

Kinds of Values You Can Assign

- Literal values:

```
float price = 2.25F;
```

- Other variables:

```
float casePrice = 189.99F;
float price = casePrice;
```

- The results of expressions:

```
float numberOrdered = 908.5F;
float casePrice = 19.99F;
float price = (casePrice * numberOrdered);
```

- Method calls (covered later)

Kinds of Values You Can Assign

You can assign values to variables using several different approaches.

- Literal values like the ones seen so far

```
int ID = 428890;
float price = 2.25F;
char myChar = 't';
boolean isOpen = false;
```

- Other variables' values to all primitive data types

```
int first = 9;
int second = first;
```

These lines create an integer called `first` and use it to store the number 9, then create another integer called `second` and use it to store a copy of `first`. If the contents of `first` are later changed this does *not* automatically change `second`. The two have the same value right now, but will be changed independently unless you specifically assign them the same value again.

Examples include the following:

```
int ID = 428890;
int saleID = 7728890;
saleID = ID;

float casePrice = 189.99F;
float price = casePrice;

char custID = 'P'
char myChar = custID;

boolean isWorkWeek = true
boolean isOpen = isWorkWeek;
```

- The results of expressions to integral, floating point, and boolean variables

For all of the following lines of code, the result of everything on the right side of the = operator is assigned to the variable on the left side of the = operator.

```
float numberOrdered = 908.5F;
float casePrice = 19.99F;
float price = (casePrice * numberOrdered);

int hour = 12;
boolean isOpen = (hour > 8);
```

- Method calls (you will learn more about this later)



Using Variables in a Program

- Use:

```
System.out.println (ID);
```

- Re-use (assign another value later):

```
ID = 428890;
```

- Combine:

```
int ID = 428890;  
int saleID;  
saleID = ID;
```

Using Variables in a Program

You will learn more about this later; however, this provides a brief example.

To use it again, such as for assigning a different value, the process is the same as assigning the value the first time:

```
ID = 428890;
```

You can use two or more variables as shown in the following examples:

```
int ID = 428890;  
int saleID;  
saleID = ID;
```

For most purposes, simply include the variable where it is needed.

```
System.out.println (ID);
```



Constants

- For values that cannot change once assigned:
`final double SALES_TAX = 6.25;`
- Cannot be changed except in original location.
- Use `final` keyword to make unchangeable.
- Use all capital letters and underscores for identifier.
- The compiler will give an error message if an attempt is made to change the constant's value.

Constants

So far, the data referred to in this module has been variables—you can create a variable, then assign a value to it, then change the value later. You also can use constants to represent a value that cannot change.

Assume you are writing part of the clothing catalog program, and need to refer to a sales tax rate. You could create the following variable:

```
double salesTax = 6.25;
```

However, this value could easily be changed by other programmers. In addition, the sales tax does not change all that frequently. The value might be better stored in a location that cannot change.

To inform the compiler of what you want, make the sales tax storage location a constant:

```
final double SALES_TAX = 6.25;
```

Anyone else on the project needing to refer to it would just use `SALES_TAX` and not need to know the rate, or be concerned with accidentally changing it.

The compiler would give an error if someone attempted to change the value in a location other than where it was originally defined.

If the next year the sales tax increased, you would change the constant in the one location where it is defined. All uses of the constant would reflect the change automatically.

Constants are also convenient for values that are extremely long, such as pi (3.14159... and so on).

Define constants with the keyword `final`. The impact of `final` in the declaration makes the constants unchangeable. Follow the recommended naming conventions: Name the constant identifier using all capital letters, with underscores separating words.

How Primitives and Constants Are Stored in Memory

When you create a variable, constant, or literal, and assign it a value, the value is stored in the computer's memory.

Take the example of the following integral variable:

```
int ID = 425566;
```

The storage location in memory of the identifier `age` contains the value 425566. The size of the storage location in memory is determined by the size of the data type; in this case, `int` (32 bits).

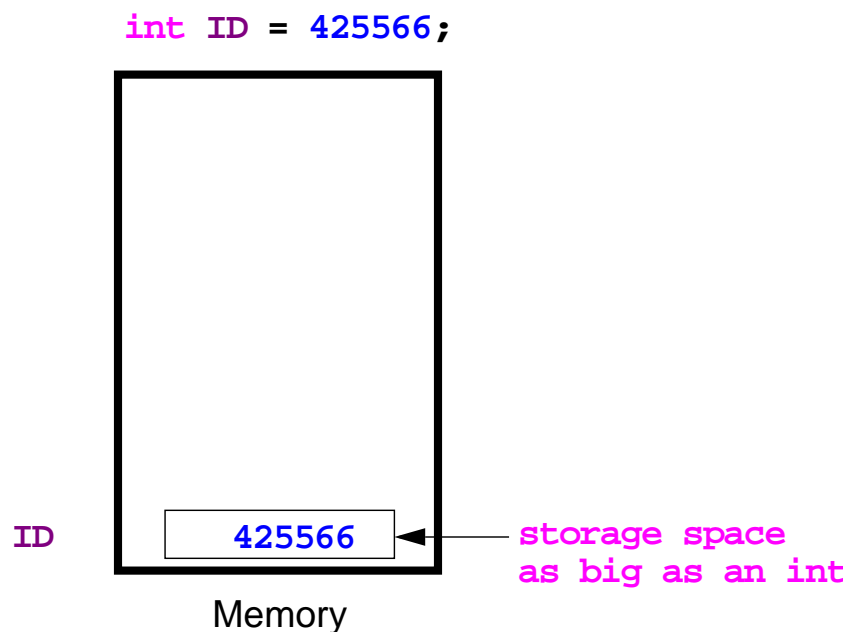


Figure 3-6 Storing Primitive Data Type Variables in Memory

Exercise 3: Using Primitive Type Variables in a Program



Exercise objective – Write the code to declare, assign values to, and use variables in a program.

Tasks

Refer to “Creating Primitive Type Variables” on page 3-33 in your student guide for information on using variables.

Open a text editor and create a file in your work folder called `Customer.java`. In that file, write a program that creates and assigns values to the following variables (use constants if appropriate):

- Customer ID
- Social security number (without dashes)
- Customer status (stored as one letter)
- Member of Frequent Buyers club?
- Total purchases for the year

Name the class `Customer`, and use the main method, as shown:

```
public class Customer
{
    public static void main (String args[])
    {
        // declare and assign values to the variables and/or constants here
        // then print them to the screen using System.out.println
    }
}
```

Use `System.out.println` to print each value to the screen with a corresponding label (such as `"Purchases are: "`).

Add another line that prints the average purchases for each month, with the label `"Average purchases each month are: "`

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Describe how computers store data using the binary system
- ☐ List the eight Java technology primitive types
- ☐ State the storage values for the primitive types
- ☐ Create identifiers according to Java technology rules and coding standards
- ☐ Write programs that follow Java technology rules and recommended coding standards
- ☐ Write code to declare and assign literal values, expressions, and variables
- ☐ Write code to declare and assign data to a constant
- ☐ Describe how variables and constants are stored in memory

Think Beyond

You designed objects like `Shirts` in Module 2, “Object-Oriented Analysis and Design.” What would you need to do in a Java program to use them?

Objectives

Upon completion of this module, you should be able to:

- Write code to create and use objects in a program
- Write code to create `Strings` to store a series of characters in a variable
- Describe how objects and `Strings` are stored in memory
- Use the `main` method
- Design and write code for a class
- Add comments to programs
- Write a basic Java program

This module continues your basic Java technology programming knowledge, begun in Module 3, “Getting Started With Java Technology Programming.” It describes how to use objects in your programs, and tells you how to pull your programs together with specific method and class syntax.

Relevance



Discussion – What do objects do in programs?

Object Overview

You learned in Module 2, “Object-Oriented Analysis and Design,” that objects are instances of classes.

In this section, you will learn how to create objects using the Java programming language so that you can get objects to do work in your programs. The following example shows a *Shirt* class, *Shirt* objects, and how to represent the *Shirt* in code.

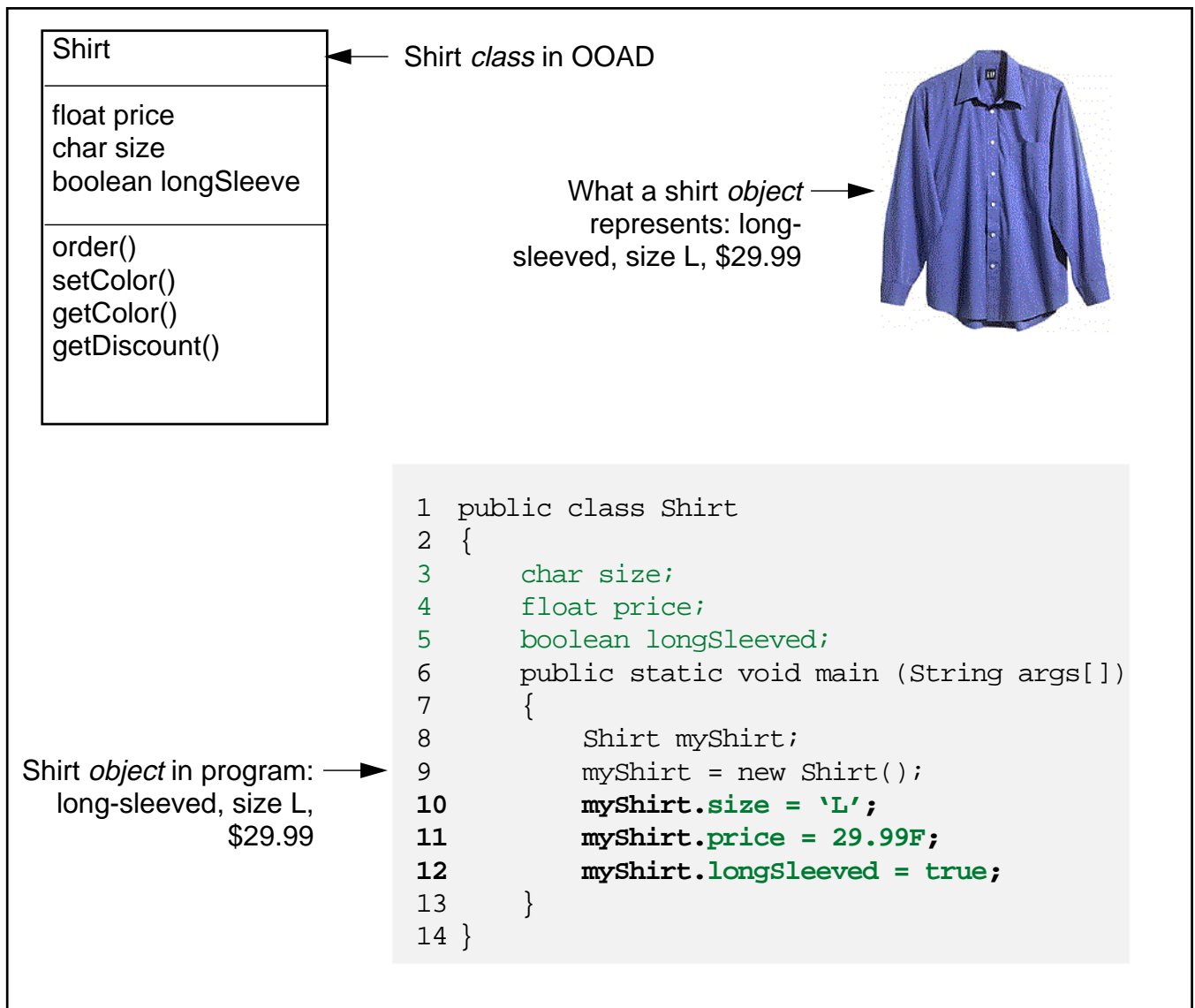


Figure 4-1 Class, Object in Real Life, and Object in a Java Program

Notes

Note – The Shirt example is used throughout this course; for simplicity, all possible variables are not shown in all examples, and different variables are shown in some examples. Variables of a certain data type are introduced later in order to demonstrate how they could be used.



Creating Object Reference Variables

- Create a variable that is a reference to an object.
- There are three steps:
 1. Declare a reference to the object.
 2. Create the object.
 3. Assign values.

Creating Object Reference Variables

As with declaring and initializing primitive type variables, there are two steps. You need to declare the object reference—what you want to call this particular object—and then create, or *initialize*, the object.

The name you use to refer to the object is called an *object reference*, or a *reference variable*.

For primitive data types, you needed two steps. For objects, you need to complete three steps:

1. Declare a reference to the object (specify its identifier and class).
2. Create, or initialize, the object using the `new` modifier.
3. Assign values to the object variables.



Sun Educational Services

Creating Objects

- Step 1 – Declaration

```
ClassName identifier;  
Shirt myShirt;
```

Variable type is *reference_to_ClassName*
(reference to *Shirt*)

- Step 2 – Initialization

```
identifier = new ClassName();  
myShirt = new Shirt();
```

Declaration and initialization together

```
ClassName identifier = new ClassName();  
Shirt myShirt = new Shirt();
```

Step 1 – Declaration

State the class that you want to create an object from, then the name you want to use to refer to the object.

Syntax:

```
ClassName identifier;
```

Example:

```
Shirt myShirt;
```

You now have an object reference `myShirt` to a `Shirt` object. You can name the reference `shirt1`, `bob'sShirt`, `aShirt`, or any other identifier, as long as it is clear that it is a reference to an object of the `Shirt` class.

Note – When you declare primitive type variables, the syntax is:

```
type identifier;
```

The principle is the same in this case; here, the *ClassName* defines the type. The type is “reference to *ClassName*.” A *Shirt* object variable’s type is “reference to *Shirt*,” a *Customer* object variable’s type is “reference to *Customer*.”

Step 2 – Initialization

After you declare the object reference, create, or *initialize*, the object.

Syntax:

Use the *new* modifier:

```
identifier = new ClassName();
```

Example:

```
myShirt = new Shirt();
```

The variable *myShirt* now is a reference to a new instance of the *Shirt* class.

You can declare and instantiate on one line. This is simply a shorter alternative to the two separate steps already shown.

Syntax:

```
ClassName identifier = new ClassName();
```

Example:

```
Shirt myShirt = new Shirt();
```



Creating Objects

Step 3 – Assigning Values

- You have not yet assigned object variables values like price.
- Variables are initialized with default values.
- To assign values:
 - ▼ Declare and initialize object variables
 - ▼ Specify object reference when setting value

Step 3 – Assigning Values to the Object Variables

Using the code in the previous examples, you can create objects but not set specific values like L, \$19.99, or “Polo shirt.”

The variables do have values, even though you have not yet assigned them. If you do not specify any initialization information when you create an object, all variables are initialized to their *default values*. Default values are 0 for integral values, 0.00 for floating point variables, false for boolean, and ‘\u0000’ for char.

If you have values that you want to assign to the object, you can do so in a number of ways. For now, simply declare and initialize the objects as shown in the previous two sections, then declare variables for each of the object’s attributes and assign values, the same way you would assign values to any variable.

In addition, put the name of the object and a period in front of the variable name when you assign the value, so the compiler knows which object should have the value.

```
1 public class Shirt2
2 {
3     char size;
4     float price;
5     boolean longSleeved;
6
7     public static void main (String args[])
8
9     {
10         Shirt2 myShirt;
11         myShirt = new Shirt2();
12         myShirt.size = 'L';
13         myShirt.price = 29.99F;
14         myShirt.longSleeved = true;
15
16         Shirt2 anotherShirt;
17         anotherShirt = new Shirt2();
18         anotherShirt.size = 'M';
19         anotherShirt.price = 22.99F;
20         anotherShirt.longSleeved = false;
21     }
22 }
```

This example now has two references to two objects, with values for their attributes.

Note – For now, when you use `main` and create objects in a program, you need to declare the variables (lines 3 through 5) as shown in the example inside the class code block, not within the `main` method code block.

Reference Variable Information

Like primitive types, every reference type variable has the same pieces of information, though the value actually stored in the variable is somewhat different:

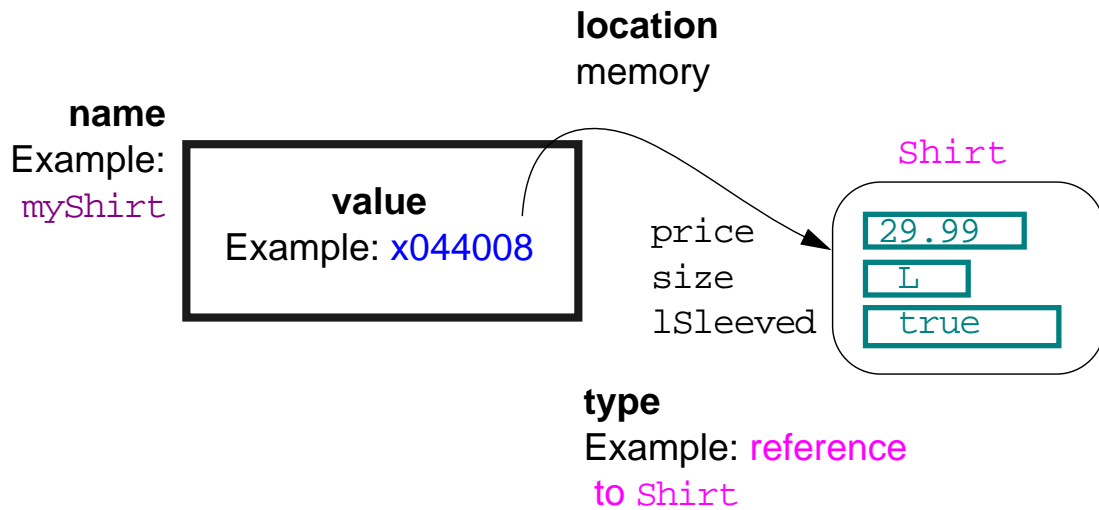


Figure 4-2 Pieces of Information Each Reference Variable Has

You will learn more about the values for reference variables in the next section, “How Reference Variables Are Stored in Memory.”

How Reference Variables Are Stored in Memory

Primitive Data Type Storage

You have already learned how primitive data type variables store data, as shown in Figure 4-3.

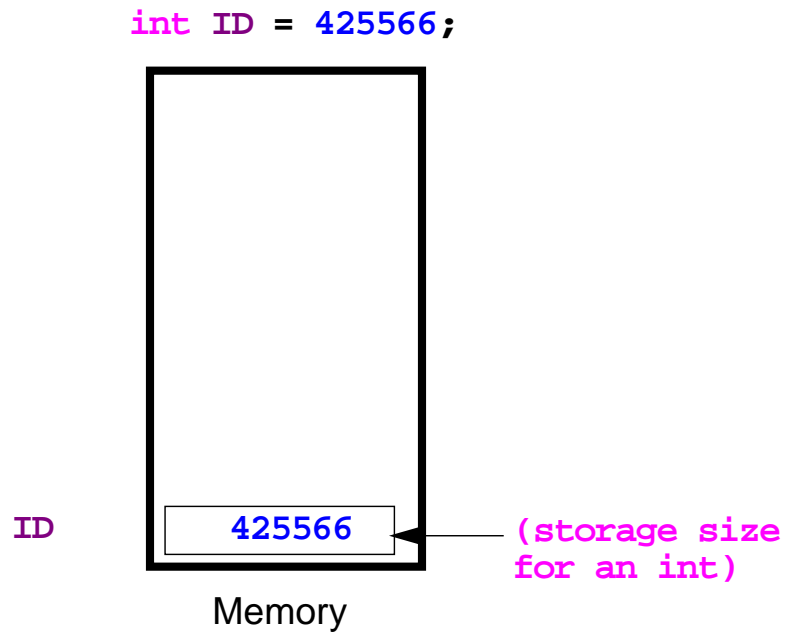


Figure 4-3 How Primitive Data Types Are Stored in Memory



Values of Primitive Variables and Object Reference Variables

- Value of a primitive data type: 428802, "Nigel Tufnel", and so on
- Value of an object reference: *reference* to the object, points to object variable's values

Values of Primitive Variables and Object Reference Variables

A primitive data type variable is its value; when you refer to `inStock`, the compiler reads 42.

However, an object rarely has just one value, like a primitive. A `Shirt` object would have many values to store: its price, style, ID, and so on.

Therefore, the value of an object reference variable stores a reference to all those values (sometimes referred to as a *handle*). This reference is the location of where the object and information about it is stored.

It can be compared to a street address. Just as you can find a friend by knowing that his or her address is 901 Harrison Drive in Fargo, North Dakota, USA, the compiler can find an object using a reference, such as 0x334009, to a specific location in memory.

This is illustrated in Figure 4-4 on page 4-13.

Variable Storage in Memory

Figure 4-4 shows how the variable types are stored differently. The values at initialization are shown (0 rather than an actual ID, for example).

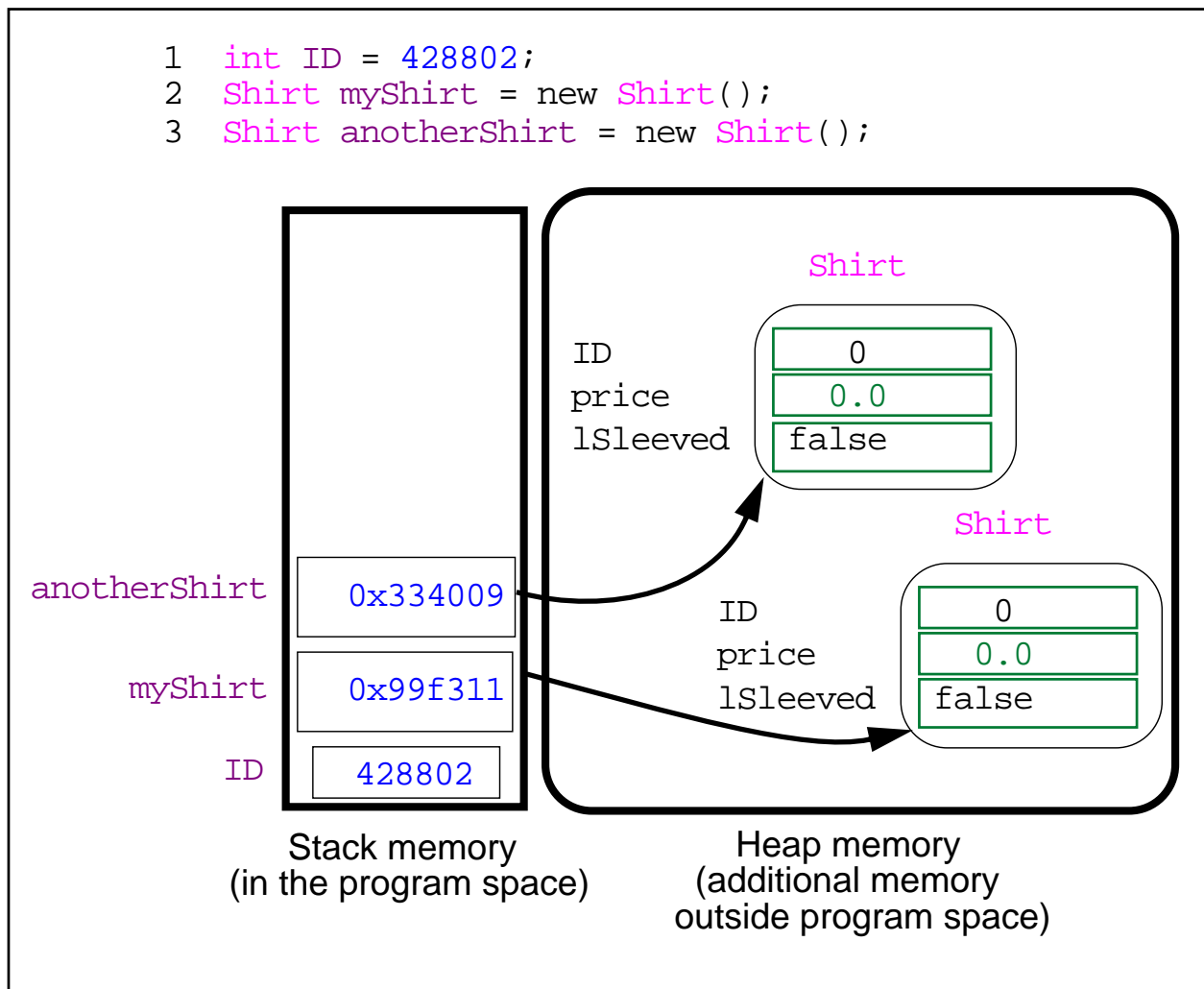


Figure 4-4 How Primitive Variables and Reference Variables Are Stored in Memory

- The *types* of the variables are `int` for the `ID` and reference to `Shirt` for the objects.
- The *values* of the variables are 428802, 0x99f311, and x334009.
- The actual objects referred to by `myShirt` and `anotherShirt` are stored at the locations in heap memory designated by the references 0x99f311 and 0x334009.

Assigning One Object Reference to Another

As shown in Figure 4-4, you created two object references to `Shirt`. This section explains what happens when you assign the first one to the second one.

```
Shirt myShirt = new Shirt();
Shirt anotherShirt = new Shirt();
anotherShirt = myShirt;
```

As with primitive reference variables, the value of the item on the right is assigned to the item on the left. In this case, the value of the reference variable `myShirt` is the reference, say `0x99f311`. Now `anotherShirt`'s value is also `0x99f311`. Both variables point to the same object, even though the other object, the one that `anotherShirt` used to point to with its reference `0x334009` still exists. Figure 4-5 illustrates this.

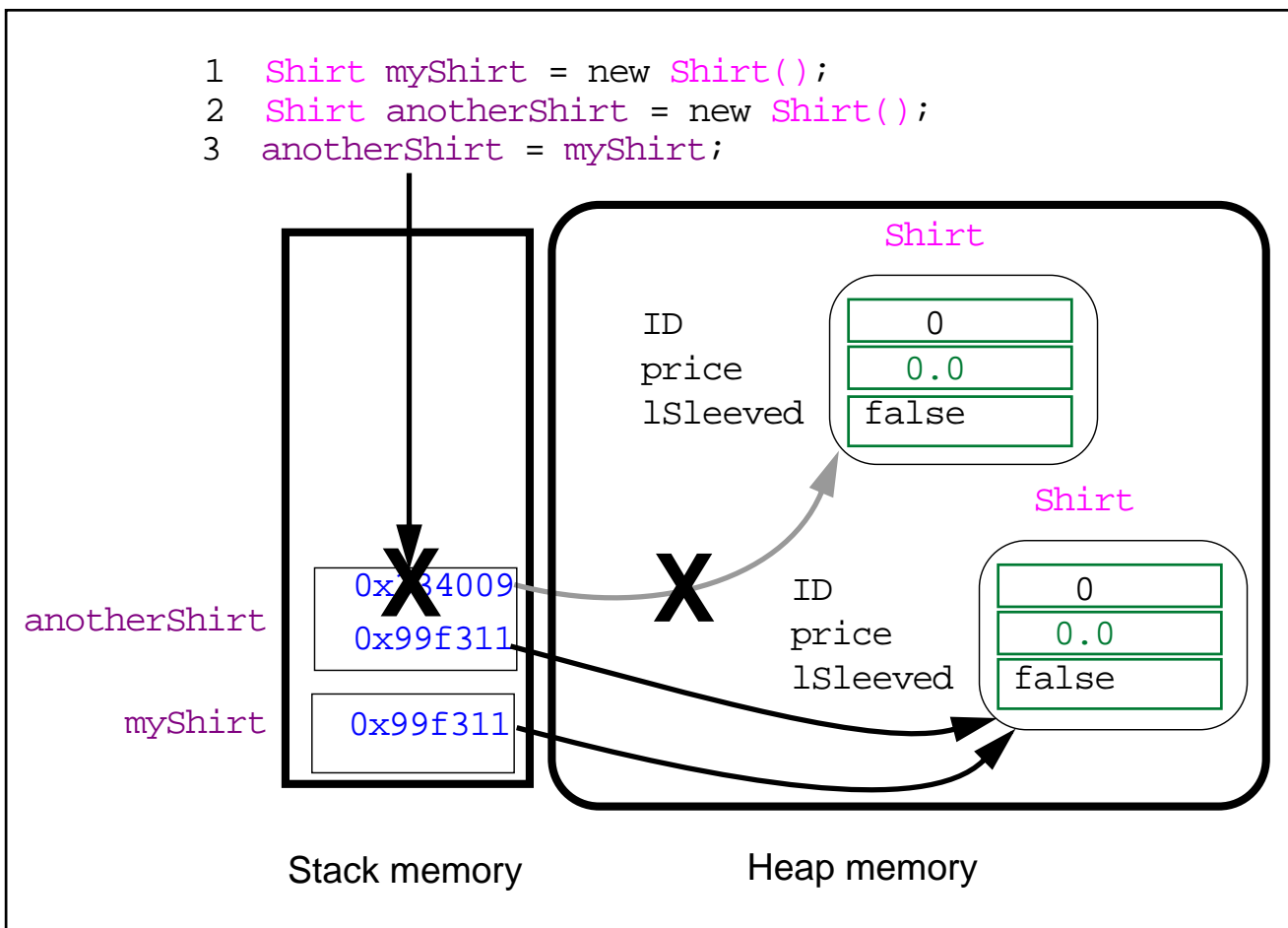


Figure 4-5 Assigning One Reference Variable to Another

Line 2 in Figure 4-5 creates a reference called `anotherShirt` and assigns a reference to a new `Shirt` object in it.

Line 3 assigns `myShirt` to `anotherShirt`. It does not duplicate the `myShirt` object; it copies the value of `myShirt` over the value of `anotherShirt`, replacing it. There are now two references to one `Shirt`, and both point to the same object at the same location.

The object that `anotherShirt` used to point to will eventually be removed. For now it still exists, though you can no longer access it.



Using the String Class as a Data Type

- To store a series of non-numeric characters, use `String` class.
- `String` is included in Java API.
- You can make `Strings` two ways: with or without the `new` modifier.

Using the String Class as a Data Type

Objects serve another purpose; the ability to store more than one character in a non-numeric variable is the responsibility of the `String` class. This class is part of the Java API, in `java.lang`.

For extra flexibility, you can make `String` variables with the `new` modifier, as already shown with other objects, or without the `new` modifier, as already shown with primitives. Both ways are shown on the following pages.



Using String and the new Modifier

- Declaration

```
ClassName identifier;  
String myName;
```

- Initialization and assigning value

```
identifier = new ClassName ("string_value");  
myName = new String("David St. Hubbins");
```

To do both steps on same line:

```
String myName = new String("David St. Hubbins");
```

Using String and the new Modifier

You now know how to create objects from classes, using the new modifier. For a String this would look like the following examples:

Declaration

Syntax:

```
ClassName identifier;
```

Example:

```
String myName;
```

Initialization and Assigning Value

Syntax:

```
identifier = new ClassName ("string_value");
```

Example:

```
myName = new String("David St. Hubbins");// Use double  
// quotation marks around  
// String values.
```

You also can do both steps on the same line.

```
String myName = new String("David St. Hubbins");
```



Using String Without the new Modifier

- Declaration

```
ClassName identifier;  
String myName;
```

- Initialization and assigning value

```
identifier = string_value;  
myName = "David St. Hubbins";
```

To do both steps on the same line:

```
ClassName identifier = value;  
String myName = "David St. Hubbins";
```

Using String Without the new Modifier

This method is shorter and more convenient, and uses the same approach you used for primitive data types. Strings are unique because they are the only class you can build objects from without using the new modifier.

It is not necessary to explicitly state the class name, `String`, because it is implicitly stated when you use the double quotation marks around the String value.

The value you put in quotation marks is not the value of *identifier*; it is a value of the String object.

Declaration

Syntax:

```
ClassName identifier;
```

Example:

```
String myName;
```

Initialization and Assigning Value

Syntax:

```
identifier = string_value;
```

Example:

```
myName = "David St. Hubbins";// Always use double  
// quotation marks around  
// String values.
```

You also can declare and assign values simultaneously.

```
ClassName identifier = value;  
String myName = "David St. Hubbins";
```

Summary

Both ways of creating Strings are the same; both of the following are read the same way by the compiler:

```
String myName = "David St. Hubbins";
```

```
String myName = new String("David St. Hubbins");
```



Sun Educational Services

Values You Can Assign to Strings

Similar to primitive types:

- Literals:

```
String name = "Nigel Tufnel";
```

- Expressions:

```
String name = "Nigel" + " Tufnel";
```

- Other variables:

```
String hisName = "Nigel Tufnel";  
String myName = hisName;
```

- Method calls

Values You Can Assign to Strings

You can assign the same four types of values as the values listed in “Kinds of Values You Can Assign” on page 3-36:

- Literals

```
String name = "Nigel Tufnel";
```

- Expressions

```
String name = "Nigel" + " Tufnel";
```

- Other variables

```
String hisName = "Nigel Tufnel";  
String myName = hisName;
```

- Method calls

How Strings Are Stored in Memory

Memory storage for String reference variables works the same way as for other object references: You have a variable with a name, such as myName, containing a reference in memory where a String object is located. The String object has one value, the string of characters you gave it. Figure 4-6 illustrates this.

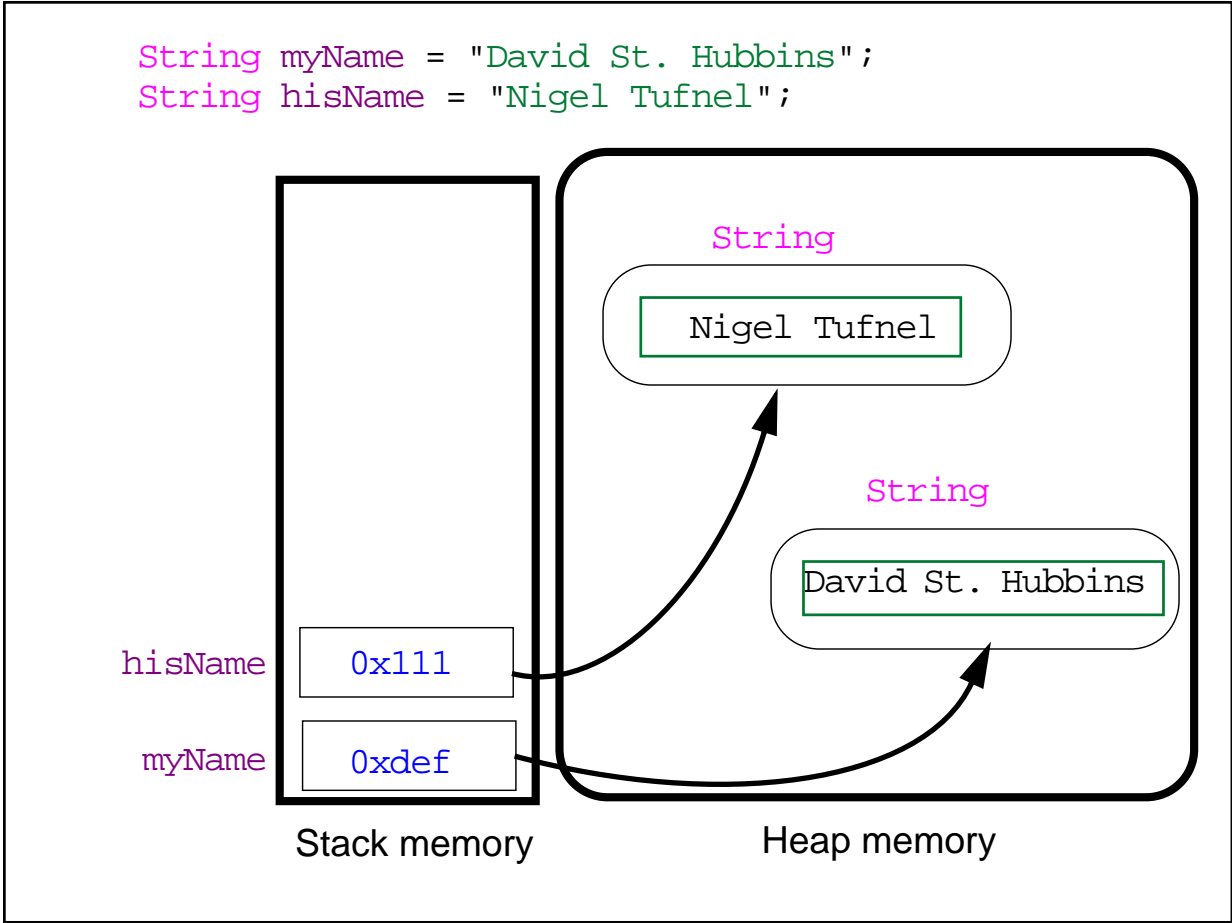


Figure 4-6 How Strings Are Stored in Memory

Using String Reference Variables

Use String variables in programs as you would primitive variables. The following example shows a program that simply states a person's name and job.

```
1 public class Introductions1
2 {
3     public static void main (String args[])
4     {
5         String name;
6         String job;
7         name = "Derek Smalls";
8         job = "drummer";
9
10        System.out.println("My name is " + name + " and I am a " + job);
11    }
12 }
```

Exercise 1: Using Objects and Strings in Programs



Exercise objective – Create objects and assign values to their variables, and use the `String` class as a variable data type.

Tasks

Follow the directions from your instructor to locate the exercises directory for this module.

1. Write a program called `Quotation2` that creates two variables, one storing a quotation and one storing the name of the person who said it. Set the values using the syntax on “Using String and the new Modifier” on page 4-17 or “Using String Without the new Modifier” on page 4-19 in your student guide. Print the quotation and the name to the screen.
2. Write a program called `Quotation3` that:
 - ▼ Defines a class called `Quotation3` with three variables: the quotation, the person who said it, and the year when it was first said
 - ▼ Creates two quotation objects
 - ▼ Assigns different values to the attributes, for each object
 - ▼ Prints the objects’ variable values to the screen



Sun Educational Services

Using the `main` Method

- Methods contain the work that the program does.
- All classes can use the "all-purpose" `main` method.
- The `main` method is the entry point for the JVM.
- `main` invokes and coordinates the rest of the program.

Using the `main` Method

The `main` method is the entry point for the JVM—`main` lets the JVM know where to start in an application. Every class automatically has a `main` method that the programmer can write in the program.

Methods contain the work that the program does. For example, methods can create and assign values to variables, complete mathematical procedures, make decisions, print to the screen, call other methods, and much more.

The `main` "all-purpose" method has been shown in several examples so far, including the one on page 3-6. Technically, you do not need to create any other methods to make your program do what you want. `main` is responsible for invoking other methods and coordinating the flow of the rest of the program.

You will use other methods, specific to the objects in a program, later in this course. For now, it is simpler to use `main` as the structure within which the program's actions take place.

main *Method Syntax*

The syntax of all methods is as follows. Methods are written within the code block of the class that they belong to.

Square brackets ([]) indicate an item is optional.

```
[modifiers] return_type method_identifier ([arguments])  
{  
    method_code_block  
}
```

For the main method, the syntax corresponds as follows:

- [*modifiers*] – Several Java technology keywords can be used to modify the way methods are stored or how they run. `public` and `static` are required modifiers for the main method.
- *return_type* – Some methods return a value. Methods can return only one value; if no value is returned, the keyword `void` is the return type. This is the case with the main method.
- *method_identifier* – The identifier is used to call the method. The identifier for the main method is `main`.

Note – Method identifiers should follow the rules and guidelines for variable identifiers.

- ([*arguments*]) – Arguments let you pass values into a method so that the task can work on different data each time you use it. (Arguments are optional; many methods do not accept arguments.)

The argument for main is `String args[]`.

- *method_code_block* – The code block of a method is the brace-enclosed sequence of statements that perform a task or tasks. In examples shown previously in this course, the task was to print to the screen. However, a wide variety of tasks, variable declarations, and so on can be in the method body.

main Examples

The following code is a basic example, using main.

```

1 public class CustomerInfo
2     public static void main (String args[])
3     {
4         Customer customer1 = new Customer();
5         customer1.ID = 423356;
6         System.out.println("Customer ID is: " + customer1.ID);
7     }
8 }

1 public class Customer
2 {
3     int ID;
4     String name;
5 }
```

This example uses System.out.println and main.

```

1 public class Train
2 {
3     public static void main(String args[])
4     {
5         System.out.println("          * * * * *");
6         System.out.println("          *");
7         System.out.println("          ");
8         System.out.println("  _I_I_oo_ii_|_|_|_|\\");
9         System.out.println(" | | - - | I | | U P |");
10        System.out.println(" _|_|_|_|_|_|_|_|_|_|");
11        System.out.println(" I=| o (_)--(_)--(_)--(_)-- O--O  O-O +++++ O-O");
12    }
13 }
```



Pre-Written Code to Use in Your Programs

Simple ways to:

- Print information to the screen
- Use command-line input in a program
- Convert `Strings` to `ints`

Pre-Written Code to Use in Your Programs

Now that you know how to use the `main` method, this section will add a few other techniques that will expand what you can do with it. They use principles that you will learn later in this course.

- `System.out.println` – Prints to the screen.
- Taking command-line input to use in `main` – Lets you type data in a terminal window that is used in the program.
- Converting `Strings` to `ints`, and `ints` to `Strings` – What you type on the command line is always considered a `String`; if you want to add numbers from the command line, for example, you need to convert them to `ints` first.



Using `System.out.println`

- `println` adds carriage return; `print` does not.

```
System.out.print("Carpe diem ");  
System.out.println("Seize the day");
```

Result: Carpe diem Seize the day

- Use empty print statement for a blank line.
- Printing multiple items within one statement:
 - ▼ Put quotation marks around text
 - ▼ To use spaces, put them inside quotation marks
 - ▼ Use `+` to connect items ("Price is: " + price)
 - ▼ Expression result can be printed (price * discount)

Using `System.out.println`

`System.out.println` is a pre-written method in the Java programming language that you have been using throughout this module. It prints text, such as "Welcome to Sun!" or variables, as well as other items not covered yet, to the terminal window. Not only is it useful to see what the program is doing, but you can use it when you are debugging, to verify what the program is doing at various points throughout your program.

Using it is reasonably simple; review the following for a few tips.

Including or Leaving Out Carriage Returns

The `ln` at the end of the method means "carriage return." If you want to stay on the same line, so that the next thing you print is on that line, then just leave out the `ln`.

This code:

```
System.out.println("Carpe diem ");  
System.out.println("Seize the day");
```

prints this:

```
Carpe diem  
Seize the day
```

This code:

```
System.out.print("Carpe diem ");  
System.out.println("Seize the day");
```

prints this:

```
Carpe diem Seize the day
```

Blank Lines

To print a blank line, just use an empty `println` statement.

This code:

```
System.out.println("Carpe diem ");  
System.out.println();  
System.out.println("Seize the day");
```

prints this:

```
Carpe diem  
  
Seize the day
```


Combining Text and Variables, or Multiple Variables

If you want to print plain text followed by a variable, such as "Price: " followed by the price variable, just use the + operator.

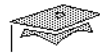
- Always surround any plain text in quotation marks.
- Spaces are seen by the compiler only when they are inside quotation marks; to put a space between Price: and the actual price, put the space inside the quotation marks.
- To add another variable or text, just add another plus sign.

Assuming that price and discount are already defined in a program; you could print the following:

```
System.out.println("Today's price," + price + ", will be reduced by " +  
discount + " starting Monday.");
```

- You can also print the result of an expression, like this:

```
System.out.println(price * discount);
```



Sun Educational Services

Using the main Method to Take Command-Line Input

- `(String args[])` lets you pass values to main from the command line.

```
1 public class Introductions
2 {
3     public static void main (String args[])
4     {
5         System.out.println("My name is " + args[0] + " and I am " +
6         args[1]);
7     }
8 }
```

- Command line: `java Introductions Floyd 29`
- Prints: `My name is Floyd and I am 29 years old.`

Using the main Method to Take Command-Line Input

The main method, in full, is:

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

`(String args[])` means that main can (but does not have to) take in and use an unlimited number of Strings as *arguments*. Arguments are values for the variables you use in your program; for instance, you could send the argument 423399 to the Shirt class as the value for the ID variable.

The person running the program provides the arguments to main by typing them on the command line after the command to run the program. It converts everything typed to a String: 34, for example, would be considered a String, not a numeric value.

This lets you write programs that are a little more true-to-life; you let the user (anyone running the program) specify the values instead of setting them within your program.

Using Command-Line Arguments in Your Programs

Refer to the values that you want the user to type by number, in the order you want them to be typed, starting with 0. `args[0]` is the first argument `args [1]` is the second argument, and so on.

```
1 public class Introductions2
2 {
3     public static void main (String args[])
4     {
5         System.out.println("My name is " + args[0] + " and I am " + args[1] +
" years old.");
6     }
7 }
```

In this program, you expect the person running it to type two arguments, a name and an age.

Sending Command-Line Arguments to the Program

To send arguments to the previous program, type something like this:

Syntax:

```
java classname valueforargs[0] valueforargs[1]...
```

Examples:

```
java Introductions Nigel 29
```

Typing that line means that the arguments Nigel and 29 are passed to the main method in the Introductions class as String arguments. 29 is not considered to be a number.

The program will print:

```
My name is Nigel and I am 29 years old.
```

An error message appears if you do not enter the right number of arguments.

Note – This is an unusual way of passing arguments; typically, another method provides the values for the arguments. Passing arguments in this way is covered later in this course.

The following code shows another example of using command-line arguments:

```
1 public class Recognition
2 {
3     public static void main(String args[])
4     {
5         System.out.println(args[0] + " has the highest sales for this
month.");
6         System.out.println(args[1] + " has the second-highest sales
for this month.");
7     }
8 }
```

To run it, you would type:

```
java Recognition name another_name
```

The program prints the following:

```
name has the highest sales for this month.
another_name has the second-highest sales for this month.
```



Converting String Arguments to Integers

- main considers all arguments Strings.
- To evaluate an argument as an int, use `Integer.parseInt`

```
Integer.parseInt(variable_to_convert)
```

```
int grade = Integer.parseInt(args[0]);
```

Converting String Arguments to Integers

The main method treats everything you type as a String; if you want to compare it to another number, or add it, you need to convert it to a number first.

Assume you typed "424452" as an argument. Even though 424452 is a number, this code would not work because an int, not a String, must be assigned to the int primitive variable ID:

```
int ID = (args[0]);
```

Use the `Integer.parseInt()` method to convert the argument to an int.

Syntax:

```
Integer.parseInt(variable_to_convert)
```

Example:

```
int ID = Integer.parseInt(args[0]);
```



Declaring Classes

- Write class declaration to express objects from OO design.
- Class declaration includes class name, variables, methods.

Declaring Classes

For each class you want to create and use in a program, you need to write a *class declaration*. This involves specifying the kind of class, the class name, the variables, and any methods.

You have been using class declarations throughout this module; this section provides more detailed information about the syntax.

When you declare classes, base them on the object-oriented analysis of the problem domain. You will need to declare a class for each object defined in the problem domain solution.

Class Declaration Syntax

To declare a class, use the following form. Optional elements are shown in square brackets ([]).

```
[modifier] class class_identifier
{
    class_code_block
}
```

- The *class_modifier* is optional (indicated by the square brackets) and may be `public`, `abstract`, or `final`. For now, just use `public`.
- `class` is a keyword telling the compiler that the code block is a class declaration.
- The *class_identifier* is the name of the class and must follow the same rules as variable identifiers (see “Variable Identifier Conventions and Rules” on page 3-27). However, class identifiers by convention should start with a capital letter like `Shirt` or `Order`.
- The *class_code_block* is for the variables and methods that make up the class. The braces { } around the *class_code_block* define where the class starts and ends.

Class Example

You can see the parts of the class syntax in the basic Java application from the beginning of this module.

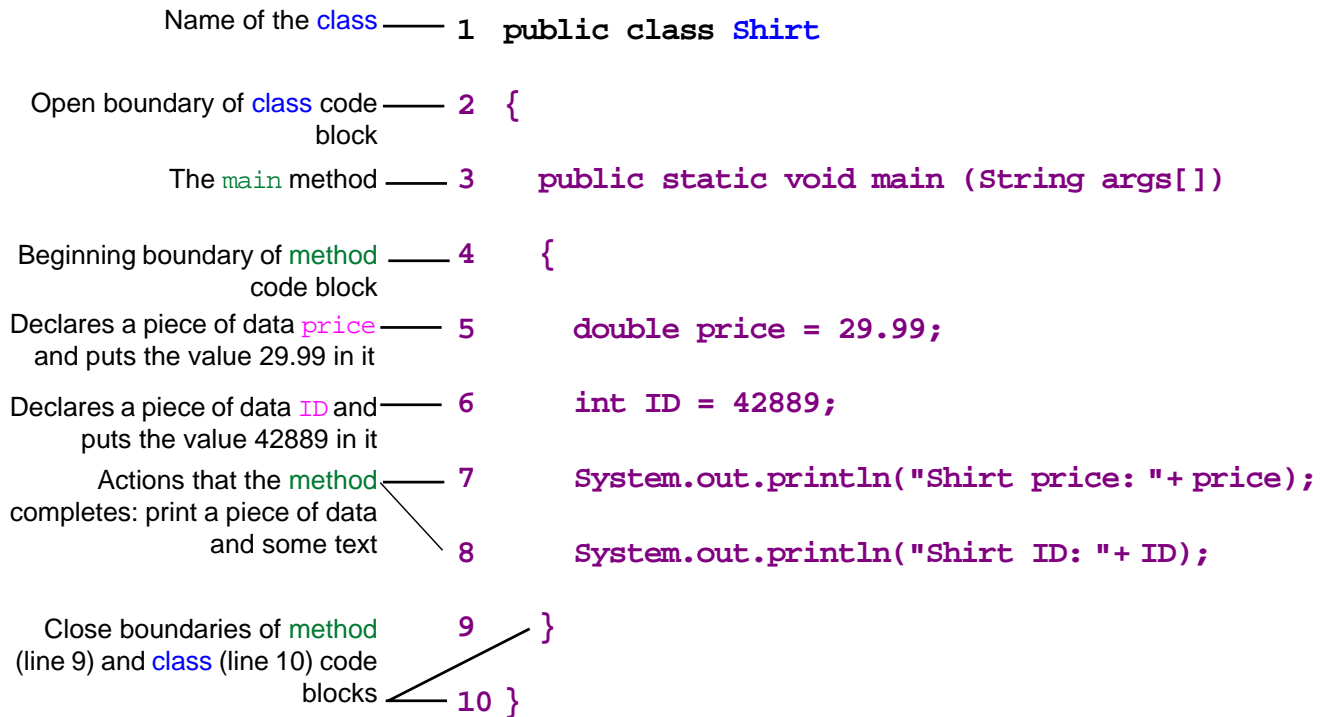


Figure 4-7 Basic Procedural Java Program

Exercise 2: Using Command-Line Arguments and String Conversion



Exercise objective – Write a program using the main method, `System.out.println`, String conversion, and command-line arguments.

Tasks

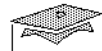
Complete the exercise for this module as directed by your instructor.

1. Copy your `Quotation2` program and change the class name to `Quotation4`. Rename the file `Quotation4.java`. Rewrite the program so that it prints the person and quotation as shown:
person said, quote
2. Make a `Quotation5` program. Rewrite the program so that instead of defining the quotation and the name within the program, the values are defined by being typed on the command line (see “Using the main Method to Take Command-Line Input” on page 4-32 of your student guide).
3. Write a program called `CircleCircumference` that calculates the circumference of a circle, when the radius is typed on the command line.

The formula for determining the circumference of a circle is:

`Circumference = pi * diameter`

Hint – Use `Math.PI`, a built-in Java constant that stores the value of pi. Convert the input to the correct data type using the information in “Converting String Arguments to Integers” on page 4-35 in your student guide.



Adding Comments to Programs

- Comments make reading programs and debugging easier.
- Comments are especially important:
 - ▼ In large programs
 - ▼ On large teams
 - ▼ During maintenance

Adding Comments to Programs

Comments within a program make it easier to figure out what the program is doing. Later in the development and maintenance process, or even later in this course when you revise programs from previous exercises, updating a program is easier when the program contains comments. It is particularly important in longer programs, on large teams where several programmers need to read the code, and in maintenance where new programmers need to figure out what the code is doing.



Comment Structures

- Single-line comments

```
// Call the printQuote method.
// Pass the quote argument from the text entry field.
// Precede quote with the source of the quotation
// defined by the user in the quote source field.
```

```
printRef.printQuote(quoteSource, userQuote);
```

- Multiple-line comments

```
/* Call the printQuote method.
Pass the quote argument from the text entry field.
Precede quote with the source of the quotation
defined by the user in the quote source field.*/
```

```
printRef.printQuote(quoteSource, userQuote);
```

Comment Structures

Two main approaches to comments are used.

- **Single-line comments** – A “//” marker tells the compiler to ignore everything to the end of the current line. For example:


```
// Call the printQuote method.
// Pass the quote argument from the text entry field.
// Precede quote with the source of the quotation
// defined by the user in the quote source field.
```

```
printRef.printQuote(quoteSource, userQuote);
```

- **Multiple-line comments** – A “/*” character combination tells the compiler to ignore everything on *all* lines up to, and including, a comment termination marker (“*/”). For example:

```
/* Call the printQuote method.
Pass the quote argument from the text entry field.
Precede quote with the source of the quotation
defined by the user in the quote source field.
*/
```

```
printRef.printQuote(quoteSource, userQuote);
```


Sun Educational Services

Comment Conventions

- Add a comment for each closing brace.

```

1 public class Shirt3
2 {
3     public static void main (String args[])
4     {
5         double price = 29.99;
6         int ID = 42889;
7         char size = 'X';
8         System.out.println("Shirt price: " + price);
9         System.out.println("Shirt ID: " + ID);
10        System.out.println("Shirt size: " + size);
11    } // end main
12} // end class

```

Comment Conventions

Many programmers make their programs easier to read by commenting the first and last line of every class, method, and other standard Java programming language structures.

```

1 public class Shirt3
2 {
3     public static void main (String args[])
4     {
5         double price = 29.99;
6         int ID = 42889;
7         char size = 'X';
8         System.out.println("Shirt price: " + price);
9         System.out.println("Shirt ID: " + ID);
10        System.out.println("Shirt size: " + size);
11    } // end main
12 } // end class

```

It might look relatively simple to find the ending braces of the class and method in this program; however, in longer programs, it can be very difficult. Commenting the structure that each ending brace belongs to makes reading and debugging much easier.

Exercise 3: Commenting Your Code

Exercise objective – Increase readability of your program.



Tasks

1. Add comments to each line of the program you just wrote. Use either comment style. Be sure to comment each closing brace.
2. Switch seats with another student and add comments to each line of one of his or her programs.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Write code to create and use objects in a program
- ☐ Write code to create `Strings` to store a series of characters in a variable
- ☐ Describe how objects and `Strings` are stored in memory
- ☐ Use the `main` method
- ☐ Design and write code for a class
- ☐ Add comments to programs
- ☐ Write a basic Java program

Think Beyond

You now know the basics of putting a procedural Java program together. What else do you need to add to write a useful program?

Objectives

Upon completion of this module, you should be able to:

- Identify, describe, and use arithmetic, logical, and boolean operators
- Recognize whether promotion will happen automatically for a statement
- Use typecasting to manage mismatched data types
- Determine when a variable must be declared with a different data type to allow assignment
- Write code to make decisions using `if`, `if/else`, and `if/else if/else` constructs
- Write nested `if` constructs
- Write code for decision control using the `switch` keyword with `if` constructs

This module teaches you how to use operators in expressions in your code, how to manage mismatched data types in statements, and how to use those expressions in decision-making constructs.

Relevance



Discussion – Discuss the following questions.

- When you learned math in school, what characters did you use to signify what the master operations were?
- When you make a decision about an event that might have several different paths, how do you ultimately settle on one over all the others?
- How do you tell a computer about all the possibilities?

Arithmetic Operators

Programs do a lot of mathematical calculating, from the simple to the complex. Arithmetic operators let you specify how the numbers should be evaluated or combined.

Standard Operators

The operators used in the Java programming language are shown in Table 5-1.

Table 5-1 Math Operators

Purpose	Operator	Example	Comments
Addition	+	sum = num1 + num2	
Subtraction	-	diff = num1 - num2	
Multiplication	*	prod = num1 * num2	
Division	/	quot = num1 / num2	
Modulus	%	mod = num1 % num2 Where num1 is 31 and num2 is 6, mod is 1	Modulus finds the remainder of the first number divided by the second number. <div style="text-align: right;"> $\begin{array}{r} 5 \text{ R } 1 \\ 6 \overline{) 31} \\ \underline{30} \\ 1 \end{array}$ </div>
Increment	++	num++ Where num was 5, it is now 6.	Increases a number by 1. The operators can be used before or after the variable; see notes on page 5-4.
Decrement	--	num-- Where num was 5, it is now 4.	Decreases a number by 1. The operators can be used before or after the variable; see notes on page 5-4.

Increment and Decrement Operators (`++` and `--`)

A common requirement in programs is to add or subtract 1 from a variable. You can do this simply by using the `+` operator, like this: `age + 1`.

However, incrementing or decrementing by 1 is such a common action that there are specific operators for it: the `++` and `--` operators.

Use these operators with caution. The result varies in some expressions, such as assignment, depending upon whether you use the pre-increment or pre-decrement expressions (`++i` or `--i`), or the post-increment or post-decrement expressions (`i++` or `i--`).

Table 5-2 lists the increment and decrement operators.

Table 5-2 Increment and Decrement Operators

Operator	Purpose	Syntax	Example
++	Pre-Increment	<code>j = ++i;</code>	<code>int i = 6;</code> <code>int j = ++i</code> <code>i is 7, j is 7</code>
	Post-Increment	<code>j = i++;</code>	<code>int i = 6;</code> <code>int j = i++;</code> <code>i is 7, j is 6</code>
--	Pre-Decrement	<code>j = --i;</code>	<code>int i = 6;</code> <code>int j = --i</code> <code>i is 5, j is 5</code>
	Post-Decrement	<code>j = i--;</code>	<code>int i = 6;</code> <code>int j = i--;</code> <code>i is 5, j is 6</code>

The following example shows basic use of the operators.

```
1 class Increment
2 {
3     public static void main(String args[])
4     {
5         int count = 15;
6         System.out.println("++count = " + ++count);
7         System.out.println("count = " + count);
8         System.out.println("count++ = " + count++);
9         System.out.println("count = " + count);
10    }
11 }
```



Operator Precedence

- Precedence rules:
 - ▼ Contents of parentheses
 - ▼ Multiplication and division, from left to right
 - ▼ Addition and subtraction
- Complex expression:

`c = 25 - 5 * 4 / 2 - 10 + 4;`

- ▼ Result from left to right: 34
- ▼ Result according to rules of precedence: 9
- Use parentheses to explicitly indicate structure.

Operator Precedence

In a complex mathematical statement with multiple operators on the same line, how does the computer pick which operator it should use first?

Rules of Precedence

To make mathematical operations consistent, Java follows the standard mathematical rules for operator precedence:

1. Operators within a pair of parentheses
2. Multiplication and division operators, evaluated from left to right
3. Addition and subtraction operators, evaluated from left to right

If the same operator appears successively in a statement, the operators are evaluated from left to right.

Example of Need for Rules of Precedence

The following example demonstrates the need for operator precedence:

```
c = 25 - 5 * 4 / 2 - 10 + 4;
```

It is not clear what the author intended, either from the statements or any supporting comments.

- Expression result if evaluated strictly from left to right: 34

```
c = 25 - 5 * 4 / 2 - 10 + 4;
```

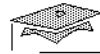
- Expression result if evaluated according to rules of precedence, indicated by the parentheses: 9

```
c = 25 - ((5 * 4) / 2) - 10 + 4;
```

Using Parentheses

Your expression will be evaluated with the rules of precedence; however, it is a good idea to use parentheses to provide the structure you intend:

```
c = (((25 - 5) * 4) / (2 - 10)) + 4;  
c = ((20 * 4) / (2 - 10)) + 4;  
c = (80 / (2 - 10)) + 4;  
c = (80 / -8) + 4;  
c = -10 + 4;  
c = -6;
```



Promotion and Typecasting

- Assignments and expressions can cause a mismatch between variable data types, and storage locations for results.
- Examples of causes:
 - ▼ Multiplication can cause data to exceed data type size
 - ▼ Division can introduce decimal places

Promotion and Typecasting

Assigning a variable or an expression to another variable, or using mathematical expressions, can lead to a mismatch between the data types of the calculation and the storage location you are using to save the result. When everything you use in a calculation is the same type, the mismatch rarely occurs. When you use many variables of different types, you are more likely to encounter this mismatch.

You need to deal with this in your code; this section covers how.



Examples of Data and Data Type Mismatch

- Right-side data smaller than left:

```
byte num1 = 53;
byte num2 = 47;
long num3;
num3 = num1 * num2;
```

- Right-side data larger than left:

```
int num1 = 53;
int num2 = 47;
byte num3;
num3 = (num1 + num2);
```

- Right-side data and type larger than left:

```
int num1 = 53;
int num2 = 47;
byte num3;
num3 = num1 * num2;
```

Examples of Data and Data Type Mismatch

Right-side Data Type Smaller Than Left-side Data Type

In some situations, you might need to assign a byte to a long.

```
byte num1 = 53;
byte num2 = 47;
long num3;
num3 = num1 * num2;
```

This is done automatically by the compiler, since no data would be lost, and the compiler allows you to store smaller data types in larger ones. In this type of mismatch, you do not need to do anything.

Right-side Data Type Bigger Than Left-side Data Type

For example, consider the following assignment:

```
int num1 = 53;
int num2 = 47;
byte num3;
num3 = (num1 + num2); // causes compiler error
```

This technically should work fine, because a `byte`, while smaller than an `int`, is large enough to store 100. However, the compiler will not make this assignment. In this type of mismatch, you need to *typecast* the right side data type down to match the left side data type.

Right-side Data Type and Data Bigger Than Left-side Data Type

This might occur in a program, as well:

```
int num1 = 53;
int num2 = 47;
byte num3;
num3 = num1 * num2;
```

2491 is too big for a `byte`, so you would not want to force the compiler to complete the assignment. In this type of mismatch, you need to change your code so that you declare the variable on the left side as a larger data type.



Sun Educational Services

Promotion

- Happens automatically when:
 - ▼ Assigning smaller type to large type
 - ▼ Assigning integral type to floating point type

```
long big = 6; // legal
int small = 99L; // illegal
```

Promotion

Promotions, such as converting a byte to a long in the previous example, happen automatically if no data would be lost by doing so:

- You assign a smaller type (on the right of the =) to a larger type (on the left of the =).
- You assign an integral type to a floating point type, because there are no decimal places to lose.

The following examples show what will be automatically promoted by the compiler and what will not be.

```
long big = 6;
```

6 is an int type; promotion works because it converts an int to a long.

```
int small = 99L;
```

99L is a long; this is illegal because the assignment asks to convert a long to an int. The long is twice the size of an int. A solution is to typecast instead.



Typecasting

- Lowers the range of a value ("chops it down")

```
variable_identifier = (target_type) value
```

```
int num1 = 53;
int num2 = 47;
byte num3;
num3 = (byte) (num1 + num2); // No data loss
```

```
int myInt;
long myLong = 99L;
myInt = (int) (myLong); // No data loss, only zeroes.
```

```
int myInt;
long myLong = 123987654321;
myInt = (int) (myLong); // Number is "chopped"
```

- Typecasting floating point to integral removes values to the right of the decimal point

Typecasting

This section covers the syntax and examples of typecasting.

Definition

Typecasting lowers the range of a value, quite literally chopping it down to a smaller size, by changing its type, such as converting a long to an int. It is typically done so that you can use methods that accept only certain types as arguments, to be able to assign values to a variable of a smaller data type, or to save memory.

Syntax and Examples

Put the *target data type* (the type the value is being typecast to) in parentheses in front of the item that you are typecasting.

Syntax:

```
variable_identifier = (target_type) value
```

The *value* can be a literal value or an expression.

Examples:

```
int num1 = 53;
int num2 = 47;
byte num3;
num3 = (byte) (num1 + num2); // No data loss

int myInt;
long myLong = 99L;
myInt = (int) (myLong); // No data loss, only zeroes.
                        // A much larger number would
                        // result in data loss.

int myInt;
long myLong = 123987654321;
myInt = (int) (myLong); // Number is "chopped"
```

If you typecast a float or double with values to the right of the decimal places to an integral type such as an int, all decimal values will be lost. However, this can be useful sometimes if you just want to round the number down to the whole number: 51.9 becomes 51, for example.



Sun Educational Services

Integral and Floating Point Data Types

- Adding integral types using + converts primitive data types to int (or higher).

```
short a,b,c;
a = 1 ;
b = 2 ;
c = a + b ;
```

- Floating point values default to double if you do not specify they are floats.

```
float float1 = 27.9;           // causes error
float float1 = 27.9F;          // would work correctly
float float1 = (float) 27.9;   // would work correctly
```

Integral and Floating Point Data Types

The Java technology compiler makes certain assumptions when it evaluates expressions; you need to understand these to make the appropriate casts or other accommodations.

Integral Data Types and Operations

When you use primitive data type values in an expression with certain operators (*, /, -, +, %), the values are automatically converted to int (or higher if necessary), then added.

The value is calculated by promoting the operands (the values you are adding) to one of the following

- What the data type of the conversion result would be
- A result type that is the same as the operands, if the operands have a larger type than the result would require

This might cause overflow or loss of precision. For example, the following example will cause an error because two of the three operands (a and b) are automatically promoted from short to an int before they are added:

```
short a, b, c;  
a = 1 ;  
b = 2 ;  
c = a + b ;
```

In the last line, a and b are converted to ints. Then the assignment operator (=) attempts to assign the int value to the short value. However, the compiler will not do the assignment.

The code will work if you do either of the following:

- Declare c as an int in the original declaration:

```
int c;
```
- Typecast the (a+b) addition in the assignment line:

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

Floating Point Data Types and Assignment

Just as integral types default to int under some circumstances, values assigned to floating point types always default to double, unless you specifically state that the value is a float.

For example, the following line would cause a compiler error. 27.9 is assumed to be a double, so a compiler error occurs because a double cannot fit into a float.

```
float float1 = 27.9;
```

Both of the following would work correctly:

- The F notifies the compiler that 27.9 is a float:

```
float float1 = 27.9F;
```
- 27.9 is cast to a float:

```
float float1 = (float) 27.9;
```

Example

The following program uses principles from this section to take your age, then tell you how many days old you are (to within one year) and how many weeks old you are.

If the L were not placed after the 365 and the 24, the program would not work because, the program would be multiplying all integers. Thus, with a large age like 100, the program would return a negative number of second.

```
1 class Age
2 {
3     public static void main(String args[])
4     {
5         int ageYears, ageDays;
6         long ageSeconds;
7
8         ageYears = Integer.parseInt(args[0]);
9         ageDays = ageYears * 365;
10        ageSeconds = (ageYears * 365 * 24L * 60 * 60);
11        System.out.println("You are " + ageDays + " days old.");
12        System.out.println("You are " + ageSeconds + " seconds old.");
13    }
14 }
```


Exercise 1: Using Operators and Casting

Exercise objective – Practice using operators and type-casting.



Tasks

Follow the directions from your instructor to locate the exercises directory for this module.

1. Rewrite the example program on 5-16 of your student guide and name it `Age.java`. Add a line to calculate and print your age in minutes and milliseconds (1/1000 of a second). Test the program with 1, 24, and 100.
2. Write a program called `Multiply` that takes two integers you type at the command line, multiplies them and prints the result. Test the program with 1-digit ints, 5-digit ints, and 9-digit ints.
3. Write a program called `Temperature` that takes a temperature in Fahrenheit that you type on the command line and prints it in Celsius. (To convert from Fahrenheit to Celsius, subtract 32, multiply by 5, and divide by 9.)

Notes

Logical and boolean Operators

In addition to arithmetic operators, there are logical operators such as `<` and `>` for less than and greater than, and boolean operators such as `&&` for AND.

If you expect a computer to make a decision, the answer is either yes or no (true or false). There is no result that is “maybe.”

Logical Operators for Primitive Data Types

Table 5-3 lists the different conditions you can test for in conditional structures and loops.

Table 5-3 Logical Operators

Operation	Operator	Example
is equal to	<code>==</code>	<code>(i == 1)</code>
is not equal to	<code>!=</code>	<code>(i != 1)</code>
is less than	<code><</code>	<code>(i < 1)</code>
is less than or equal to	<code><=</code>	<code>(i <= 1)</code>
is greater than	<code>></code>	<code>(i > 1)</code>
is greater than or equal to	<code>>=</code>	<code>(i >= 1)</code>

boolean *Operators*

You will also need the ability to make a single decision based on more than one comparison. Under such circumstances you apply boolean logic and operators. Table 5-4 lists the boolean operators in the Java programming language.

Table 5-4 Boolean Operators

Operation	Operator	Example
AND	&&	<pre>int i = 2; int j = 8; ((i < 1) && (j > 6));</pre>
OR		<pre>int i = 2; int j = 8; ((i < 1) (j > 6));</pre>
NOT	!	<pre>int i = 2; int j = 8; (!(i < 1));</pre>



Basic Parts of an if Construct

- An if construct lets you perform certain statements if a condition is true.
- Most include these three parts:
 - ▼ *initialize* statement (optional)
 - ▼ *boolean_expression* (required)
 - ▼ *code_block*, runs if *boolean_expression* is true (required)

Basic Parts of an if Construct

An if statement, or an if *construct*, lets you perform certain statements if a condition is true. There are a few variations on the basic if construct, but the same three elements are included: *initialize* statement, *boolean_expression*, and *code_block*.

Assume, for example, that at work you drink coffee only in the mornings. If it is morning, you drink coffee; otherwise, you do not. The elements would be part of this example:

1. The *initialize* statement or statements are occasionally included in if statements. They set variables that you will use in the statement to the appropriate values. You could set `int hour` to 10:00 if it is already 10:00 AM.

Note – This is optional and is not a standard part of an if statement.

2. The *boolean_expression* is tested. The expression in the coffee example is `hour >= 8 && hour < 12`. (The variable used here is typically the one initialized in *initialize*.)

3. If *boolean_expression* is true, the *code_block* is run. The statement in this example would be `drink()` or a similar method.

If *boolean_expression* is not true, the program skips to the brace marking the end of the `if` construct code block.

Basic if Construct

The if construct allows your program to make simple decisions based on stored values.

Syntax:

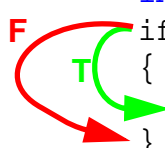
```
// optional: initialize variable in boolean_expression
if (boolean_expression)
{
    code_block;           // Braces are required if more
                        // than one statement is present.
}                        // You can use them with only one
                        // statement to make code clearer.
```

If the *boolean_expression* is true, the *code_block* is completed; if not, the program skips to the second brace.

Example:

The following example shows the code for deciding whether to drink coffee, based on whether it is morning (between 8:00 AM and 12:00 PM, on a military time system).

```
1 public class Coffee
2 {
3     public static void main (String args[])
4     {
5         int hour = Integer.parseInt(args[0]);
6         if (hour >= 8 && hour <12)
7         {
8             System.out.println("Drink coffee");
9         }
10    }
11 }
```



T = Processing if *boolean_expression* is true

F = Processing if *boolean_expression* is false

Figure 5-1 Simple if Construct: Coffee

This example prints the letter grade for a corresponding number grade. The number grade is typed into the program on the command line, when you run the program, and compared to numbers in the code.

```

1 public class LetterGrade
2 {
3     public static void main (String args[])
4     {
5         int grade = Integer.parseInt(args[0]);
6         if (grade >= 90)
7         {
8             System.out.println("A");
9         }
10    }
11 }

```

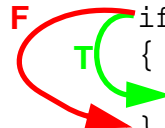


Figure 5-2 Simple if Construct: Letter Grade

As the program runs, it checks whether the boolean expression is true or false.

- If it is true then the statement or statements are executed
- Otherwise, the statement or statements are ignored

Choosing Between Two Statements (if/else)

Often you will need to decide between two different statements with one condition (a boolean expression). Use `if` to state what happens if the boolean expression is true, and `else` to state what happens if it is false.

For example, assume that you drink coffee in the morning and tea the rest of the time. You could set up a program to determine which beverage you should be drinking.

Syntax:

```
// optional: initialize variable in boolean_expression
if (boolean_expression)
{
    code_block;
}
else
{
    code_block;
}
```

Example:

This is another example of the coffee decision-making process.

```
1 public class Coffee
2 {
3     public static void main (String args[])
4     {
5         int hour = Integer.parseInt(args[0]);
6         if (hour >= 8 && hour <12)
7         {
8             System.out.println("Drink coffee");
9         }
10        else
11        {
12            System.out.println("Drink tea");
13        }
14    }
15 }
```

Figure 5-3 Simple if/else Construct: Coffee

This example expands the LetterGrade program.

```

1 public class LetterGrade
2 {
3     public static void main (String args[])
4     {
5         int grade = Integer.parseInt(args[0]);
6         if (grade >= 90)
7         {
8             System.out.println("A");
9         }
10        else
11        {
12            System.out.println("Not an A");
13        }
14    }
15 }

```

Figure 5-4 Simple if/else Construct: Letter Grade

Choosing From More Than Two Statements (if/else if/else)

You can chain `if` and `else` structures together to state multiple outcomes for several different boolean expressions. This provides more power, to state multiple possible expressions and outcomes.

If in the coffee example you drink coffee in the morning, tea in the afternoon, and water at any other time of day, you could express that in the additional *boolean_expression* and `else` portions of the syntax.

Syntax:

You can include multiple `else if` statements.

```
// optional: initialize variable in boolean_expression
if (boolean_expression)
{
    code_block;
}
else if (another_boolean_expression)
{
    code_block;
}
else
{
    code_block;
}
```

Example:

This program expands the coffee example.

```

1 public class Coffee
2 {
3     public static void main (String args[])
4     {
5         int hour = Integer.parseInt(args[0]);
6         F if (hour >= 8 && hour <12)
7             T {
8                 System.out.println("Drink coffee");
9             }
10        F else if (hour >= 12 && hour <5)
11            T {
12                System.out.println("Drink tea");
13            }
14        else
15        {
16            System.out.println("Drink water");
17        }
18    }
19 }

```

Figure 5-5 Simple if/else if/else Construct: Coffee

This program expands the LetterGrade example.

```

1 public class LetterGrade
2 {
3     public static void main (String args[])
4     {
5         int grade = Integer.parseInt(args[0]);
6         if (grade >= 90)
7         {
8             System.out.println("A");
9         }
10        else if ((grade < 90) & (grade >= 80))
11        {
12            System.out.println("B");
13        }
14        else if ((grade < 80) & (grade >= 70))
15        {
16            System.out.println("C");
17        }
18        else if ((grade < 70) & (grade >= 60))
19        {
20            System.out.println("D");
21        }
22        else
23        {
24            System.out.println("Fail");
25        }
26    }
27 }

```

Figure 5-6 Simple if/else if/else Construct: Letter Grade

The following example selects a card at random from a deck.

```

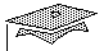
1 public class PickACard
2 {
3     public static void main(String args[])
4     {
5         int whichCard = (int) (Math.random() * 52);
6         System.out.println("The number generated was " + whichCard);
7         int suit = whichCard / 13;
8         int number = whichCard % 13;
9
10        if (number == 0)
11        {
12            System.out.print("Ace of ");
13        }
14        else if (number == 10)
15        {
16            System.out.print("Jack of ");
17        }
18        else if (number == 11)
19        {
20            System.out.print("Queen of ");
21        }
22        else if (number == 12)
23        {
24            System.out.print("King of ");
25        }
26        else
27        {
28            System.out.print((number+1) + " of ");
29        }
30
31        if (suit == 0)
32        {
33            System.out.println("Diamonds");
34        }
35        else if (suit == 1)
36        {
37            System.out.println("Spades");
38        }
39        else if (suit == 2)
40        {
41            System.out.println("Clubs");
42        }
43        else
44        {

```

```
45      System.out.println("Hearts");
46    }
47  }
48 }
```

This example shows a more complex decision construct, using boolean operators.

```
1 public class IfDaysInMonth
2 {
3     public static void main(String args[])
4     {
5         int month = Integer.parseInt(args[0]);
6         if (month == 1 || month == 3 || month == 5 || month == 7 ||
7             month == 8 || month == 10 || month == 12)
8         {
9             System.out.println("There are 31 days in that month.");
10        }
11        else if (month == 2)
12        {
13            System.out.println("There are 28 days in that month.");
14        }
15        else if (month == 4 || month == 6 || month == 9 || month == 11)
16        {
17            System.out.println("There are 30 days in that month.");
18        }
19        else
20        {
21            System.out.println("Invalid month.");
22        }
23    }
24 }
```

Testing Equality Between Strings

- Using `==` with `Strings` compares values (references).
- To test equality of the string of characters, use `equals`.

```

1 public class Naming
2 {
3     public static void main (String args[])
4     {
5         String name1;
6         String name2;
7         name1 = "David St. Hubbins";
8         name2 = "Nigel Tufnel";
9         if (name1.equals(name2))
10        {
11            System.out.println("Same name.");
12        }
13    }
14}

```

Testing Equality Between Strings

If you use the `==` and other boolean operators from page 5-20 with `Strings`, it tests whether the strings' *references in memory* are equal, not their contents. If you want to test equality between the strings of characters, such as whether the name "Nigel Tufnel" is equal to "David St. Hubbins," use `equals`.

Example

```

1 public class Naming
2 {
3     public static void main (String args[])
4     {
5         String name1;
6         String name2;
7         name1 = "David St. Hubbins";
8         name2 = "Nigel Tufnel";
9         if (name1.equals(name2))
10        {
11            System.out.println("Same name.");
12        }
13    }
14 }

```



Sun Educational Services

Nested if Statements

- Used when considering multiple factors before making a decision
- Example: Multiple conditions (weather, adult or child) determining what you do each day

Nested if Statements

Sometimes you might need to run an `if` statement as part of another `if` statement. Consider what you would need to decide if there were multiple factors to consider before making a decision. The example in this section shows how to use nested `if` statements to make sure you are prepared for the first factor, the weather, and to determine how to behave based on a second factor, your age (carefree child or responsible adult).

```
1 public class WhatToDo
2 {
3     public static void main(String args[])
4     {
5         //weather should be "snow", "rain", "sleet", or "sunny"
6         String weather = args[0];
7         int age = Integer.parseInt(args[1]);
8         if (weather.equals("snow"))
9         {
10             if (age < 15)
11             {
12                 System.out.println("Build a snowman.");
13             }
14             else if (age < 90)
15             {
16                 System.out.println("Shovel the driveway.");
17             }
18             else
19             {
20                 System.out.println("Enjoy watching the snow.");
21             }
22         }
23         else if (weather.equals("rain") || weather.equals("sleet"))
24         {
25             if (age >=16)
26             {
27                 System.out.println("Drive safely.");
28             }
29             else
30             {
31                 System.out.println("Walk safely.");
32             }
33         }
34         else
35         {
36             if (age > 15 && age < 25)
37             {
38                 System.out.println("Hang out at the beach.");
39             }
40             else
41             {
42                 System.out.println("Enjoy the sun.");
43             }
44         }
45         if (! weather.equals("sunny"))
46         {
```

```
47      System.out.println("Bring an umbrella.");  
48  }  
49  }  
50 }
```

Exercise 2: Using the `if` Construct



Exercise objective – Practice using `if`, `if/else`, and `if/else if` constructs.

Tasks

1. Write a program called `Division` that does the following:
 - ▼ Takes three command-line arguments
 - ▼ Divides the third number by the first and prints the result
 - ▼ Divides the third number by the second and prints the result.
 - ▼ Checks to be sure the first and second numbers are not equal to zero
2. Write a program called `DaysOfWeek` that prints the day of the week, based on command-line input of 1 through 7.

The switch Keyword

if Construct Example

Consider the next if structure, which tests a single variable against a range of values.

```

1 public class CustomerStatusIf
2 {
3     public static void main (String args[])
4     {
5         char status; // Customer status is represented by a char
6                     // status; value is from command-line args[]
7
8         status = args[0].charAt(0); // asks for a char from input
9
10        if (status == 'a')
11        {
12            System.out.println("Process order");
13        }
14        else if (status == 'b')
15        {
16            System.out.println("Process order with discount");
17            System.out.println("Process order, mention sale");
18        }
19        else if ((status == 'c') || (status == 'd'))
20        {
21            System.out.println("Process order, mention sale");
22        }
23        else if (status == 'e')
24        {
25            System.out.println("Forward call to manager");
26            System.out.println("Do not process orders");
27        }
28        else
29        {
30            System.out.println("Ask for Status letter on catalog");
31        }
32    }
33 }
```

Figure 5-7 Excessively Complex if Construct

This if structure looks somewhat long and complex. It has also forced some repetition because the third statement, printing the phrase "Process order and mention sale times," appears twice on lines 17 and 21. The code could be rewritten to prevent this duplication, but the code would then become even more complicated looking.

if Construct Rewritten Using switch

Figure 5-7 could be re-written using the switch construct:

```

1 public class CustomerStatusSwitch
2 {
3     public static void main (String args[])
4     {
5         char status;// Customer status is represented by a char
6             // status; value is from command-line args[]
7
8         status = args[0].charAt(0);
9
10        switch (status)//"switch" sets the variable that will
11            // be compared to values in the rest
12            // of the program.
13        {
14            case 'a':// "case" means "if the value after the word
15                //switch is equal to", a boolean expression
16                System.out.println("Process order");
17                break;
18            case 'b':
19                System.out.println("Process order with discount");
20            case 'c':
21            case 'd':
22                System.out.println("Process order, mention sale");
23                break;
24            case 'e':
25                System.out.println("Forward call to manager");
26                System.out.println("Do not process order");
27                break;
28            default:
29                System.out.println("Ask for Status letter on catalog");
30                // The default statement has the same function
31                // as the else statement
32        }
33    }
34 }
```

Figure 5-8 Rewriting if Construct Using switch and break

In this example, if `status` were equal to `a`, lines 1 through 17 would be processed, then the next line to be processed would be line 32. If `status` were equal to `b`, then lines 1 through 23 would be processed, even though case 2's statement is on line 19, because it has no `break`.

Each case acts as a label for some code. The label defines the entry point into the `switch`. The `break` line causes the flow to exit the `switch`.

Syntax for switch Constructs

Use the following syntax when writing switch constructs.

```
// optional: initialize variable
switch (variable)
{
    case literal_value:
        code_block;
        break; // break is typical but not required if you
               // want to process subsequent statements.

    case another_literal_value:
        another_code_block;
        break;

    // any additional case labels, statements and breaks
}
```



Syntax for switch Constructs

- *variable* – Can be only char, byte, short, or int
- case – Equivalent to “if the variable after switch is”
- case labels – Entry points to the block of statements
- *literal_value* – Must be literals
- *code_block* – The actions to perform
- break – Ends a switch statement
- default – Means “else”

Three new keywords are used: case, default, and break. The following are important rules for using them:

- *variable* – The variable type, status in the previous example, can be only char, byte, short, or int.
- case – The case keyword is equivalent to saying “if the variable listed after switch is equal to”. In this example, case means “if s ==”.

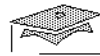
The case labels are entry points to the block of statements. If the break statement were completely omitted in the previous example, then when `s == 1` all the statements would be processed.

- *literal_value* – The values after case, called *case labels*, must be literals. They cannot be variables, expressions, or method calls.
- *code_block* – The actions to perform (what to do when you get a match). You can use any valid statement or method call.

- `break` – Ends a `switch` statement.

You do not have to use `break` to terminate all cases. It permits a case without any statements to act as an `or` operator (`|`) for two literals. This characteristic is called *fall-through*.

- `default` – Means “else”. The statement following it is performed if none of the previous statements were true. The `default:` case is the same as the `else` in an `if` construct. `default:` is optional.



When to Use switch Constructs

When all the boolean expressions:

- Are equality tests
- Are tests against a *single* variable, such as `customerStatus`
- Test the value of an `int`, `short`, `byte`, or `char` type

When to Use switch Constructs

The `switch` construct is only for testing equality, not whether values are greater than or less than a single value. You cannot use the `switch` construct to test against more than one value, and you can only use it with integral data types.

The example is a good candidate for converting to a `switch` construct because all the boolean expressions:

- Are equality tests
- Are tests against a *single* variable, such as `customerStatus`
- Test the value of an `int`, `short`, `byte`, or `char` type

The following example prints the number of days in the month, when a number is typed on the command line.

```
1 public class SwitchDaysInMonth
2 {
3     public static void main(String args[])
4     {
5         int month = Integer.parseInt(args[0]);
6         switch(month)
7         {
8             case 1:
9             case 3:
10            case 5:
11            case 7:
12            case 8:
13            case 10:
14            case 12:
15                System.out.println("There are 31 days in that month.");
16                break;
17            case 2:
18                System.out.println("There are 28 days in that month.");
19                break;
20            case 4:
21            case 6:
22            case 9:
23            case 11:
24                System.out.println("There are 30 days in that month.");
25                break;
26            default:
27                System.out.println("Invalid month.");
28                break;
29        }
30    }
31 }
```

Note – You could nest other loop constructs inside switch statements.

Exercise 3: Using the switch Construct



Exercise objective – Practice using the `switch` construct in decision-making programs.

Tasks

1. Convert the days of the week exercise you wrote previously so that it now uses `switch`. Call it `DaysOfWeek2`.
2. Write a program called `Month` that returns the name of the current month, based on command-line input of 1 through 12 for the month, and the current year. Use `switch` to tell how many days are in each month. (Allow for leap year, as well.)

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Identify, describe, and use arithmetic, logical, and boolean operators
- ☐ Recognize whether promotion will happen automatically for a statement
- ☐ Use typecasting to reconcile mismatched data types
- ☐ Determine when a variable must be declared with a different data type to allow assignment
- ☐ Write code to make decisions using `if`, `if/else`, and `if/else if/else` constructs
- ☐ Write nested `if` constructs
- ☐ Write code for decision control using the `switch` keyword with `if` constructs

Think Beyond

What elements are missing from the decision constructs you learned?

Objectives

Upon completion of this module, you should be able to:

- Write code using `while` loops
- Write code using `for` loops
- Write nested loops
- State when to use a `for` loop or a `while` loop
- Write code using `do` loops
- Describe how to use the `continue` keyword to control loops within `while`, `for`, or `do` loops

This module covers the different types of loop constructs, which let you control repetition of a set of statements.

Relevance



Discussion – What are some situations when you would want to continue performing a certain action, as long as a certain condition existed?

Would you perform the same action over and over again, or would you change part of the routine?



Loops

- Enable you to:
 - ▼ Perform certain actions as long as a certain condition exists when checked
 - ▼ Change the value of the item being checked
- Have the following components:
 - ▼ *initialize*
 - ▼ *boolean_expression*
 - ▼ *code_block*
 - ▼ *update* (or increment)

Loops

A loop, in Java and other programming languages, enables you to check and recheck a decision to execute and re-execute a block of code.

Loops also let you increment or decrement the value of the item being checked, to control the number of times you repeat an action.

For example, on a Monday at work, you might start drinking coffee at 8:00 AM and keep drinking until it is noon. A loop could express how that would be done in a program.

All loops have the following components.

1. The *initialize* statement or statements are processed first and only once, setting variables that you will use in the loop to the appropriate values. If it is already 10:00 AM when you get to work, you would initialize `hour` to 10.
2. The *boolean_expression* is tested; this determines whether the *code_block* will be run. The expression in the coffee example is `hour >= 8 && hour <= 12`. The variable used here is typically the one initialized in *initialize*.
3. The *code_block* is run if the *boolean_expression* is true. The statement in this example would be `drink()` or printing to the screen “Keep drinking coffee” or a similar method. If it is not true, the program skips to the brace marking the end of the code block.
4. *update* is run, changing the variable specified in *initialize* based on what happened in *code_block*. The time in hours or minutes is incremented or decremented, using a statement such as `hour++`, `hour +1`, and so on.

This is optional, but is typically used.

The while Loop

Use the syntax shown here for the while loop.

Syntax:

```
initialization
while (boolean_expression)
{
    code_block;
    update; // optional
}
```

Examples:

This example counts down from 10 to 0, then prints “Blast Off”.

```
1 public class WhileCountDown
2 {
3     public static void main(String args[])
4     {
5         int count = 10;
6         while (count >= 0)
7         {
8             System.out.println(count);
9             count--;
10        }
11        System.out.println("Blast Off");
12    }
13 }
```

Figure 6-1 Basic while Loop: Countdown

The following variation on the classic “Hello World” program prints a statement as long as the counter variable is less than 100.

```
1 public class WhileHello
2 {
3     public static void main(String args[])
4     {
5         int counter=0;
6         while (counter<100)
7         {
8             System.out.println("Hello. My name is Inigo Montoya.");
9             counter++;
10        }
11    }
12 }
```

Figure 6-2 Basic while Loop: Hello



Sun Educational Services

Nested Loops

- Required to print this:

```
cccccc  
cccccc  
cccccc
```

- Inner loop prints rows, outer loop prints columns
- "Rows go slow, columns go fast"

Nested Loops

Consider what is necessary to draw a rectangle like the one shown here:

```
@@@@@@@@@@  
@@@@@@@@@@  
@@@@@@@@@@
```

You could use a `while` loop to draw one row of the rectangle. Then, you could place that loop inside another loop to draw three rows. This inner loop is a *nested loop*. This is a very useful technique for solving a variety of problems.

The following example shows how this could be done.

The code in Figure 6-3 prints a rectangle of @ symbols, ten columns wide by three rows long (three rows composed of 10 @ symbols each). Printing each row is done by the inner loop; the outer loop prints that code three times.

```

1 public class NestedWhileLoop
2 {
3     public static void main (String args[])
4     {
5         int colLength = 3;
6         int rowLength = 10;
7         int colCount = 0;
8         int rowCount;
9
10        while (colCount < colLength)
11        {
12            rowCount = 0;
13            while (rowCount < rowLength)
14            {
15                System.out.print('@');
16                rowCount++;
17            }
18            System.out.println();
19            colCount++;
20        }
21    }
22 }

```

Figure 6-3 Nested while Loop

An easy way to conceptualize nested loops is that “rows go slow, columns go fast.”

Discussion – How could you alter the code in Figure 6-3 to print a rectangle 4 by 11?

How could you make the rectangle 3 by 10 without using the numbers 0, 1, 3, or 10?



Exercise 1: Using the while Loop

Exercise objective – Practice using the while loop.



Tasks

Follow the directions from your instructor to locate the exercises directory for this module.

1. Write a program called `WhileOne` that counts from 1 to 10.
2. Print the days of the week using a while and the switch statement previously developed. Call it `DaysOfWeek3.java`.
3. Write a program called `WhileTimes` that uses for loops to print the first 10 times tables.

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100



The for Loop

- Loops through a sequence of statements a pre-determined number of times
- Same as `while`, but more centralized structure

The for Loop

The `for` loop allows your program to loop through a sequence of statements a pre-determined number of times. It operates in exactly the same way as the `while` loop, including the fact that it is a zero/many loop, but has a more centralized structure for counting through a range of values.

Syntax:

The structure of the for loop is as shown; the numbers indicate the order in which they are completed:

1. The *initialize* statement or statements are processed first.
2. The *boolean_expression* is tested.
3. If it is true, the *code_block* is run. If it is not true, the program skips to the brace marking the end of the code block.
4. *update* is run.

Steps 2–4 repeat until the *boolean_expression* is false. At that point, as with the while loop, the program skips immediately from the *boolean_expression* to the end of the code block (the ending brace surrounding the *code_block*).

```

      1 (run only once)           2           4
for ( initialize[, initialize]; boolean_expression; update[, update] )
{
    3
    code_block;
}
```

The following bullets explain the syntax.

- *initialize* – This section is processed once, before any other part of the loop. Loop counters should be initialized here. Any variable *declared* here can be used only within the loop.

Note – You can declare only one variable within a loop, though you can initialize as many as you need.

- *boolean_expression* – This section is processed just before each iteration of the loop. If the condition is true then the loop will iterate again, otherwise it will not.
- *code_block* – This is the body, the statements to be processed with every loop iteration.
- *update* – This section is processed after the body but before each subsequent re-test of the *boolean_expression*. Loop counters should be updated here. Separate multiple updates with commas.

Examples:

This example repeats the while loop countdown example from page 6-5, rewritten using a for loop, and makes the computer beep while doing so.

```
1 public class ForCountDown
2 {
3     public static void main(String args[])
4     {
5         for(int count=10; count>=0; count--)
6         {
7             System.out.println(count);
8         }
9     System.out.println("Blast Off");
10 }
11 }
```

Figure 6-4 Basic for Loop: Countdown

This example repeats the while loop hello example from page 6-6, rewritten using a for loop.

```
1 public class ForHello
2 {
3     public static void main(String args[])
4     {
5         for(int counter=0; counter<10; counter++)
6         {
7             System.out.println("Hello. My name is Inigo Montoya.");
8         }
9     }
10 }
```

Figure 6-5 Basic for Loop: Hello

Consider this situation. You need to simultaneously count up from 0 in steps of 2 *and* down from 100 in steps of 3, stopping when the first number becomes larger than the second. The for loop for this is:

```
1 public class TwoNumbersLoop
2 {
3     public static void main (String args[])
4     {
5         int i, j;
6         for (i=0, j=100 ; i <= j ; i=i+2, j=j-3)
7             System.out.println(j + " is not smaller than " + i + " yet");
8     }
9 }
```

The following example shows a nested for loops, printing a rectangle of @ symbols, rewriting the while loop on page 6-8 and using command-line arguments to input the height and width.

```
1 public class NestedForLoop
2 {
3     public static void main(String args[])
4     {
5         int width = Integer.parseInt(args[0]);
6         int height = Integer.parseInt(args[1]);
7
8         for(int row=0; row < height; row++)
9         {
10             for(int col=0; col < width; col++)
11             {
12                 System.out.print("@");
13             }
14             System.out.println();
15         }
16     }
17 }
```

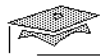
Exercise 2: Using the for Loop

Exercise objective – Practice using the for loop.



Tasks

1. Write a program called `ForOne` that counts from 1 to 10.
2. Write a program called `ForTimes` that uses `for` loops to print the first 10 times tables.



The do Loop

- The do loop is:
 - ▼ Similar to `while` loop
 - ▼ A *one/many iterative loop*
- `while` loop and `for` loop are *zero/many iterative loops*

The do Loop

You can also use the do loop to perform functions similar to the `while` loop.

The `while` loop and the `for` loop are *zero/many iterative loops*. This means that the condition part is processed before the body of the loop and if it is immediately false then the body will not be processed at all.

The do loop is a *one/many iterative loop*. The condition is at the bottom of the loop and is processed *after* the body. The body of the loop is therefore processed at least once. If you want the statement or statements in the body to be processed at least once, use a do loop instead of a `while` loop or `for` loop.

Syntax:

```
initialize // optional
do
{
    update // optional, position depends on program
    code_block;
}
while (boolean_expression);    // Semicolon is mandatory.
```

Note – The semicolon after the *boolean_expression* part of the loop is mandatory. That is because in this structure, the boolean expression is at the end of the loop, whereas in a while loop, the while is at the beginning of the loop leading into the body. Semicolons are used only when you are ending something.

Other structures such as the while loop have either a single statement to process (so the entire construct ends with a semicolon) or a code block (so it ends with a closing brace). This is not true of the do loop.

Examples:

The following example generates random numbers until it generates a 3. There is no *update* because the random number changes the value being checked each time.

```
1 public class FindAThree
2 {
3     public static void main(String args[])
4     {
5         int num;
6
7         do
8         {
9             num = (int) (Math.random() * 20) + 1;
10            System.out.println(num);
11        }
12        while (num != 3);
13    }
14 }
```

Figure 6-6 Basic do Loop: Find a 3

Rewriting a while Loop With a do Loop

Consider the problem of rolling two dice, one after the other. The point is to keep rolling the second until it shows the same number as the first. The following code shows how this could be achieved using the while loop.

Note – Assume the methods for rolling each die are available.

```
1 public class RollWhile
2 {
3     public static void main (String args[])
4     {
5         int die1, die2;
6         die1 = rollDie1();
7         die2 = rollDie2();
8         while (die1 != die2)
9         {
10            die2 = rollDie2();
11        }
12    }
13 }
```

Lines 3 and 6 are the same, rolling the second die and storing the value in the variable die2. This structure must be used because you must ensure that the line is processed at least once. The same program is better written using a do loop:

```
1 public class RollDo
2 {
3     public static void main (String args[])
4     {
5         int die1, die2;
6         die1 = rollDie1();
7         do
8         {
9            die2 = rollDie2();
10        }
11        while (die1 != die2);
12    }
13 }
```

The statement on line 5 appears only once, because the do loop ensures it is processed at least once.

Exercise 3: Using the do Loop

Exercise objective – Practice using the do loop.



Tasks

Write a program called `DoTwoRandom` that generates and prints two random numbers in the range 0 to 5, ensuring they are not the same. Run your program several times to make sure it works correctly.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications



Comparing Loop Constructs

- `while` – Iterates indefinitely through statements and performs the statements zero or more times
- `do` – Iterates indefinitely through statements and performs the statements *one* or more times
- `for` – Steps through statements a predefined number of times

Comparing Loop Constructs

The `for`, `while`, and `do` loops operate very similarly; however, in any given situation, one is probably better than the other. Use the following guidelines:

- Use the `while` loop to iterate indefinitely through statements and perform the statements zero or more times.
- Use the `do` loop to iterate indefinitely through statements and perform the statements *one* or more times.
- Use the `for` loop to step through statements a predefined number of times.

The `for` loop is more compact and easier to read than the `while` loop because it was designed for counting through a discrete range of values.



The continue Keyword

- Permits you to end a loop iteration
- Used inside `while`, `for`, and `do` loops only
- Should be used only when the alternative code is much more complex

The continue Keyword

The `continue` keyword permits you to end a loop iteration. It can be used inside `while`, `for`, and `do` loops only.

Consider a `for` statement that loops from 1 to 50, and, if the number is a multiple of 9, adds 1 to a count of multiples of 9.

- If the number in that loop iteration is not a multiple of 9, you must use `continue` to terminate a loop iteration.
- If it is a multiple of 9, you will proceed to the next statement, which increments the `multiplesOf9` counter.

Example:

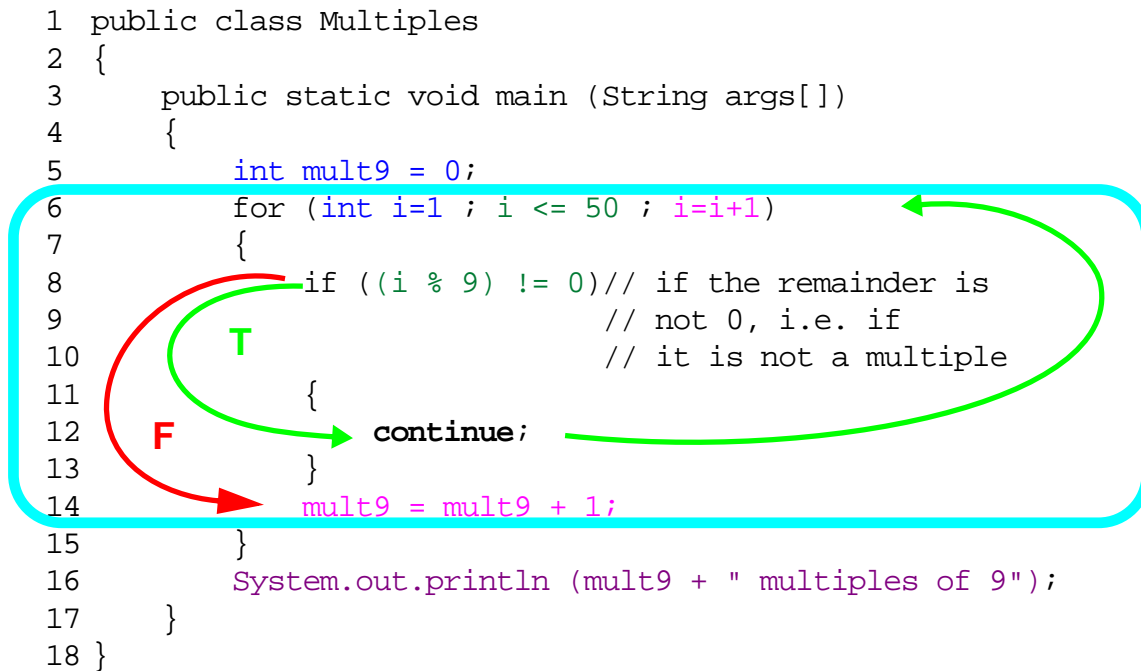


Figure 6-7 Using the continue Keyword

If the `if` statement is true (if there is a non-zero remainder of the number divided by 9, and therefore it is not a multiple of 9), then it completes the code block.

The statement in the code block is `continue`, which means that the entire `for` loop iteration should be halted and the program should skip back up to the top (the iterative part) of the `for` loop on line 17.

If the `if` statement is false (the number is a multiple of 9), then the processing within the `if` statement code braces is skipped and the next line, 14, is processed.

When to Use `continue`

Do not use `continue` any more than necessary. Use it only when the alternative code is quite complex and is made simpler by the `continue` statement.

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Write code using `while` loops
- ☐ Write code using `for` loops
- ☐ Write nested loops
- ☐ State when to use a `for` loop or a `while` loop
- ☐ Write code using `do` loops
- ☐ Describe how to use the `continue` keyword to control loops within `while`, `for`, or `do` loops

Think Beyond

You have been using procedural programming (all actions within `main`) so far. How would you apply multiple methods to the loops and decision constructs you learned in this and the previous module?

Objectives

Upon completion of this module, you should be able to:

- Describe why using multiple methods supports object orientation
- Write a method declaration for a method other than `main`
- Write a method that calls another method
- Pass parameters to a method and receive a return value
- Define static and object methods and state when to use each
- Explain what overloaded methods are and why they are useful
- Write code using overloaded methods

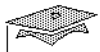
This module describes how to extend your knowledge of the `main` method to declare and call other methods.

Relevance



Discussion – How do you structure or implement the functionality of an object? That is, how do you program the actions an object can perform so that when you want the object to perform the action, you just tell it to do so?

If you ask an object to do something, such as asking an Order to submit itself to the processing warehouse or to find the total amount for itself, how much information do you expect to have to provide?

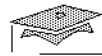


Overview of Method Use

- Programs so far have shown all actions inside `main`.
- This module covers how to divide a program's actions into task-specific methods.

Overview of Method Use

So far, this course has presented the concept that a program is a sequence of statements executed in order, all defined within the `main` method code block. However, a program might have many separate actions that it completes. Defining methods besides `main` lets you separate the statements into code blocks that can run independently of each other.



Advantages of Method Use

- Lets you ask an object to do something
- Defines exactly what an object can do
- Lets you get and set values
- Makes programs more readable and easier to maintain
- Makes development and maintenance quicker
- Enables reusable software

Advantages of Method Use

There are numerous advantages to defining class-specific methods, instead of using `main` again and again.

- Methods are the primary way to ask an object to do something.
- Methods define exactly what an object can do. Instead of having to list exactly what you want an object to do, you simply pick from a list of what it can do (its list of methods).
- Methods are the primary approach, in good object-oriented programs, for “getting” and “setting”—obtaining variable values from an object, and specifying the variables’ values. “getter” methods and “setter” methods are typically part of any object.

- Related to the previous item, methods are required to modify certain variables that are private—modifiable only by the class's own methods.

Making variables private is an important part of writing good programs, and modifying those variables correctly is a significant function of methods. This is covered later in this course.

- Methods make programs more readable and easier to maintain.

It is much easier to figure out what a program is doing by reading the code if there are several different methods with names that match their functions, instead of `main`.

- Methods make development and maintenance quicker.
- Methods are central to reusable software.



Sun Educational Services

Worker Method and Calling Method

- You have been writing programs like this, up to now:

```
public class AllInMain
{
    public static void main (String args[])
    {
        int int1 = 42;
        int int2 = 24;
        System.out.println(42 + 24);
    }
}
```

- This approach is simple, but:
 - ▼ Impractical for larger applications
 - ▼ Does not use OO
 - ▼ Has other disadvantages

Worker Method and Calling Method

Previously, you wrote code like this, with everything in main:

```
public class AllInMain
{
    public static void main (String args[])
    {
        int int1 = 42;
        int int2 = 24;
        System.out.println(int1 + int2);
    }
}
```

This is impractical in larger applications and it does not implement OO, as well as numerous other disadvantages.

The structure of your programs changes when you use methods. There are two basic divisions of the code you write to use methods, each typically in a separate class, in a separate file:

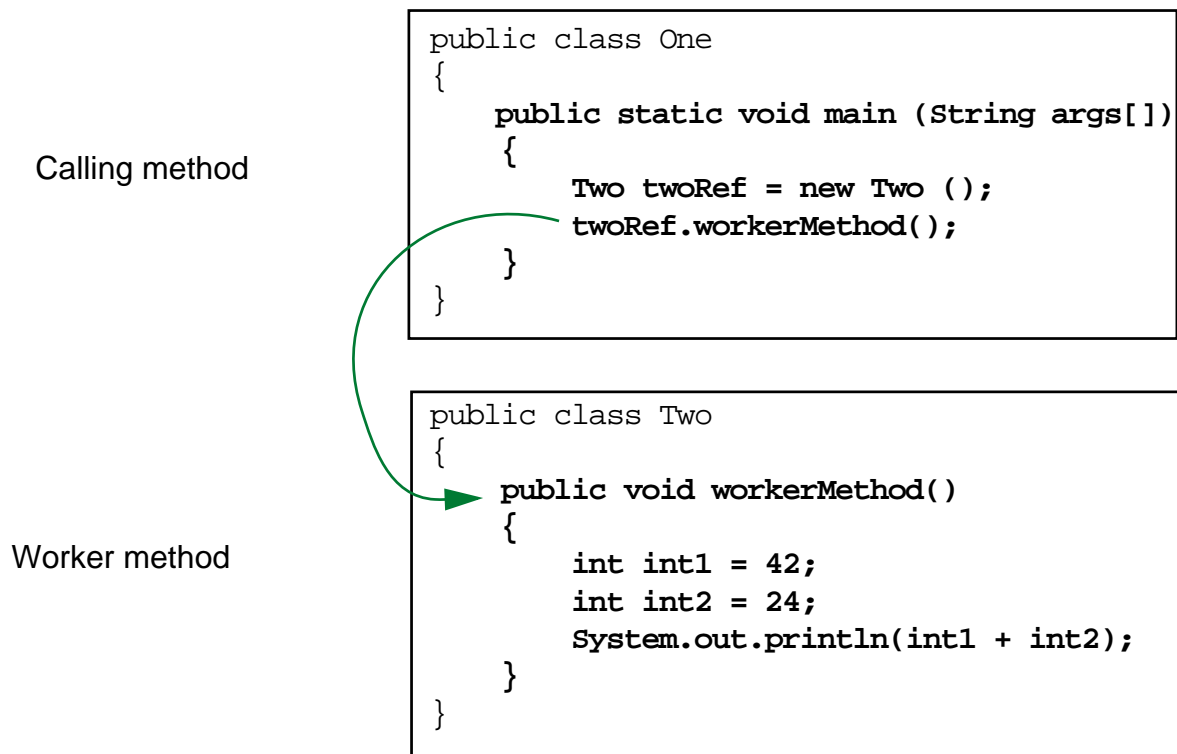


Figure 7-1 Calling Method and Worker Method



Worker Method and Calling Method

- A *calling method* tells an object to do something for you, with one of its methods.
 - ▼ Calling methods often contain normal method processes, in addition to calling the worker method.
- The object contains *worker methods* to do work for you:
 - ▼ Might take input information
 - ▼ Might return one piece of information

Calling Method

You need to write code within a *calling method* that tells an object to do something for you, with one of its methods.

Note – The calling method often includes its own actions in addition to calling the worker method: printing to the screen, performing if/then or looping constructs, and other standard actions that methods perform.

Worker Method

You need a class that has some methods that complete tasks you want done. This is the *worker method*. This method might or might not require additional information when it runs. It also might return one piece of information to the calling method.

They are typically written in two different classes, in two different files. (You can have a calling method and a worker method in the same class, but this occurs less commonly in most applications.) `main` is often used as the calling method.

The following example shows a very basic program, to highlight the separation of the worker and calling methods. The calling method just calls two methods in `WorkerClass`; those two methods print the values of variables set in their methods.

```

1 public class CallingClass
2 {
3
4     // main is the calling method
5     public static void main(String args[])
6     {
7
8         // This line creates an object of the class with worker methods
9         // to specify which object's methods should run, in lines 14-15
10    WorkerClass workerObject = new WorkerClass();
11    // the next two lines use the object to
12    // call worker methods method1 and method2
13    workerObject.worker1();
14    workerObject.worker2();
15    }
16 }

1 // this class contains the worker methods for this example
2 public class WorkerClass
3 {
4     // code block for worker1
5     public void worker1()
6     {
7         int ID = 44559;
8         System.out.println("The ID is " + ID);
9     }
10    // code block for worker2
11    public void worker2()
12    {
13        float price = 29.99F;
14        System.out.println("The price is " + price);
15    }
16 }

```

Notes



Declaring Methods

- Worker and calling methods have the same syntax structure.
- All possible method elements are not included in all examples.

Declaring Methods

You learned how to declare the actions that the `main` method could complete in “Using the main Method” on page 4-25. Syntax and examples are repeated here in order to show expanded information about declaring other methods.

You declare all methods using the same basic structure, regardless of whether they are calling or worker.

Details are included in this syntax section that are covered in subsequent sections in this module. Use the syntax here as a reference once you have covered all aspects of using methods (arguments, return types, and so on.)

Syntax

Methods are written within the code block of the class to which they belong.

You can have zero or more method declarations within a class. It does not matter what order the methods appear within a class; the order in which they run is determined when they are called (calling is explained further in “Calling a Method” on page 7-15).

```
modifiers return_type method_identifier ([arguments])  
{  
    method_body  
}
```

- **modifiers** – Several Java keywords can be used to modify the way methods are stored or how they run. For now, use `public`.
- **return_type** – Some methods are used to calculate and return a value. Methods can return only one value.

If the method returns a value, use the appropriate type such as `int`, `String`, and so on. If no value is returned, use the keyword `void` as the return type.

You must use the `return` keyword to send the return value back to the calling method:

```
return return_value_or_expression  
return (int1 + int2);
```

- **method_identifier** – The identifier is the name of the method that is used to call it.

Follow the rules and guidelines for variable identifiers when you write method identifiers.

- ([**arguments**]) – Arguments are changeable information, provided when you run a method. It is how you allow values to be passed into a method. Arguments are variables used specifically for methods to complete their tasks.

A multiply method, for example, might take two or more arguments: the numbers to multiply together.

Arguments are optional. If there are no arguments, include an empty set of parentheses after the method identifier.

To declare the arguments, list the type first, then the argument identifier. Separate each type/identifier set with commas, if there are two sets or more.

```
method_identifier (type identifier [, type identifier])
```

```
public void computeAge (int myInt);  
public void startOrder (String name, char custType);
```

Arguments are the input to a method, and the return value (whose type you specify with **return_type**) is the output.

- **method_body** – The body of a method is the brace-enclosed code block or blocks that contain the sequence of statements to perform the task. You can include loops, print statements, variable declarations, and many other statements inside the method body.

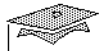
In a calling method, this is where you would call the worker method. (More detail on how to do this is included later.)

The actions within a method declaration are completed in the order they are listed.

If there is a return value, it is sent back to the calling method using the `return` keyword, in the method body. The type of the value returned must match the declared type.

If there is nothing within the method body, put the two braces ({}) at the end of the method line, after the arguments, if any.

Note – If you declare variables inside the braces of the method, you can only use them there. You can use a variable within the set of braces where it was declared, and in any subordinate code blocks.



Calling a Method

- Typically call the method from a different method.
- Calling method pauses and the worker method takes over.
- Calling method resumes at point after it called the worker method.
- The calling method and the worker method can be in the same class or in different classes.
- Call methods in any order.

Calling a Method

Use the information in this section to call methods.

Overview

To make an object run one of its methods, call the method from a different method.

Calling a method means that the calling method pauses and the worker method takes over. When the worker method completes, then the calling method resumes at the point after it called the worker method.

There is no limit to the number of method calls that a method can make.

The calling method and the worker method can be in the same class or in different classes.

From the calling class, you can call methods in any order; they do not need to be completed in the order they are listed in the class where they are declared (the class containing the worker methods).

You can call a method in the same class, or in a different class. The way of calling each is different.

Syntax and Examples

Same Class

Calling a method in the same class is quite simple; write the calling method declaration code and include the name of the worker method and its arguments, if any. (You will learn about passing arguments later.)

Syntax:

```
workermethod ( argument_values );
```

Example:

```
1 public class DisclaimerOneFile
2 {
3     public void callMethod()
4     {
5         // calls the printDisclaimer method
6         printDisclaimer();
7     }
8
9     public void printDisclaimer ()
10    {
11        System.out.println("No instructors were harmed in
the development of this course.");
12    }
13}
```

Different Class

To call a method in a different class, first instantiate an object of the class that the worker class belongs to. (See “Creating Object Reference Variables” on page 4-5.) Then use the reference variable for the object to call the class.

Syntax:

- In one class, you need the following lines in the method code block.

```
WorkerClass workerclass_ref = new WorkerClass();  
workerclass_ref.workermethod([argument_values]);
```

- In the class containing the worker methods, simply declare the method and what it does as you normally would.
- Each class should be in a separate file; you could have WorkerClass.java and CallingClass.java, for example.

Examples:

This example is the same as the example of calling a method from the same class, except that now a `callMethod` method in `DifferentClass` makes the `Disclaimer` object do work for it (call the `printDisclaimer` method).

```

1 public class DifferentClass
2 {
3     public void callMethod()
4     {
5         // makes a Disclaimer object
6         Disclaimer disc = new Disclaimer();
7
8         // calls the printDisclaimer method in the
9         // Disclaimer class
10        disc.printDisclaimer();
11    }
12}

1 public class Disclaimer
2 {
3     public void printDisclaimer ()
4     {
5         System.out.println("No instructors were harmed in
the development of this course.");
6     }
7 }

```

This example shows a Shirt class again, with another class calling its method.

```
1 public class GetInfo
2 {
3     public static void main(String args[])
4     {
5         Shirt shirt1 = new Shirt();
6
7         shirt1.printInfo();
8     }
9 }

1 public class Shirt
2 {
3     int ID = 44339;
4     char size = 'L';
5     public void printInfo ()
6     {
7         System.out.println("ID: " + ID);
8         System.out.println("Size: " + size);
9     }
10 }
```



Passing Arguments

- Include them in the parentheses of the calling method
- Pass literals or variables
- List them in the order stated in the method declaration

Passing Arguments

To pass arguments from one method to another, include them in the parentheses of the calling method. You can pass literal values or variables, but not expressions like `(num1 + num2)`, as arguments. As when you assign literal values to variables, use quotation marks around Strings, use F after a float, and so on.

List the arguments in the same order in which they are listed in the called method's method declaration, and pass all required arguments. The compiler checks to see if the type, order, and number of the parameters match.

Syntax:

As shown in the syntax for calling methods, if there are arguments in the called method, include them in the method call statement in parentheses.

```
workermethod ( argument_values );
```

Examples:

The following examples show the previous disclaimer and shirt programs rewritten with arguments being passed.

This example shows the disclaimer being passed from one class to the other.

```
1 public class DifferentClass2
2 {
3     public void callMethod()
4     {
5         Disclaimer2 disc = new Disclaimer2();
6
7         disc.printDisclaimer("No instructors were harmed in
the development of this course.");
8     }
9 }

1 public class Disclaimer2
2 {
3     public void printDisclaimer (String statement)
4     {
5         System.out.println(statement);
6     }
7 }
```

This example shows a Shirt class again.

```

1 public class GetInfo2
2 {
3     public static void main(String args[])
4     {
5         // makes a Shirt object
6         Shirt2 theShirt = new Shirt2();
7
8         // calls the printDisclaimer method in the
9         // Disclaimer class
10        theShirt.printInfo(44339, 'L');
11    }
12}

1 public class Shirt2
2 {
3     int ID;
4     char size;
5
6     // declare arguments with different names than variables
7     public void printInfo (int shirtID, char shirtSize)
8     {
9         ID = shirtID; // assign arguments to variables
10        size = shirtSize;
11        System.out.println(ID);
12        System.out.println(size);
13    }
14}

```


This example shows numeric values being passed to a method that does a mathematical operation and if construct. Both methods are in the same class.

```

1 public class Arguments
2 {
3     public void passArguments()
4     {
5         subtract( 3.14159F, 9F );
6     }
7
8     public void subtract( float first, float second
9     )
10    {
11        if ((first - second) >= 0)
12        {
13            System.out.println("Positive");
14        }
15        else
16        {
17            System.out.println("Negative");
18            System.out.println("Try again");
19        }
20    }

```

The diagram illustrates the flow of arguments between two methods in the same class. A blue arrow points from the opening curly brace of the `passArguments()` method (line 4) to the opening curly brace of the `subtract()` method (line 8). Two pink arrows originate from the arguments `3.14159F` and `9F` in the `passArguments()` call (line 5). One pink arrow points from `3.14159F` to the `float first` parameter of the `subtract()` method (line 8). The other pink arrow points from `9F` to the `float second` parameter of the `subtract()` method (line 8). The parameters `float first` and `float second` are enclosed in pink rounded rectangles.

Figure 7-2 Passing Arguments



Receiving Return Values

- The return value is returned to the same place in your code where you called it from:

```
mathRef.add(2, 4);
```

- The expression in which you called the method is equal to the return value.
- You can combine the call and the use of the return value in one line, when using the return value.

```
int addedNumbers = mathRef.add(2, 4);
```

```
if ( add(2, 4) > currentDate);
```

Receiving Return Values

If you call a method that returns a value, such as an add method that adds two numbers and returns the result, you can use the return value in the calling method.

The following example shows the basics of receiving a return value.

```
1 public class ReceivesValues
2 {
3     public static void main (String args[])
4     {
5         AddsValues adder = new AddsValues();
6         int sum = adder.returnSum();
7
8         System.out.println("The value returned is " + sum);
9     }
10 }

1 public class AddsValues
2 {
3     public int returnSum()
4     {
5         int int1 = 4;
6         int int2 = 17;
7         return (int1 + int2);
8     }
9 }
```

The return value is tracked in memory and returned to the same place in your code where you called it from—the line where you specified the method name (and arguments, if any).

The method is run and the expression in which you called the method is equal to the return value.

This example shows the Shirt program again, this time receiving a return value. In addition to passing the ID and size, it also passes the number in stock and the price.

```

1 public class GetInfo3
2 {
3     public static void main(String args[])
4     {
5         float invValue;
6         Shirt3 theShirt = new Shirt3();
7
8         invValue = theShirt.printInfo(44339, 'L', 29.99F, 340);
9         System.out.println("Value of inventory: " + invValue);
10    }
11 }

1 public class Shirt3
2 {
3     int ID;
4     char size;
5     int inStock;
6     float price;
7     public float printInfo (int shirtID, char shirtSize, float
shirtPrice, int shirtInStock)
8     {
9         ID = shirtID;
10        size = shirtSize;
11        inStock = shirtInStock;
12        price = shirtPrice;
13        System.out.println("ID: " + ID);
14        System.out.println("Size: " + size);
15        System.out.println("Standard price: " + price);
16        System.out.println("Number in stock: " + inStock);
17        if (ID <= 9999)
18        {
19            ID = ID + 50000; // updates 4-digit format IDs to 5-digit
20            System.out.println("Updated ID: " + ID);
21        }
22        if ((ID > 55524) && (ID < 51000)) // puts items on sale
23        {
24            price = (float)(price * .75);
25            System.out.println("Sale price: " + price);
26        }
27        return (inStock * price);
28    }
29 }

```

As shown in the `GetInfo` class on line 8, when you want to use the return value in code in the calling method, you can combine the call and the use of the return value in one line.

Exercise 1: Using Methods



Exercise objective – Practice using methods and using two files to create and run programs.

Tasks

Follow the directions from your instructor to locate the exercises directory for this module.

For each of the following, make two files: one with the given name, and another with the same name, but with `Test`, `Create`, or another word added, that contains `main` and accesses the other class. For example, make files called `StoreVal` containing the class, and `StoreValTest.java` containing `main`.

1. Write a program called `StoreValTest` that implements the `StoreVal` class from this module. You will have to type in `StoreVal` as well. (Create two files, `StoreVal.java` and `StoreValTest.java`.)
2. Declare a `Shirt` class that has a price, item ID, and type (such as Oxford or polo). Declare methods that return those three values. (These are `get` methods.) Write another class that calls and prints those values. (You will need to create two files, one called `Shirt.java` that declares the shirt variables and methods, and one called `CreateShirt.java` that calls the methods and prints the values.)
3. Write new versions of the programs in the previous exercise; call them `Shirt2.java` and `CreateShirt2.java`. In `Shirt2.java`, add methods that set the values for the three variables and take arguments as the values. In `CreateShirt2.java`, call those methods, passing the appropriate values.



Object Methods and Static Methods

- Object methods need an object to be created.
- Static methods can be run without an object.

Object Methods and Static Methods

So far, you have learned how to call methods by creating an object of the class the method belongs to, then getting that object to run the method. This is an *object method*; you need to create an object in order to make the method run.

You have also been using methods that do not require an object instantiation, such as `main` and `Integer.parseInt`. These are static methods; you can ask them to do work for you without creating an object first. Sometimes you need methods that are related to the class, but do not operate on the variables associated with a particular object.

Object Methods

This program containing a standard object method, `setInfo` in the `Shirt` class, needs an object to run.

```

1 public class SendInfo
2 {
3     public static void main(String args[])
4     {
5         Shirt4 theShirt = new Shirt4();
6         theShirt.setInfo('L', 29.99F, 44990, 12);
7     }
8 }

1 public class Shirt4
2 {
3     float price;
4     int ID;
5     char size;
6     int inStock;
7     public void setInfo (char shirtSize, float shirtPrice, int
shirtID, int numInStock)
8     {
9         price = shirtPrice;
10        ID = shirtID;
11        size = shirtSize;
12        inStock = numInStock;
13        System.out.println(price + ", " + shirtID + ", " + size + ", "
+ inStock);
14    }
15 }

```


Static Methods

A static method performs its function without regard to a particular object. You could make the `setInfo` method static if you wanted the `Shirt` class attributes to have the same information, regardless of which shirt it is (this would be useful for setting specific default values).

Syntax for calling static methods:

```
method_class.method( )
```

Examples:

Calling a static method, one line:

```
DayOfMonth.returnDay( )
```

This is an example of a method that would be useful if you have a numerical size of the shirt and you want to know if it is a small, medium, large, or extra large. This is a static method because:

- It does not directly use any attributes of the class.
- You might want to be able to call this generic utility even if you do not have a shirt object.

```
1 public class StaticShirt
2 {
3     float price;
4     int ID;
5     char size;
6     public static char convertShirtSize(int numericalSize)
7     {
8         if (numericalSize < 16)
9         {
10             return 'S';
11         }
12         else if (numericalSize < 20)
13         {
14             return 'M';
15         }
16         else if (numericalSize < 24)
17         {
18             return 'L';
19         }
20         else
21         {
22             return 'X';
23         }
24     }
25 }
```

Static Methods in the Java API

An example of a common static method is `Math.random()`; most of the `Math` methods included with the Java API are static. The following example calls `Math.random()`.

```
1 public class Get
2 {
3     public static void main (String args[])
4     {
5         StaticExample ex = new StaticExample();
6         ex.getNumber();
7     }
8 }

1 public class StaticExample
2 {
3     public void getNumber ()
4     {
5         System.out.println("A random number: " + Math.random());
6     }
7 }
8 }
```



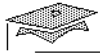
When to Declare a Static Method

- Define a method as static when:
 - ▼ It is not important which individual object you perform the operation on.
 - ▼ It is important that the method's actions be run before instantiating objects.
 - ▼ The responsibility that the method fulfills does not logically belong to an object.

When to Declare a Static Method

Define a method as static when:

- It is not important which individual object you perform the operation on. The result of $7*10$, or the current temperature, or the time, are the same whether the result is assigned to `myObject1` or `myObject2`.
- It is important that the method's actions be run before instantiating objects.
- The responsibility that the method fulfills does not logically belong to an object.



Method Overloading

- Several methods that do the same job but with different parameters
- Can have same name if *signature* is unique

```
public int printName ([String name])  
{  
    // method_body  
}
```

Method Overloading

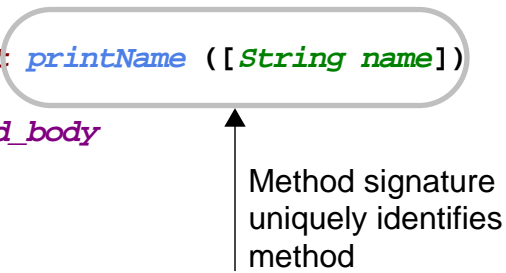
It is often necessary to have several methods which all do the same basic job but which accept different arguments. The `returnSum` method shown on page 7-25 takes two `ints` as arguments. But if the values had been floats, such as 9.3, the method would not work.

In the Java programming language, there can be several methods in a class that have the same name but different arguments. This is *method overloading*.

The methods can have the same name as long as they have a different argument list. Just as you can distinguish between two students in the same class named Chris by calling them “Chris in the green t-shirt” and “Chris with the beeper,” you can distinguish between two methods by their name and their arguments.

The combination of name and argument list is known as the *method signature*, shown in Figure 7-3:

```
public int printName ([String name])
{
    method_body
}
```



Method signature uniquely identifies method

Figure 7-3 Method Signature

Many methods in the Java API are overloaded, including one you have been using throughout this course, `System.out.println`. You can print Strings, ints, floats, and so on using this method, because it is overloaded with different arguments, for different data types.

The specification for this method, within the `PrintStream` class, is shown in Figure 7-4 on page 7-37.

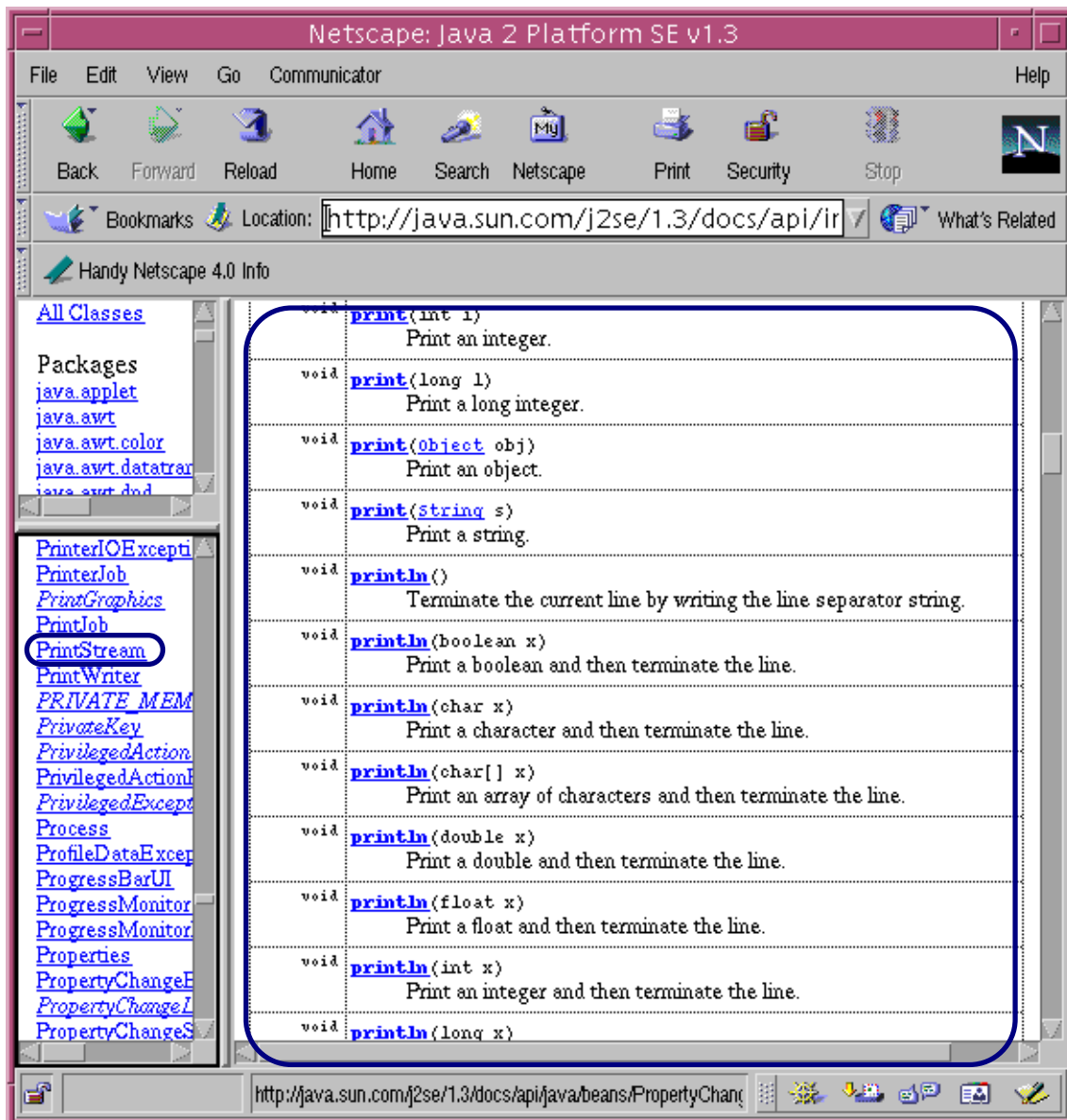


Figure 7-4 Java API Specification: `print` and `println`, Overloaded Methods

Without overloading, you would need to create multiple methods with different names, just to print different data types—`printlnint`, `printlnfloat`, and so on—or else do a lot of conversion before using the method.

When you write code, keep in mind that you need to define overloaded methods if the actions the method completes will need to be performed on several different types of data. You can also use overloading for other purposes; to take multiple arguments, for instance, if you want to multiply two numbers, three numbers, or four numbers together:

```
multiply(int one, int two);  
multiply(int one, int two, int three);  
multiply(int one, int two, int three, int four);
```

The example in Figure 7-5 on page 7-39 shows the overloaded `getValuePlus` method, which can use either an `int` argument called `more` or a `double` argument called `more`.


```
1 public class Example
2 {
3     public static void main (String args[])
4     {
5         StoreVal store = new StoreVal();
6
7
8         store.setValue(9);
9
10
11         int i = store.getValuePlus(2);
12
13
14         double d = store.getValuePlus(3.6);
15     }
16 }

1 public class StoreVal
2 {
3     int value;
4     void setValue(int val)
5     {
6         value = val;
7     }
8     int getValuePlus(int more)
9     {
10         return (value + more);
11     }
12     double getValuePlus(double more)
13     {
14         return (value + more);
15     }
16 }
```

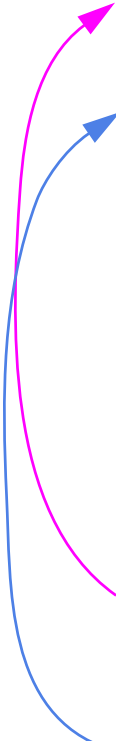


Figure 7-5 Using Overloaded Methods

Exercise 2: Using Overloaded Methods

Exercise objective – Practice using overloaded methods.



Tasks

Add overloaded methods to `Shirt2.java` (call it `Shirt3.java`) that print the ID, the ID and the price, or the ID, the price, and the type. Create a new file to test the methods called `CreateShirt3.java`.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Describe why using multiple methods supports object orientation
- ☐ Write a method declaration for a method other than `main`
- ☐ Write a method that calls another method
- ☐ Pass parameters to a method and receive a return value
- ☐ Define static and object methods and state when to use each
- ☐ Explain what overloaded methods are and why they are useful
- ☐ Write code using overloaded methods

Think Beyond

Throughout this course, you have learned at various stages about private members, encapsulation, obtaining and setting data through methods instead of manipulating it directly, and other object-oriented principles. Discuss the following:

- Definition of encapsulation
- Advantages of encapsulation
- What can happen if all variables are public

Objectives

Upon completion of this module, you should be able to:

- Define encapsulation and its advantages
- Define the modifiers `public` and `private` and use them in programs
- Describe the rules for variable scope
- Write code that encapsulates private members
- Explain the purpose of constructors
- Write code to use the default constructor
- Write code to define a new constructor and use it

This module describes how to bring together the OO and Java programming language concepts presented so far in this course to write encapsulated, object-oriented programs using good OO design and implementation.

Relevance



Discussion – If you drove a car to class today, consider what would happen if you had to directly manipulate the gears, fuel injection, exhaust system, and other car parts, rather than just using the steering wheel, key, and gear shift.



Sun Educational Services

Encapsulation

- Operations and attributes are its *members*.
- Members can be *public* or *private*.
- Most or all variables should be kept private.
- Variables are modified by methods of their own class.
- Other classes *interface* with only a few parts of every other class.
- Implementation can change without changing the interface.

Encapsulation

You learned earlier in this course in Module 2, “Object-Oriented Analysis and Design,” about the importance of encapsulation to object-oriented programs.

This section briefly reviews the concepts of encapsulation, then describes how to write encapsulated Java programs.

Private Members

An object’s operations and attributes are together referred to as its *members*. The members of an object can be *public* or *private*.

In an ideal program, most or all of the variables of a class are kept private. This means that they cannot be modified or viewed directly by classes outside their own class; they can only be modified or viewed by methods of that class, which have built-in checks and business logic to make sure inappropriate values are not assigned to the variables.

Private Implementation, Public Interface

When you encapsulate programs this way, other classes *interface* with only a few parts of every other class. This reduces the dependence of one class on another to stay consistent. For example, the Printing class can change how it accesses drivers and networks as much as the programmer wants, but if the name and arguments of its `print` method stay the same, other code does not have to change.

The way that other classes interact with a class is called the *interface*; the way that a class completes its tasks is called the *implementation*. In a good object-oriented program, the implementation can change without affecting the interface.

- If a variable or method is defined as public, then it is part of the interface and other classes can access it.
- If a variable or method is defined as private, then it is part of the implementation and only the class's own members can access it.

Encapsulation Examples

Elevator: The Wrong Way

The following example shows what not to do, when writing the program to control an elevator.

```
1 public class BadElevator
2 {
3     public boolean doorOpen=false;
4     public int floor = 1;
5
6     public final int TOP_FLOOR = 5;
7     public final int BOTTOM_FLOOR = 1;
8 }

1 //Example of how somebody might use the elevator class:
2 public class BadTestElevator
3 {
4     public static void main(String args[])
5     {
6         BadElevator el = new BadElevator();
7         el.doorOpen = true; //passengers get on
8         el.doorOpen = false; //doors close
9         el.floor--;          //go down to floor 0
10        //((below bottom of building)
11        el.floor++;
12        el.floor = 7;        //jump to floor 7 - (only 5 floors in building)
13        el.doorOpen = true; //passengers get on/off
14        el.doorOpen = false;
15        el.floor = 1;        //plummet to the first floor
16        el.doorOpen = true; //passengers get on/off
17        el.floor++;          //elevator moves with door open
18        el.doorOpen = false;
19        el.floor--;
20        el.floor--;
21    }
22 }

23 //All of the problems would be impossible to cause if program had proper
24 //encapsulation and error checking in the elevator
```

Elevator: The Right Way

The following example shows how to encapsulate parts of the program, to prevent unnecessary deaths.

```

1 public class Elevator
2 {
3     private boolean doorOpen=false;
4     private int floor = 1;
5     private int weight = 0;
6
7     final int CAPACITY = 1000;
8     final int TOP_FLOOR = 5;
9     final int BOTTOM_FLOOR = 1;
10    final String MUSIC = "Lyle Lovett";
11
12    //The next two methods are the equivalent of a "setDoorOpen" method,
13    //but in the case of an elevator openDoor and closeDoor seem like
14    //more intuitive method names
15    public void openDoor()
16    {
17        doorOpen = true;
18    }
19
20    public void closeDoor()
21    {
22        checkWeightSensors();
23
24        if (weight <= CAPACITY)
25        {
26            doorOpen = false;
27        }
28        else
29            //The computer beeps 10 times
30            {
31                for (int counter=0; counter<10; counter++)
32                    System.out.print((char) 7);
33            }
34    }
35
36    //In reality the elevator would have weight sensors to
37    //check the actual weight in the elevator, but for the sake
38    //of simplicity we just pick a random number to represent the
39    //weight in the elevator
40    private void checkWeightSensors()
41    {

```

```
42     weight = (int) (Math.random() * 1500);
43     System.out.println("The weight is " + weight);
44 }
45
46 public void goUp()
47 {
48     if (!doorOpen)
49     {
50         if (floor < TOP_FLOOR)
51         {
52             floor++;
53             System.out.println(floor);
54         }
55         else
56         {
57             System.out.println("Already on top floor.");
58         }
59     }
60     else
61     {
62         System.out.println("Doors still open!");
63     }
64 }
65
66 public void goDown()
67 {
68     if (!doorOpen)
69     {
70         if (floor > BOTTOM_FLOOR)
71         {
72             floor--;
73             System.out.println(floor);
74         }
75         else
76         {
77             System.out.println("Already on bottom floor.");
78         }
79     }
80     else
81     {
82         System.out.println("Doors still open!");
83     }
84 }
85
86 public void setFloor(int goal)
87 {
```

```
88     while (floor != goal)
89         if (floor < goal)
90             {
91                 goUp();
92             }
93         else
94             {
95                 goDown();
96             }
97     }
98
99     public int getFloor()
100     {
101         return floor;
102     }
103
104     // Equivalent of "getOpen" except "is" sounds better than
105     // "get" with booleans
106     public boolean isOpen()
107     {
108         return doorOpen;
109     }
110     public String playMusic()
111     {
112         System.out.println("Mmm, the soothing strains of " + MUSIC);
113         return MUSIC;
114     }
115 }
```

```
1 public class TestElevator
2 {
3     public static void main(String args[])
4     {
5         Elevator el = new Elevator();
6         el.openDoor();
7         el.closeDoor();
8         el.goDown();
9         el.goUp();
10        el.goUp();
11        el.openDoor();
12        el.closeDoor();
13        el.goDown();
14        el.openDoor();
15        el.goDown();
16        el.closeDoor();
17        el.goDown();
18        el.goDown();
19        el.playMusic();
20
21        int curFloor = el.getFloor();
22        if (curFloor != 5 && ! el.isOpen())
23        {
24            el.setFloor(5);
25        }
26        el.openDoor();
27        el.closeDoor();
28        if (el.getFloor() != 3 && ! el.isOpen())
29        {
30            el.setFloor(3);
31        }
32    }
33 }
```

Time Program

This example shows how to use encapsulation to check for valid data and other elements in a time program.

```

1 public class Time implements java.io.Serializable
2 {
3     private int second, minute, hour;
4
5     public Time(int s, int m, int h)
6     {
7         second = s;
8         minute = m;
9         hour = h;
10    }
11
12    public Time()
13    {
14    }
15
16    public void setSecond(int s)
17    {
18        if (s >= 0 && s < 60)
19            second = s;
20        else
21            System.out.println("Invalid seconds value, not set");
22    }
23
24    public void setMinute(int m)
25    {
26        if (m >= 0 && m < 60)
27            minute = m;
28        else
29            System.out.println("Invalid minutes value, not set");
30    }
31
32    public void setHour(int h)
33    {
34        if (h >= 0 && h < 24)
35            hour = h;
36        else
37            System.out.println("Invalid hours value, not set");
38    }

```



```
39 public void setTime(int s, int m, int h)
40 {
41     setSecond(s);
42     setMinute(m);
43     setHour(h);
44 }
45
46 public void tick()
47 {
48     second++;
49
50     if (second == 60)
51     {
52         second = 0;
53         minute++;
54
55         if (minute == 60)
56         {
57             minute = 0;
58             hour++;
59
60             if (hour == 24)
61                 hour = 0;
62         }
63     }
64 }
65
66 public int getSecond()
67 {
68     return second;
69 }
70
71 public int getMinute()
72 {
73     return minute;
74 }
75
76 public int getHour()
77 {
78     return hour;
79 }
80
```

```
81 public String toString()
82 {
83     return hour + ":" + minute + ":" + second;
84 }
85
86 public boolean equals(Time compareMe)
87 {
88     if (second == compareMe.second && minute == compareMe.minute &&
hour == compareMe.hour)
89         return true;
90     else
91         return false;
92 }
93
94 public boolean withinAnHour(Time compareMe)
95 {
96     if ((hour == compareMe.hour - 1 && minute >= compareMe.minute)
97         || (hour == compareMe.hour      && minute <= compareMe.minute))
98         return true;
99     else
100         return false;
101 }
102 }
```



Sun Educational Services

Implementing Encapsulation

- Put `public` or `private` in front of members

```
private int myInt;  
public String name;  
  
public void getName()  
{  
    return name;  
}
```

Implementing Encapsulation

In pure object-oriented systems, all attributes are private and can be changed or accessed only through operations in the public interface. Public methods include those used to get and set attributes in the implementation.

Syntax

The Java programming language has a series of modifiers that can be used to restrict access to a member in a variety of ways, ranging from making a member accessible to everyone to locking the member completely so that no one except the object itself can even read the data. This section will cover the two primary modifiers, `public` and `private`.

To make a variable or method `private` or `public`, just put the `private` or `public` modifier in front of it.

```
private int myInt;  
public String name;  
  
public void getName()  
{  
    return name;  
}
```

Examples

These examples show what happens when you try to access variables and methods designated as `public` or `private`.

public *Modifier*

You can put the `public` modifier in front of a member variable or method to mean that code in any other class can use that part of the object. Consider the following example. All members of `PublicClass` are public, so other classes can easily modify variable values and call methods.

```
1 public class PublicExample
2 {
3     public static void main (String args[])
4     {
5         PublicClass pc = new PublicClass();
6         pc.publicInt = 27;
7         pc.publicMethod();
8     }
9 }

1 public class PublicClass
2 {
3     public int publicInt;
4     public void publicMethod()
5     {
6         System.out.println(publicInt);
7     }
8 }
```




Figure 8-1 Modifying Public Members of a Class

This code will compile and run because the `publicInt` and `publicMethod` methods are public.

private *Modifier*

Put the `private` modifier in front of a member variable or method if you do not want any classes outside the object's class to use that part of an object. Consider the following example. The member variables and methods are protected from other classes by the `private` modifier.

```

1 public class PrivateExample
2 {
3     public static void main (String args[])
4     {
5         PrivateClass pc = new PrivateClass();
6         pc.privateInt = 27;
7         pc.privateMethod();
8     }
9 }

1 class PrivateClass
2 {
3     private int privateInt;
4     private void privateMethod()
5     {
6         System.out.println(publicInt);
7     }
8 }

```

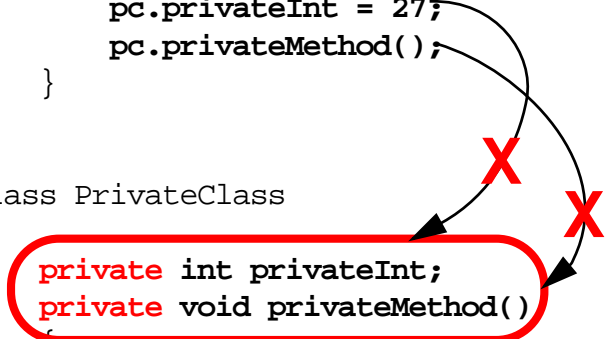
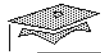


Figure 8-2 Failed Attempt to Access Private Members

This code will not compile. The `main()` method is attempting to change the value of a private member and invoke a private method.

Typical Implementation

The most typical approach is to make variables private but methods public, so that the variables can be modified by the methods. This is discussed in the next section, “get and set Methods.”



Sun Educational Services

get Methods and set Methods

- When variables are private, they must be accessed by member methods.
- `get` and `set` methods obtain and assign values.

get and set Methods

If you make variables `private`, how can someone access them? The answer is to provide a `public` method for each of the ways that you expect people to use the variables, such as setting the value and getting the current value.

Most variables are private, so *get methods* and *set methods* are part of most classes. `get` methods obtain the value of variables and `set` methods assign them values. If you define an `Item` class, for instance, you need methods like `getPrice` and `setPrice` for the price variable; `getSize` and `setSize` for the size variable; and so on.

The following examples demonstrate a set method and get method.

```

1 public class EncapsulatedItem
2 {
3     private int itemID;
4     public void setItemID(int newValue)
5     {
6         itemID = newValue;
7     }
8     public int getItemID()
9     {
10        return itemID;
11    }
12 }

```

While the code shown is syntactically correct and does satisfy the principles of encapsulation, it does not go far enough. The methods still provide unrestricted access to the `itemID` variable; there is no built-in logic to make sure that the correct values are set, and so on.

You should add extra lines of business logic to validate the action being taken. For example, if the valid range for the `itemID` variable is 1 to 10000, the `setItemID()` method would be:

```

1 public class EncapsulatedItem2
2 {
3     private int itemNumber;
4     public void setItemNumber(int newValue)
5     {
6         if ((newValue >= 1) && (newValue <= 10000))
7         {
8             itemNumber = newValue;
9         }
10    }
11    public int getItemNumber()
12    {
13        return itemNumber;
14    }
15 }

```


Exercise 1: Writing Encapsulated Applications



Exercise objective – Practice implementing encapsulation.

Tasks

Follow the directions from your instructor to locate the exercises directory for this module.

1. Make a class called `DateOne` to represent a date. Make the appropriate attributes (class variables.) Do not put any methods in yet. For now, do not make the variables private.

In `DateOneTest.java`, make two objects of the date class. Put values into each of the attributes of each of the objects. Print out each of the date objects.

2. Make a copy of `DateOne` and call it `DateTwo`. Make the variables in the class private. Make a copy of `TestDateOne` and call it `TestDateTwo`; update the references so that they now refer to the `DateTwo` class. Recompile the program. Does it work?
3. In a class called `DateThree`, add the appropriate get and set methods for each attribute of the class.

Make three more methods:

- ▼ `daysInMonth` – Takes a parameter that is a month number and returns the number of days in that month.
- ▼ `setDate` – Takes parameters for each attribute of the date, verifies it, and assigns it. While verifying the date, make sure that the day is acceptable for that particular month. For example, February 31 would be invalid.
- ▼ `print` – Prints the date.

Make a `TestDateThree.java` file, as well, updating references. Test your `Date` class in the main of the other class.

4. Make a class called `Rectangle` that represents a rectangle using private `width` and `height` variables. Make the following public methods:
 - ▼ `getHeight` returns the height of the rectangle
 - ▼ `getWidth` returns the width of the rectangle
 - ▼ `setHeight` verifies the data and assigns the new value to the `height`
 - ▼ `setWidth` verifies the data and assigns the new value to the `width`
 - ▼ `getArea` returns the area of the rectangle
 - ▼ `getPerimeter` returns the perimeter of the rectangle
 - ▼ `draw` draws the rectangle using `*`s.
 - ▼ Write the main method in another class `TestRectangle` to test the `Rectangle` class (call the methods, and so on).



Sun Educational Services

main Method Placement

- For any program that you want to run, the class in the file you run must contain a `main` method.
- If not, the following message will appear:

```
In class class: void main(String argv[]) is not defined
```

- If you call parts of the class from another class that uses a `main` method, the program will run correctly.

main Method Placement

You can write classes that do not contain the `main` method, and they will compile correctly. However, for any program that you want to run, the class in the file you run must contain a `main` method. If it does not, a message similar to `In class class: void main(String argv[]) is not defined` would appear, and the program would not run.

If you called parts of the class from another class that used a `main` method, the program would run correctly.



Variable Scope

- All variables are not available throughout a program.
- *Variable scope* means where a variable can be used.

Variable Scope

This has been touched on in a previous module; however, it is particularly relevant now that you are writing complete object-oriented programs. *Variable scope* means where in a program a variable can be used.

Some variables are not available for use throughout a class, even if they are public. They are available only in the closest set of braces, and in any subordinate code blocks.

The following example shows the scope for the variables age and name.

```
1 public class NameAge
2
3 { // begin scope of int age
4     private int age = 32;
5     public void print ()
6
7     { // begin scope of String name
8         String name = "Derek Smalls";
9         System.out.println("My name is " + name + " and I am " + age + "
years old.");
10    } // end scope of String name
11
12    public String stateName ()
13    {
14        return name; // this would cause
15                      // an error
16    }
17 } // end scope of int age
```

It is important to keep variable scope in mind at all times, but particularly when using loops and if statements. Variables declared in loops or if statements only exist in that block.



Sun Educational Services

Constructors

- Constructors allow you to specify values for objects when you create them.
- You have been using a default constructor throughout most of the course.
- A constructor has the same name as its class.

Constructors

Constructors allow you to specify values for objects when you create them. Constructors allow you to instantiate objects (with the new modifier) and to specify, when you instantiate an object, what variables are initialized and the values they have.

Example

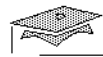
Assume that a constructor was defined for the `Hat` class. The constructor declaration is similar to a method declaration.

```
1 public class Hat
2 {
3     private String type;
4     public Hat(String hatType)
5     {
6         type = hatType;
7     }
8 }
```

Now, when you write code to place an order or do anything else requiring you to create a hat, you must instantiate the `Hat` object and specify its type in one step:

```
1 class Order
2 {
3     Hat hat1 = new Hat("Fedora");
4 }
5
```

A constructor has the same name as the class it belongs to: For example, the `Order` class's constructors are all named `Order`. You can have multiple constructors in a class if each one has a different signature.



Sun Educational Services

How Constructors Are Made Available

- The default constructor
- Constructors that you explicitly create

How Constructors Are Made Available

Constructors are made available in two different ways:

- The default constructor is automatically a part of any class. This is the type of constructor that was called in the previous modules when you used the new modifier.
- You can explicitly create constructors. If you do so, the default constructor is no longer available.



The Default Constructor

- If a class has no constructor, a default constructor is inserted.
- When you use `new` to instantiate an object, `new` automatically calls the class's default constructor.

```
Shirt shirt1 = new Shirt();
```

- The compiler will insert the default constructor.

The Default Constructor

All classes must have at least one constructor. If the Java programming language compiler finds a class that does not have a constructor it will insert one for you, called the *default constructor*.

It does not actually allow you to specify any initialization values for any object variables; it is only created to fulfill the requirements of the compiler.

You have been using the default constructor for several modules. When you use the `new` modifier to instantiate an object, `new` automatically calls the class's default constructor.

```
Shirt shirt1 = new Shirt();
```

`Shirt()` is the default constructor for the `Shirt` class.

Consider the following Shirt class:

```
1 public class Shirt
2 {
3     String name;
4 }
```

The compiler will insert the default constructor, *as if* the original class had been as shown:

```
1 public class Shirt
2 {
3     String name;
4
5     // the function of the default constructor
6     // (does not appear in code)
7     public Shirt()
8     {
9     }
10 }
```

The compiler inserts the default constructor only when you do not provide one of your own. If you do write a constructor, the compiler will not provide the default one.

Note – A common mistake is to create a constructor, then attempt to use the default constructor. Declaring your own constructor or constructors prevents the compiler from inserting the default constructor in your code.



Sun Educational Services

What Constructors Can Do

- Very flexible
- Can set values, pass arguments, and so on
- Example: create `Customer` object whose name is `New Customer` if no name is specified

Declaring and Using Constructors

Use this section to declare and use constructors in your programs.

What Constructors Can Do

You can define anything in the statements of the constructor code block. Typically, it is to set a certain variable to a specific value, or to specify arguments that you can pass values to when creating the object.

For example, you could define a `Customer()` constructor to always set the customer name to `New Customer` if you do not want to allow a blank name. You could declare a different constructor with the arguments `firstName` and `lastName`, to be passed when the customer record is created.

Declaring and Using a Constructor

The only requirement in a declaration is to include the constructor name and braces within the class code block. You can choose to include arguments, a statement within the code block, and so on.

Syntax of a Constructor Declaration:

```
modifiers class ClassName
{
    ClassName([arguments])
    {
        [variable_initialization_statements];
    }
}
```

Specify the arguments using the *type identifier* format, as with methods, such as (String hatType).

Syntax for Constructor Use:

```
modifiers class Class
{
    ClassName object_ref = new ClassName([argument_values]);
}
```

If the constructor declaration includes arguments, you must provide values for all of them when you use the constructor.

Examples

The following declaration and constructor call define a constructor `Shirt(String type, char size)` that takes two arguments, and create a shirt with the object reference `myShirt`, with `type` and `size` values passed from the calling class, `Order`.

```
1 //Declaration of constructor with arguments:
2 public class Shirt
3 {
4     private String type;
5     private char size;
6     public Shirt(String theType, char theSize)
7     {
8         type = theType;
9         size = theSize;
10    }
11 }

1 //Use:
2 public class MakeOrder
3 {
4     Shirt myShirt = new Shirt("Polo", 'L');
5 }
```

The following use would cause an error:

```
1 //Use:
2 public class MakeOrder
3 {
4     Shirt myShirt = new Shirt();
5 }
```

The following declaration and constructor call define and use a constructor that specifies that the item is an oxford shirt. Every time you call this `Shirt()` constructor, a shirt is created with the type `Oxford Shirt`.

```
1 //Declaration of constructor with no arguments:
2 public class OxfordShirt
3 {
4     String type;
5     char size;
6     public OxfordShirt()
7     {
8         type = "Oxford Shirt";
9     }
10 }

1 //Use:
2 public class OrderShirt
3 {
4     Shirt myShirt = new Shirt();
5 }
6
```

Overloading Constructors

Like methods, constructors can be overloaded. This offers a variety of ways that objects can be created in a single class. In the following example, there are two constructors, `Shirts()` and `Shirts(String shirtType)`.

```
1 public class Shirts
2 {
3     String type;
4     char size;
5
6     // Shirt() is the constructor that takes no arguments
7     Shirts()
8     {
9         type = "Oxford Shirt";
10    }
11
12    // Shirts(String type) is another constructor, which takes
13    // one argument
14    Shirts(String shirtType)
15    {
16        type = shirtType;
17    }
18 }
```

You can now instantiate objects of the `Shirts` class using either of two constructors.

Exercise 2: Using Constructors

Exercise objective – Practice creating and using constructors.



Tasks

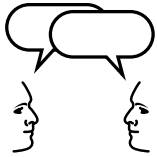
1. Write a program called `Customer.java`. Declare a `Customer` class with variables for a salutation (such as Ms.), first name, middle name, last name, and address, with three constructors:
 - ▼ One creates a new customer with no values
 - ▼ One takes only a first name and a last name
 - ▼ One takes a salutation (such as Ms.), first name, middle name, and last name

Test the program with a `CustomerTest.java` program.

2. Write two additional constructors that take an address with or without a country being specified.

Test the program.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Define encapsulation and its advantages
- ☐ Define the modifiers `public` and `private` and use them in programs
- ☐ Describe the rules for variable scope
- ☐ Write code that encapsulates private members
- ☐ Define static and object methods and state when to use each
- ☐ Explain the purpose of constructors
- ☐ Write code to use the default constructor
- ☐ Write code to define a new constructor and use it

Think Beyond

You are now familiar with defining variables of different data types to represent and manipulate objects in the real world. What would you do to manage or manipulate large groups of items of the same type (tens, hundreds, or thousands of items)?

Objectives

Upon completion of this module, you should be able to:

- Describe an array
- Explain why arrays are useful
- Write code to use arrays for storing primitive values
- Write code to use arrays for storing references to objects
- Write code to use one- and two-dimensional arrays

This module describes how to use arrays to manage multiple values in the same variable.

Relevance



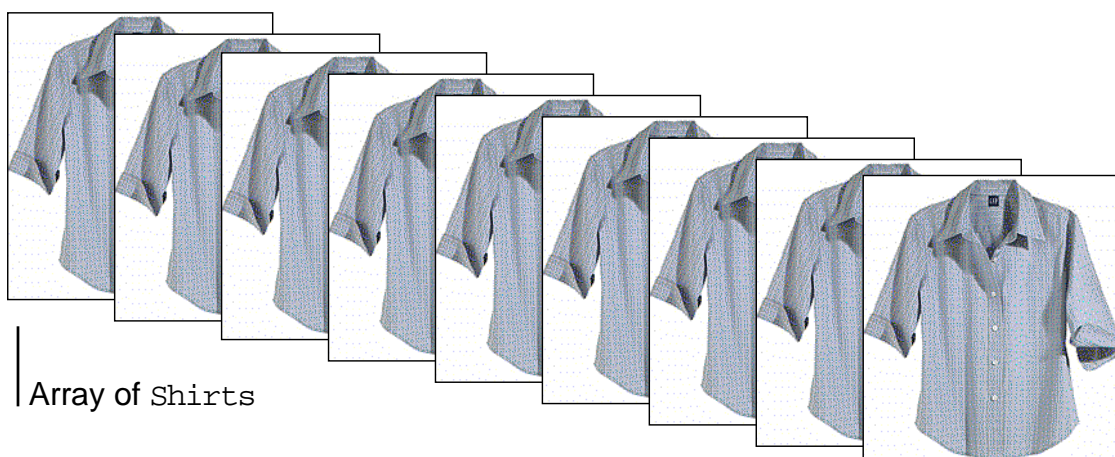
Discussion – What do you do if you have a group (for example, 25) of items of the same type? Do you create 25 different variables for each item? What if you have 100 items?

Array Overview

Arrays let you easily create many variables, all of exactly the same type, without needing to identify each one uniquely. Examples are shown in Figure 9-1.

int	int	int	int	int	int	int	int	int	int	int	int
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Array of ints



Array of Shirts

Array of Strings	—	Nigel Tufnel
		David St. Hubbins
		Derek Small
		Viv Savage
		Ian Faith
		Tommy Pischedda
		Ronnie Pudding
		Mick Shrimpton
		Marty DiBergi

Figure 9-1 Examples of Items Created Using Arrays

This section illustrates why this is necessary in your programs.

The Problem: Creating Many Variables of the Same Type

There are disadvantages to using standard variables for a large number of values.

Example

In previous modules, if you wanted to create the numbers from 1 to 10, such as the ages of ten people in a medical insurance application, you would have had to declare and name ten individual integral variables.

This is shown in the following example:

```
1 public class Ages
2 {
3     public void makeAges()
4     {
5         int aOne = 1;
6         int aTwo = 2;
7         int aThree = 3;
8         int aFour = 4;
9         int aFive = 5;
10        int aSix = 6;
11        int aSeven = 7;
12        int aEight = 8;
13        int aNine = 9;
14        int aTen = 10;
15    }
16 }
```




The Problem: Creating Many Variables of the Same Type

- Disadvantages:
 - ▼ Becomes unmanageable
 - ▼ Is tedious
 - ▼ Duplicates work
 - ▼ Requires unique identifiers for each variable

Disadvantages

Now, imagine the length of a program in which you need to store the ages of 100,000 people.

This approach to storing large numbers of values has numerous disadvantages:

- As the number of values increase, it becomes increasingly unmanageable.
- It is tedious.
- It forces you to duplicate statements which perform the same task on different values.
- It requires unique identifiers for each variable, increasing the likelihood of creating bugs through typing mistakes and so on.



The Solution: Arrays

- Arrays let you store as many values as you need.
- Arrays can have primitive types or reference types.

```
char [] status;    // char array
int [] numbers;   // int array
String [] names;  // String array
Shirt [] shirts;  // Shirt array
```

- Each part of the array is an *element*.

```
numbers [6];      // 7th item in an array of 0 to 6
numbers [10];     // 11th item in an array of 0 to 10
```

The Solution: Arrays

Arrays are the solution to this problem. You can store as many values as you need in an array, using just one identifier, and a number stating how many values it contains.

You can declare references to arrays of any type, primitive types or reference types:

```
char [] status;    // char array
int [] numbers;   // int array
String [] names;  // String array
Shirt [] shirts;  // Shirt array
```

Each part of the array is an *element*; if you declare an array of 100 ints, there are 100 elements.

In order to refer to a specific element within the array, you state the identifier and a specific *index*, specifying whether the variable is the 7th item in the array, the 11th, and so on. The index is the number that represents the location, or position, in the array.

```
numbers [6];      // 7th item in an array of 0 to 6
numbers [10];     // 11th item in an array of 0 to 10
```



Arrays and the main Method

- You have already used `main` to take an array of arguments from the command line.

```
public static void main (String args[]);  
args [0];
```

Arrays and the main Method

You have already used arrays with the main method, which takes an unspecified number of Strings:

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

You specified which variable you wanted like this:

```
args [0];
```



Arrays and Type

- Arrays are references to objects
- Array of `ints` has the type "array of `int`"
- Array of `Shirts` has the type "array of `Shirt`"

Arrays and Type

You can create an array of primitive types, or an array of references to object types; as shown in Figure 9-1 on page 9-3, you can create an array of `ints` or an array of `Shirts`.

Once you declare the array, each array variable has the reference type "array of *type*". The variable names could be of type "array of reference to `String`". The variable numbers could be of type "array of `int`".



Creating Primitive-Type Arrays

- Arrays of primitive types or reference types are objects.
- Three steps:
 1. Declaring
 2. Instantiating
 3. Initializing

Creating Primitive-Type Arrays

Because arrays are actually object references, there are three steps to using them:

1. Declaring – State the name and type of the reference
2. Instantiating – Create the array itself using the new keyword and specifying length
3. Initializing – Specify the value for each variable within the array if you want a value other than the default value (zero or the equivalent)

Declaring Primitive Arrays

Syntax:

There are two approaches to declaration syntax, putting the brackets before the array identifier, and putting the brackets after the array identifier.

```
type [] array_identifier;  
type array_identifier [];
```

Example:

```
int [] ages;  
char status[];
```

The first way, *type [] array_identifier*, is generally preferred because it is more obvious when you read it in code (the brackets are closer to the beginning of the line).

Examples in the rest of this module will show the brackets before the identifier.

When you declare an array, the compiler and the JVM have no idea how large the arrays will be. You have declared reference variables which do not currently point to any objects. The objects are created when you state how large you require the arrays to be (instantiation).

Instantiating Primitive Arrays

Arrays are objects, so you create the new array using the `new` keyword. The bracket in the declaration (`[]`) makes the declarations different.

Syntax:

```
array_identifier = new type [length];
```

Example:

```
status = new char [20];  
ages = new int [5];
```

You can declare and instantiate arrays in one statement each as follows:

Syntax:

```
type [] array_identifier = new type [length];
```

Example:

```
char [] status = new char [20];  
int [] ages = new int [5];
```

Initializing Primitive Arrays

When you instantiate an array object, every element is initialized to the zero value for the type you specified. In the case of the char array `status`, each value is initialized to the zero (`\0000` is the null character of the Unicode character set). For the int array `squares` it will be the integer value 0.

Syntax:

```
array_identifier[index] = value;
```

Example:

```
int [] ages;  
ages = new int[5];  
ages[0] = 19;  
ages[1] = 42;  
ages[2] = 92;  
ages[3] = 33;  
ages[4] = 46;
```

Declaring, Initializing, and Instantiating Primitive Arrays

If you know now the actual values you want an array to hold, you can declare, initialize, and instantiate all in the same line.

Syntax:

```
type [] array_identifier = {comma-separated list of values or expressions};
```

Example:

The following statement is equal to the previous instantiation example:

```
int [] ages = {19, 42, 92, 33, 46};
```

This syntax is permitted only in the same statement (the same line of code) as the array declaration. This is true for primitives and objects. The following would return an error:

```
int [] ages;  
ages = {19, 42, 92, 33, 46};
```


Creating Reference-Type Arrays

Creating arrays of objects, like `Strings` or `Shirts`, is very similar to the process for primitive types. The same three steps are involved:

1. Declaring
2. Instantiating
3. Initializing

Declaring Reference Arrays

Syntax:

```
type [] array_identifier;  
type array_identifier [];
```

Example:

```
Shirt [] shirts;  
String names[];
```

Instantiating Reference Arrays

Syntax:

```
array_identifier = new type [length];
```

Example:

```
names = new String [7];  
shirts = new Shirt [5];
```

You can declare and instantiate arrays in one statement each as follows:

Syntax:

```
type [] array_identifier = new type [length];
```

Example:

```
String [] names = new String [7];  
Shirt [] shirts = new Shirt [5];
```

Instantiating an array of Shirt references does not create the Shirt objects. This must be done separately.

Initializing Reference Arrays

Reference arrays are initialized to null references.

Initializing One Element at a Time

Syntax:

```
array_identifier[index] = value;
```

Example:

```
Shirt [] shirts = new Shirt [3];  
  
// Use the various constructors  
// to create the objects in the array  
shirts[0] = new Shirt();  
shirts[1] = new Shirt("Chambray", 'L');  
shirts[2] = new Shirt("Dress", 'M');
```

Declaring, Initializing, and Instantiating Reference Arrays

If you know the actual values you want an array to hold, you can declare, initialize, and instantiate all in the same line. You can set values for each object using constructors.

Syntax:

```
type [] array_identifier = {comma-separated list of values or expressions};
```

Example:

The following code is all on the same “line” because there is only one semicolon, at the end after the final initialization.

```
Shirt [] shirts = {new Shirt(),  
                  new Shirt("Chambray", 'L'),  
                  new Shirt("Dress", 'M')};
```

Accessing a Variable Within an Array

Each element of an array is accessed by a number representing its position within the array. To use the elements of the array, use the square brackets to state the index number required.

Note – The first element of an array is index 0, so the last element of a 6-element array is index 5.

Syntax:

array_identifier[index]

Example: Assigning a Literal Value to an Array Element

```
status [0] = '3';  
names [1] = "Derek Smalls";  
ages [1] = 19;  
price [2] = 9.99F;
```

Example: Assigning an Array Element to Another Variable

```
int i = ages[3];
```

Exercise 1: Creating and Using Arrays

Exercise objective – Use arrays in a program.



Tasks

Follow the directions from your instructor to locate the exercises directory for this module.

1. Create a basic program called `StorePrices` that creates an array of five prices and assigns values. Print the value of the first and last values to the screen.
2. Write a program that creates and assigns values to an array of `Shirt` objects. In the `Shirt` class, declare at least two variables such as size and price. Create another class in another file that creates the array of shirts. (Refer to “Creating Reference-Type Arrays” on page 9-13 in your student guide, including “Declaring, Initializing, and Instantiating Reference Arrays” on page 9-15.)



How Arrays Are Stored in Memory

- All arrays are objects.
- Declaring an array declares reference variables.

How Arrays Are Stored in Memory

As noted before, in the Java programming language, the arrays you create are actually objects, regardless of whether the type you specify is primitive or reference. The declarations do not therefore declare the actual array but the array reference variables.

The following pages illustrate how primitive and object arrays are stored in memory.

Primitive Variables and Arrays

Figure 9-2 shows how primitive arrays are stored in memory, in comparison to primitive variables.

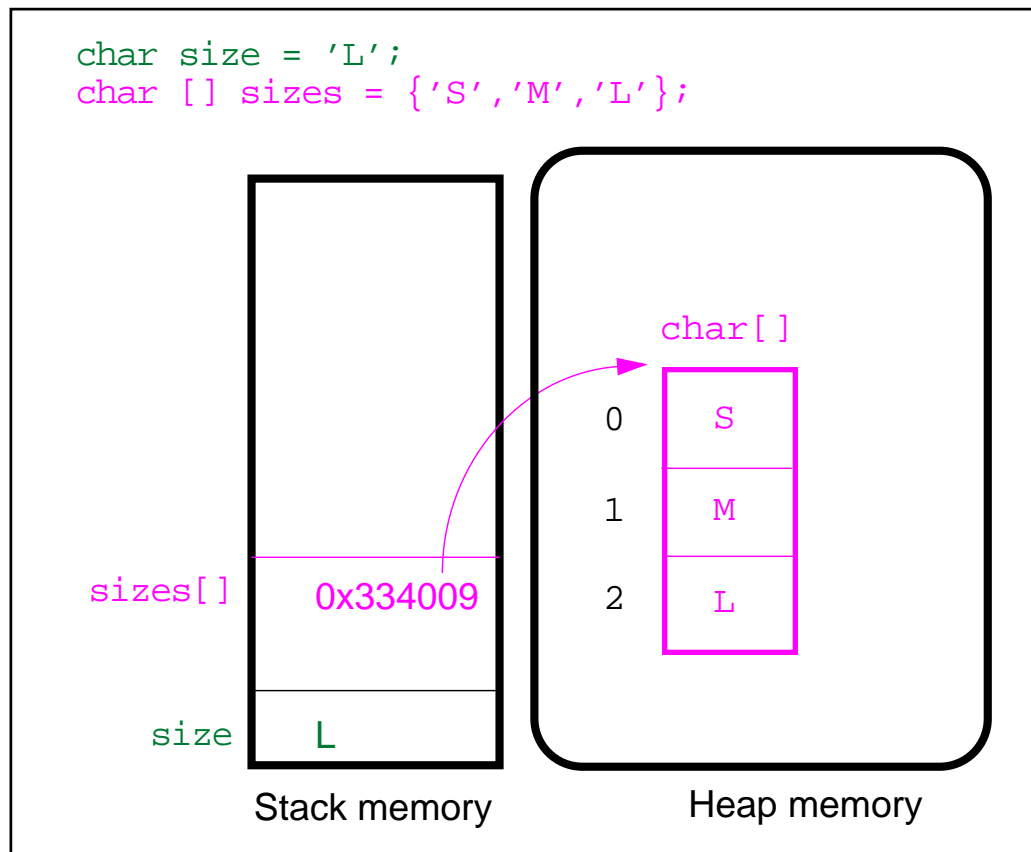


Figure 9-2 How Primitive Arrays Are Stored in Memory

The value of `size` is `L`, the value of `sizes[]` is `0x334009` and points to an object of type “array of char” with three values. The value of `sizes[1]` is the char `M`.

Reference Variables and Arrays

Figure 9-3 shows how object arrays are stored in memory.

Note – Assume that only the following Shirt constructor is defined in the Shirt class: `Shirt(String type, char size);`

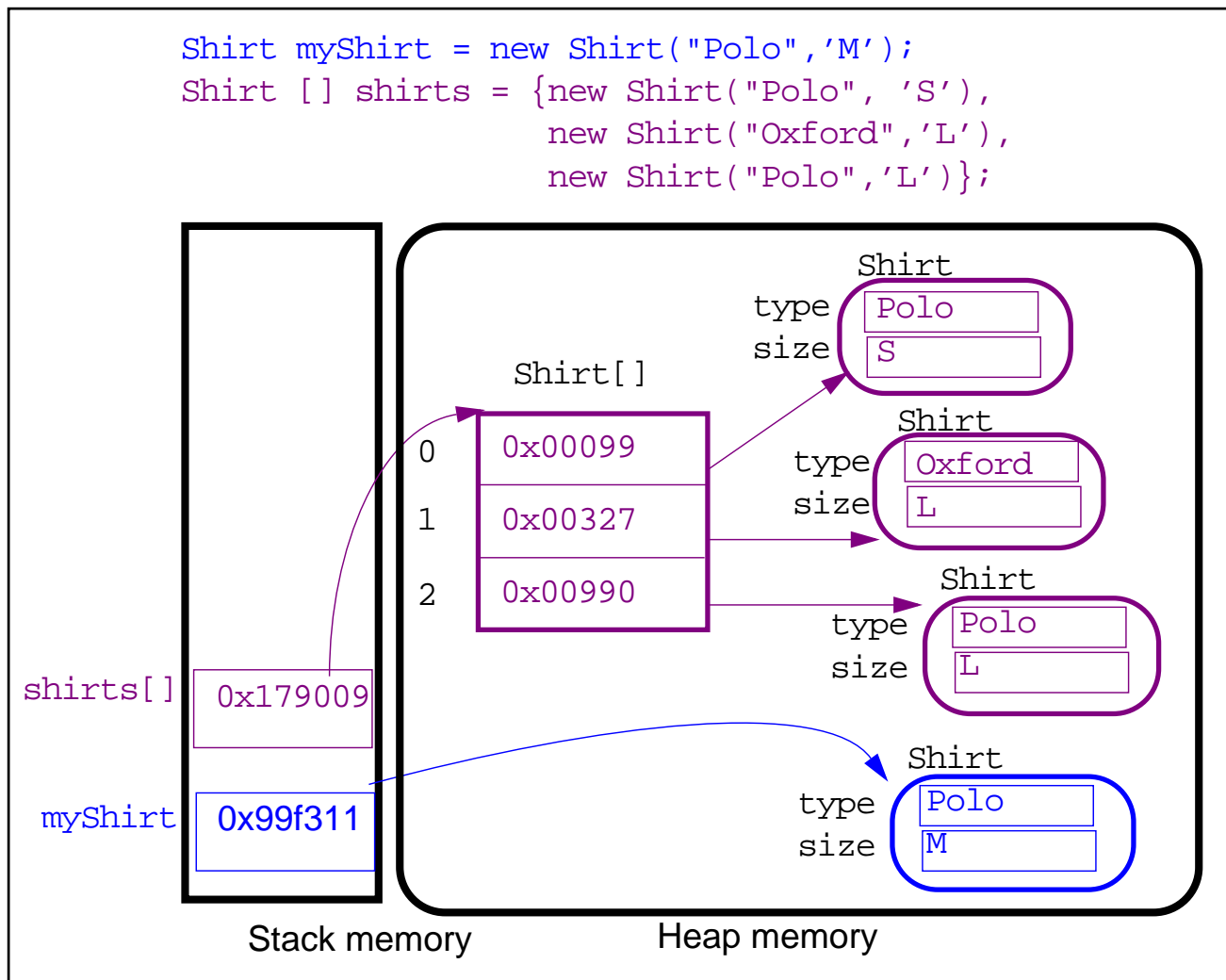
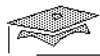


Figure 9-3 How Reference Arrays Are Stored in Memory

The value of `myShirt` is `x99f311`, pointing to an object of type `Shirt` with the values `Polo` and `M`. The value of `shirts[]` is `x179009`, pointing to an object of type "array of Shirts" with three values: `0xf0099`, `0x00327`, and `0x00990`. The value of `shirts[2]` is `0x00990`, which points to an object of type `Shirt` with the values `Polo` and `L`.



Array Bounds

- An array is an object and an object knows its state.
- Therefore, an array knows its length.
- The number of elements in an array is stored as part of the array object.
- Attempting to access an element of the array that does not exist causes an error.

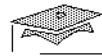
Array Bounds

An array is an object and an object knows its state, so an array knows its length.

An array's length is also referred to as its *bounds*. The bounds of an array like `month[12]` is from `month[0]` to `month[11]`, because all arrays start with element zero (0).

The number of elements in an array is stored as part of the array object. The value is used by the JVM to ensure that every access to the array corresponds to an actual element of the array.

If you attempt to access an element of the array that does not exist, such as specifying `month[21]` in your code for a squares array with length [12] you will receive an error.



Finding the Length of an Array

- You can determine the length of an array at runtime using the length member variable.

```
array_identifier.length
```

```
float salesAmounts[] = new float [10];
```

- Stepping through the array:

```
float salesAmounts[] = new float [10];
for (float i = 0; i < salesAmounts.length; i=i+1)
{
    loop_statements
}
```

Finding the Length of an Array

You can determine the length of an array at runtime using the length member variable.

Syntax:

```
array_identifier.length
```

Example:

```
float salesAmounts[] = new float [12];
```

`salesAmounts.length` is equal to the length of the `salesAmounts` array when it was declared, such as 12, if they are monthly sales amounts.

To step through an array, you could use the following for construct:

```
float salesAmounts[] = new float [12];
for (float i = 0; i < salesAmounts.length; i=i+1)
{
    loop statements
}
```

Notice that the limit of the loop is determined by comparison with `list.length`, rather than with the hard-coded value 12. This is generic code; the program will continue to work if the array length is increased to 15, for instance.



Sun Educational Services

Setting Array Values Using a Loop

- Setting specific values for each element can be tedious.
- A loop can do it automatically for values that increment.

Setting Array Values Using a Loop

You probably found setting values for arrays somewhat tedious, because you need to write a line for each element in the array. You can sometimes use a loop to automatically go through each element in the array length and set a value, based on an increment. This does not work for arrays of names, for example, but is quite useful if you need to create a series of consecutive numbers.

Use the length member variable in the loop. The following example sets values for the array `streets`, for a city with 112 consecutively numbered streets.

```
1 public class CreateStreets
2 {
3     public static void main (String args[])
4     {
5         Streets streetsObject = new Streets();
6         streetsObject.nameStreets();
7         streetsObject.printInfo();
8     }
9 }

1 public class Streets
2 {
3     int i;
4     int [] streets = new int [112];
5     public void nameStreets()
6     {
7         for (i = 0; i < streets.length; i++)
8             streets [i] = i+1;
9     }
10
11     public void printInfo()
12     {
13         System.out.println ("First street number is " + streets[0]);
14         System.out.println ("Last street number is " + streets[i-1]);
15     }
16 }
```

Exercise 2: Using Loops and Arrays

Exercise objective – Use loops to create an array.



Tasks

1. Create files called `IntArray` and `TestIntArray` that create an array of integers to represent the numbers 1 through 1000. Print the array length to the screen.
2. Write an `Ages` program that will fill an array of ten positions with the ages of ten people you know. (Hard-code the ages into your program, don't try to use user input.) Calculate and print the oldest age, the youngest age, and the average age.
3. Copy the `Ages` program and write a program called `InputAges` to perform those calculations on whatever values the user types in on the command line. (Thus, the ages are in the array `args` in the main method.)
4. Write a program called `TimesArray` to put the 10 times table into an array. Then print the times table from the array.



Two-Dimensional Arrays

```
type [][] identifier = new type [size_1][size_2];

// Declares and instantiates one-dimensional array, 12 elements
int [] monthlYsalesAmounts = new int[12];

// Declares and instantiates two-dimensional array, 376 elements
int [][] repSales = new int[33][12];
```

Two-Dimensional Arrays

All the arrays you have seen so far have been one-dimensional, like a list of values. You also can store data using two-dimensional arrays.

Two-dimensional arrays can hold values like the monthly sales amounts for each customer service representative. A two-dimensional array can be compared to a spreadsheet, with rows and columns for two different types of values.

Two dimensions simply require more sets of square brackets; the process of creating and using them is otherwise the same.

Syntax:

Use an additional set of brackets and a size.

```
type [][] identifier = new type [size_1][size_2];
```

Example:

The following examples show monthly sales amounts, and monthly sales amounts over five years.

```
// Declares and instantiates a one-dimensional array of 12 elements
int [] monthlYsalesAmounts = new int[12];
```

```
// Declares and instantiates a two-dimensional array of 60 elements
int [][] YearlySales = new int[5][12];
```

The following example uses the length variable and steps through the values in both dimensions for the two-dimensional array, fiveYears, with a dimension for 12 months and 5 years.

```
1 public class Arrays
2 {
3     public static void main (String args[])
4     {
5         int [][] fiveYears = new int[5][12];
6         int yr, mth;
7
8         // Write code to fill the array before you print it.
9         ...
10
11        for (yr=0 ; yr < fiveYears.length ; yr=yr+1)
12        {
13            for (mth=0 ; mth < fiveYears[yr].length ; mth=mth+1)
14            {
15                System.out.println("" + fiveYears[yr][mth]);
16            }
17        }
18    }
19 }
```


Exercise 3: Two-Dimensional Arrays

Exercise objective – Use two-dimensional arrays in a program.



Tasks

1. Write a program called *Checkers* that creates a two-dimensional array of chars. The board should be 8 by 8. Then assign values to the appropriate elements of the array to start a game (specify the value of each element as R for red, B for black).
2. Write a program called *CSRs* that tracks the names of the six CSRs in the two sales departments (each department has three CSRs). Assign a name to each CSR, and print all information to the screen.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Describe an array
- ☐ Explain why arrays are useful
- ☐ Write code to use arrays for storing primitive values
- ☐ Write code to use arrays for storing references to objects
- ☐ Write code to use one- and two-dimensional arrays

Think Beyond

Previous modules have mentioned and shown examples of the Java API, which contains classes you can add to your code. What are some API classes you have seen so far, and what other classes would be useful to find in the API? What applications might they have in programming the online clothing store?

Objectives

Upon completion of this module, you should be able to:

- Define inheritance
- Declare superclasses and subclasses
- Write code to implement inheritance
- Test for valid inheritance
- Define packages
- Use classes from packages in programs

This module describes the concepts of and how to implement inheritance in the Java programming language.

Relevance



Discussion – What does inheritance mean, in the context of object orientation?

Inheritance Overview

The online clothing company that examples in this course have been drawn from requires a personnel system to support the organization and storage of Manager objects, CSR objects (customer service representatives), and so on. The managers, CSRs, and other employees share many characteristics, such as name, address, salary, and so on.

Note – The diagram does not show all possible members; selected members are shown to keep the example simple.

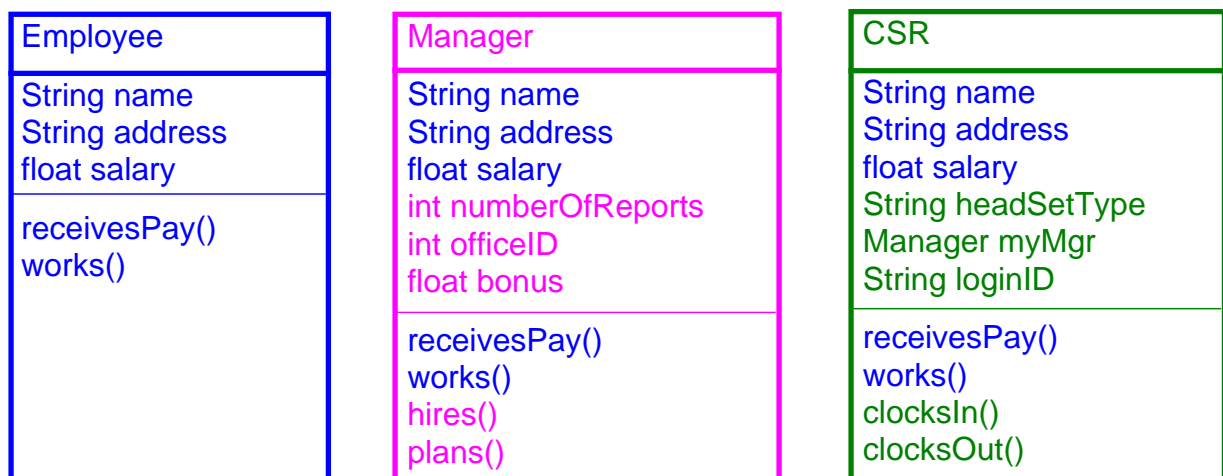


Figure 10-1 Classes With Duplicate Members

To reduce the duplication, each type of employee could *inherit* characteristics from an Employee class.

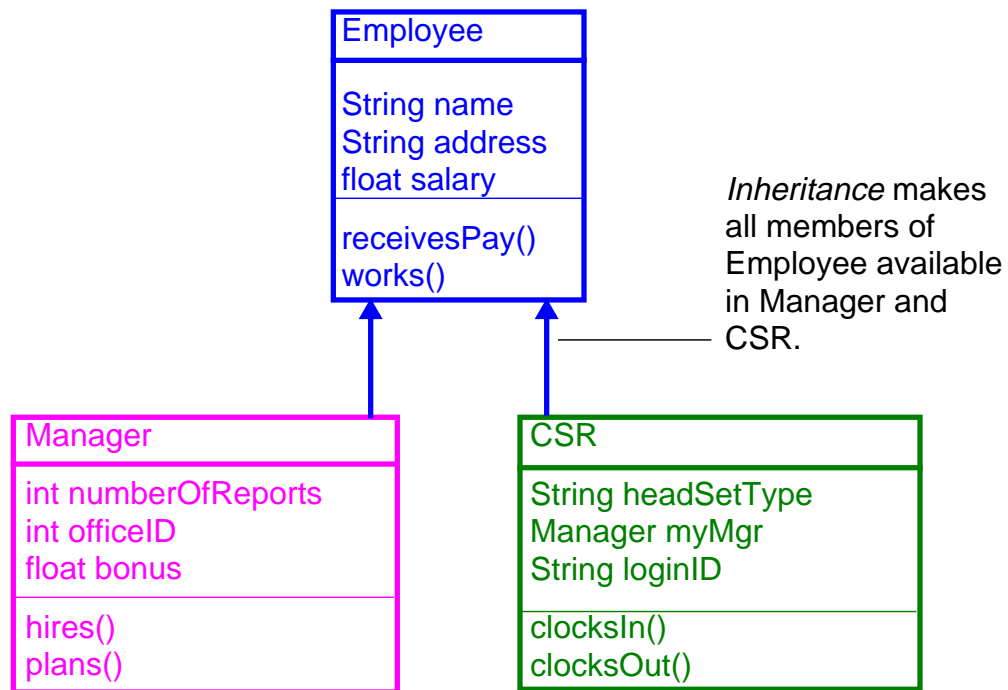


Figure 10-2 Inheritance: Manager and CSR Inherit From Employee



Inheritance Definition

- Lets common members be defined in one class and shared by other classes
- Class inherited from: superclass or parent class
- Class that inherits: subclass or child class

Inheritance Definition

Inheritance enables programmers to have common members (variables and methods) defined in one class, then have subsequent classes base themselves, or *inherit* from, that class. Another way to phrase it is that the subsequent classes *extend* the common class.

The class containing members common to several other classes is called the *superclass* or the *parent class*. The classes that inherit from, or extend, the superclass, are called the *subclasses* or the *child classes*.

Syntax and Examples

Syntax:

Use the Java `extends` keyword to indicate inheritance in your program. The following is the subclass declaration:

```
modifiers class subclass_name extends superclass_name
{
    subclass_code_block
}
```

Example:

Manager and CSR *extend* (the process of inheriting from) the Employee class. The following are declarations of the Employee, Manager and CSR classes.

```
1 public class Employee
2 {
3     // the following members are common to Employee,
4     // Manager, and CSR
5     String name;
6     String address;
7     float salary;
8     public float receivesPay()
9     {
10         return salary;
11     }
12     public int works(int workDone)
13     {
14         return workDone;
15     }
16 }
```

```
1 public class Manager extends Employee
2 {
3     int numberOfReports;
4     int officeID;
5     float bonus;
6     public void hires(String newEmployee)
7     {
8         // method tasks
9     }
10    public void plans()
11    {
12        // method tasks
13    }
14    // and so on
15 }
```

```
1 public class CSR extends Employee
2 {
3     String loginID;
4     String headsetType;
5     String startTime;
6     String endTime;
7     Manager myMgr = new Manager();
8     public String clocksIn()
9     {
10        // method tasks
11        return startTime;
12    }
13    public String clocksOut()
14    {
15        // method tasks
16        return endTime;
17    }
18    // and so on
19 }
```

Adding Abstraction

You could go up from Employee, rather than down, and create a class called Person from which both Customers and Employees would inherit. (The diagram does not show all possible members.)

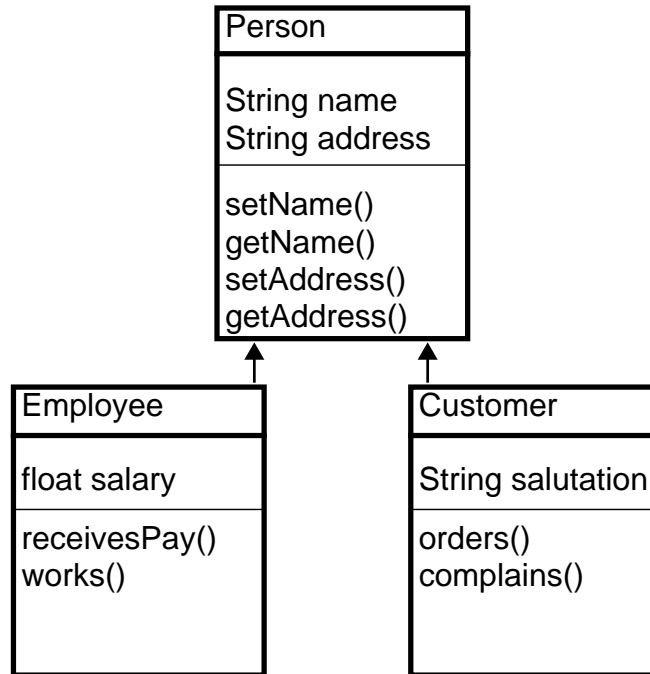


Figure 10-3 Inheritance: Manager and CSR Inherit From Employee

The following example shows the code to define each class. The methods for setting and getting names are divided into three categories, for the first name, middle initial, and last name.

```
1 public class Person
2 {
3     private String firstName;
4     private String lastName;
5     private char middleInitial;
6
7     public void setFirstName(String newVal)
8     {
9         firstName = newVal;
10    }
11    public void setLastName(String newVal)
12    {
13        lastName = newVal;
14    }
15    public void setMiddleInitial(char newVal)
16    {
17        middleInitial = newVal;
18    }
19    public void setName(String f, char m, String l)
20    {
21        setFirstName(f);
22        setMiddleInitial(m);
23        setLastName(l);
24    }
25    public String getFirstName()
26    {
27        return firstName;
28    }
29    public String getLastName()
30    {
31        return lastName;
32    }
33    public char getMiddleInitial()
34    {
35        return middleInitial;
36    }
37 }
```

```
1 public class Customer extends Person
2 {
3     private String salutation;
4     public void setSalutation(String newVal)
5     {
6         salutation = newVal;
7     }
8     public void setName(String f, char m, String l, String s)
9     {
10        setFirstName(f);
11        setMiddleInitial(m);
12        setLastName(l);
13        setSalutation(s);
14    }
15    public String getSalutation()
16    {
17        return salutation;
18    }
19    public void print()
20    {
21        System.out.println(getSalutation() + " " + getFirstName() + " " +
22        getMiddleInitial() + ". " + getLastName());
23    }
```

```
1 public class Employee extends Person
2 {
3     int SSN;
4
5     public void setSSN(int newVal)
6     {
7         SSN = newVal;
8     }
9     public int getSSN()
10    {
11        return SSN;
12    }
13    public void print()
14    {
15        System.out.println(getSSN() + ": " + getFirstName() + " " +
16        getMiddleInitial() + ". " + getLastName());
17    }
```

```
1 public class MyPeople
2 {
3     public static void main(String args[])
4     {
5
6         Customer first = new Customer();
7         Customer second = new Customer();
8         Employee third = new Employee();
9         first.setName("Marge", 'P', "Gunderson", "Officer");
10        second.setName("Jerry", 'J', "Lundegaard", "Sales Director");
11        third.setName("Carl", 'P', "Showalter");
12        third.setSSN(293992345);
13        first.print();
14        second.print();
15        third.print();
16    }
17 }
```



Testing Inheritance

- A class can inherit from only one superclass at a time.
- Check inheritance with “is a” phrase.
- The Employee, Manager, and CSR example is correct:
 - ▼ A Manager is an Employee
 - ▼ A CSR is an Employee

Testing Inheritance

In one view of the pure OO paradigm, a class can inherit from only one superclass at a time (although some languages, like C++, allow a class to inherit from multiple parents). The Java programming language, like OO, permits only single inheritance.

Each class can inherit the members of only one other class, so it is very important to consider the best use of inheritance, and to use it only when it is completely valid or unavoidable.

The way to check if a proposed inheritance link is valid is to use the “is a” phrase.

The example in Figure 10-2 on page 10-4 shows a correct example of inheritance. A CSR is an Employee; a Manager is an Employee.

Consider the following two classes:

```
1 public class Shirt
2 {
3     char size;
4     String material;
5     float price;
6     int numberOfButtons;
7     // and so on
8 }

1 public class Skirt
2 {
3     char size;
4     String material;
5     float price;
6     int length;
7     // and so on
8 }
```

There is a lot of duplication between the two classes, so you might consider using inheritance and define the `Skirt` class this way:

```
1 public class Shirt
2 {
3     char size;
4     String material;
5     float price;
6     // and so on
7 }

1 public class Skirt extends Shirt
2 {
3     int length;
4     // and so on
5 }
```

This appears at first sight to be all right. Now use the validation phrase to check the validity of this inheritance: “A *subclass* is a *superclass*.” If it makes sense, keep it in your model. In this case, however, the validation phrase is “a skirt is a shirt” so the inheritance is invalid.

The validation phrase shows that this system of inheritance does not work. You can eliminate the duplicate members between classes by looking for a better superclass.

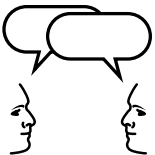
```
1 public class Clothing
2 {
3     char size;
4     String material;
5     float price;
6     // and so on
7 }

1 public class Shirt extends Clothing
2 {
3     int numberOfButtons;
4     // and so on
5 }

1 public class Skirt extends Clothing
2 {
3     int length;
4     // and so on
5 }
```

Then apply the validation phrase: "A skirt is clothing; a shirt is clothing."

Discussion – What other classes in the online clothing system would use inheritance? Test the inheritance you come up with to be sure each is valid.





Sun Educational Services

Using Code From the Java API

- Hundreds of classes are available to make writing your programs easier.
- Classes in the `java.lang` package are implicitly imported into all programs.
- Import or use the fully qualified name of classes in all other packages.

Using Code From the Java API

Another way you can use code from classes someone else has written is to use the Java API. You have been doing it throughout this course without it being pointed out, through `System.out.println` and other code. The Java API provides hundreds of classes that you can use in order to make your life easier. There are two categories: Classes that are implicitly available in any code you write, and classes you need to either import, or fully qualify.

Implicitly Available Classes

Classes in the `java.lang` package are implicitly imported into all programs. `java.lang` contains classes which form the core of the language, such as `String`, `Math`, `Integer`, and `Thread`.

Classes You Must Import or Fully Qualify

You need to import or use the fully qualified name (package and class name) of classes in all other packages, including the following:

- `java.awt` contains classes that make up the AWT. This package is used for constructing and managing the graphical user interface of the application.
- `java.applet` contains classes that provide applet-specific behavior.
- `java.net` contains classes for performing network related operations and dealing with sockets and URLs.
- `java.io` contains classes that deal with file I/O.
- `java.util` contains utility classes for tasks such as generating random numbers, defining system properties, and using date and calendar related functions.

Using the Classes

When you want to use classes from the API, you can import the whole package, or you can specify the fully qualified name of the class you are using.

Most programmers import the class or the package, rather than typing the fully qualified name.

Importing

Use the `import` statement to tell the compiler in which package to find the classes.

Syntax:

Use the following statement to import a package, in the first line of your program:

```
import package_name[classname]
```

Example:

```
1 import java.awt.*; // or import java.awt.Button
2 public class MyPushButton extends Button
3 {
4     // class code block
5 }
```

The `*` indicates that all classes in the package are imported. You can import the entire package, or only a specific class.

Note – The `import` statements must precede all class declarations.

Specifying the Fully Qualified Name

Instead of importing the `java.awt` package, you could refer to the `Button` class as `java.awt.Button` throughout the program.

```
2 public class MyPushButton2 extends java.awt.Button
3 {
4     // class statements
5 }
```

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Define inheritance
- ☐ Declare superclasses and subclasses
- ☐ Write code to implement inheritance
- ☐ Test for valid inheritance
- ☐ Define packages
- ☐ Use classes from packages in programs

Think Beyond

What other advantages might inheritance provide?

Objectives

Upon completion of this appendix, you should be able to:

- Explain the advantages of polymorphism
- Describe how to implement polymorphic methods
- Explain how to create arrays containing multiple object types
- Define and give examples of containment
- Describe interfaces and how they are used

This appendix describes how to implement more advanced object-oriented concepts in the Java programming language.

Relevance



Discussion – Could you use an instance of a superclass to refer to an instance of a subclass? Why or why not?



Containment

- Objects that interact or are dependent on each other are coded using containment.
- Containment can be:
 - ▼ CSR and computer
 - ▼ Computer and hard disk

Containment

Objects typically have relationships, both in the real world and in your program. As students, you are related to this class, your instructor, and these materials. How each object behaves can affect the other objects in the system. These relationships are called *containment*. When you write programs using containment, you create objects that contain references to other objects.

There are different kinds of containment. In the online clothing store example, a CSR and his computer work together to take orders. The computer itself contains a hard disk; the computer and hard disk objects also have a containment relationship.

Containment involves “uses a” relationships (the CSR uses a computer) and “has a” relationships (the computer has a hard disk).

Containment primarily affects the items that are used (the computer) or that compose an object (the CPU): if the object they are part of is removed, are they removed with it? If you remove a computer, the hard disk is removed too. If you remove a CSR, the computer probably stays.

Containment Examples

DirectClothing sells suits for women. There is no such thing as a suit that exists by itself, however—the suit item number actually references a shirt, skirt, and jacket.

The composing class initializes the objects it is composed of within its statement block.

```
1 class Shirt
2 {
3     // class_code_block
4 }
5
6 class Jacket
7 {
8     // class_code_block
9 }
10
11 class Skirt
12 {
13     // class_code_block
14 }
15
16 public class Suit
17 {
18     Shirt myShirt = new Shirt();
19     Jacket myJacket = new Jacket();
20     Skirt mySkirt = new Skirt();
21     // class_code_block
22 }
23 class Order
24 {
25     public static void main(String args[])
26     {
27         Suit suit1 = new Suit();
28     }
29 }
```

The Suit class is declared and then instantiates the three objects it is composed of.

Figure A-1 shows how this composition example would be stored in memory.

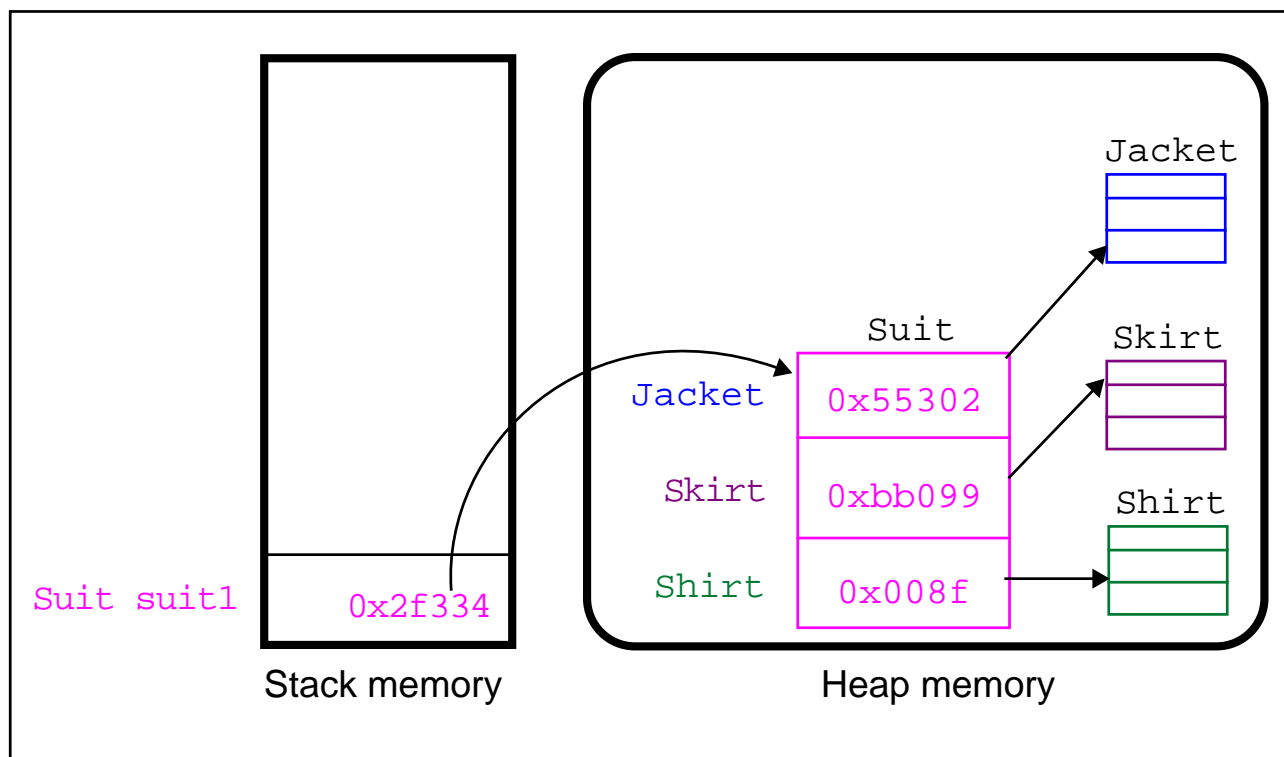


Figure A-1 Composition and Memory

Consider this example, as well. Each CSR has a computer and a headset.

```

1 public class CSR
2 {
3     Computer myComputer;
4     Headset myHeadset;
5
6     public void setComputer (Computer aComputer)
7     {
8         myComputer = aComputer;
9     }
10
11     public setHeadset (Headset aHeadset)
12     {
13         myHeadset = aHeadset;
14     }
15 }

```



Sun Educational Services

Polymorphism

- Means "many forms"
- Each of the subclasses is a form of the superclass it extended
- For example, polymorphism lets you refer to subclass objects using a reference to the superclass

Polymorphism

Polymorphism means "many forms." It is used in OO to emphasize the fact that inheritance extends one class into one or more others. Each of the subclasses *is a* form of the superclass it extended, which means that it is another *form* of that class. A superclass with multiple subclass would have many forms of itself.

Review the diagram from the inheritance example, shown again in Figure A-2.

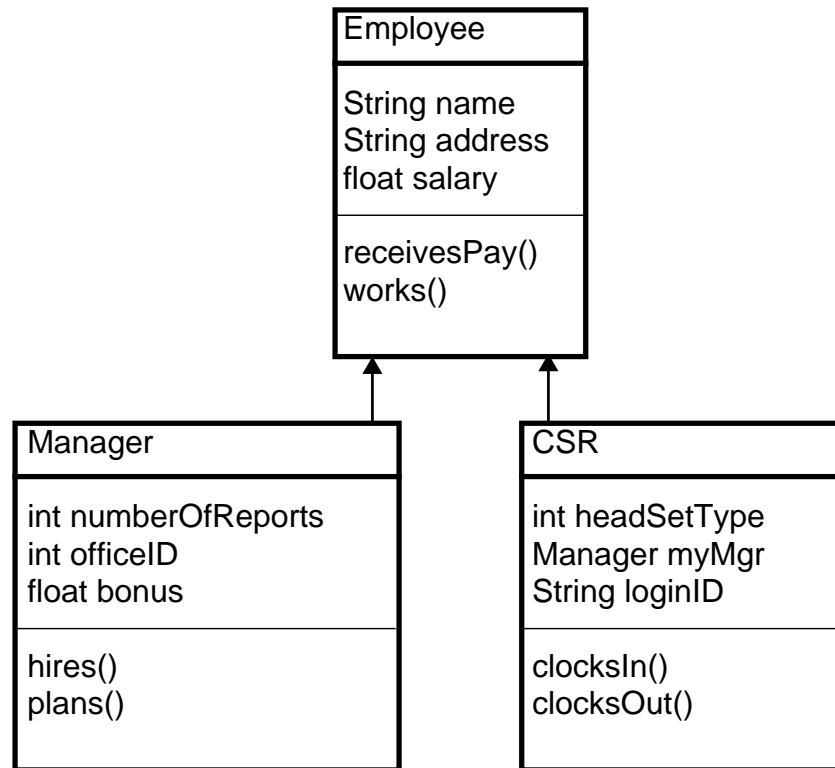


Figure A-2 Inheritance

Polymorphism can go down in the inheritance hierarchy, but not across or up. That is, if you create a **Manager** object, you cannot use it to refer to **CSR** or **Employee** objects or members. However, if you create an **Employee** object, you can refer to **Manager** or **CSR** objects or members.

Figure A-3 illustrates how this works in memory. **Employee** objects (superclasses) can refer to any objects or object members within the inheritance hierarchy; **CSR** and **Manager** objects (subclasses with no subclasses of their own) can refer only to themselves.

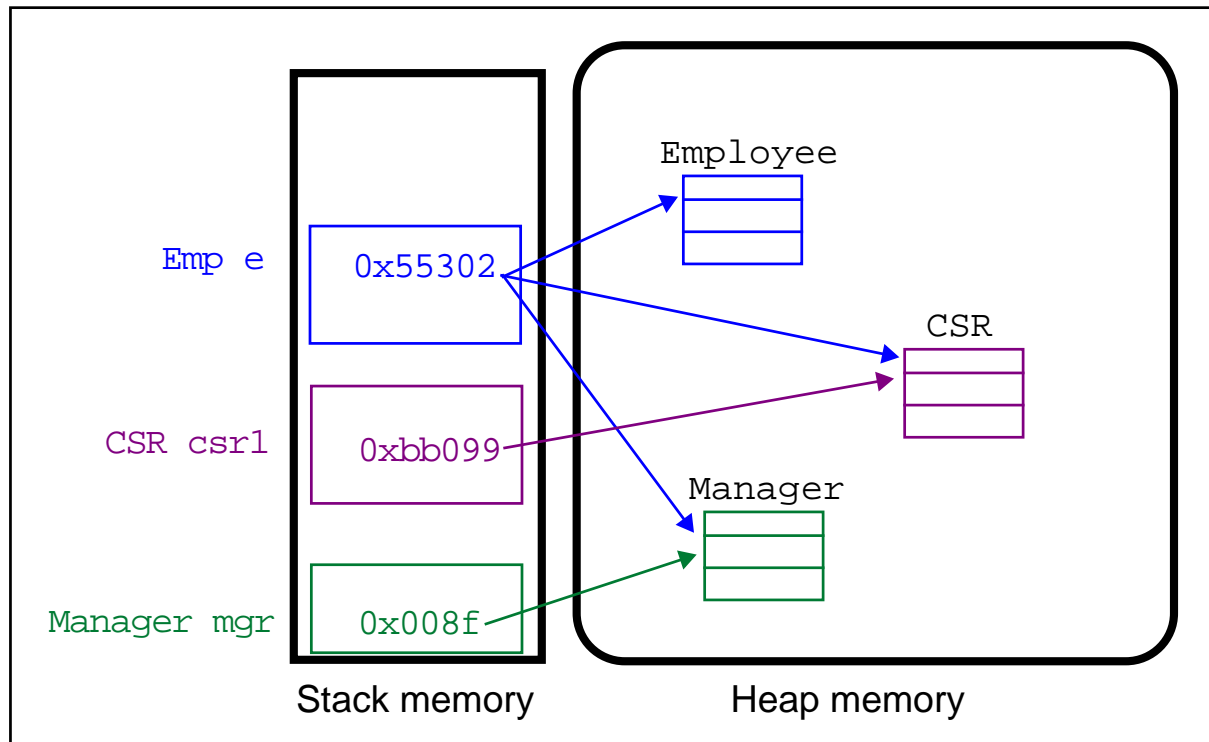


Figure A-3 Polymorphism and Memory



Referring to Multiple Possible Objects

- You can use a reference variable of a superclass to store address of an object of a subclass.
- A reference variable declared in the `Employee` class could refer to:
 - ▼ An `Employee` object
 - ▼ A `Manager` object
 - ▼ A `CSR` object

Referring to Multiple Possible Objects

Polymorphism allows you to use a reference variable of the superclass type to store the address of an object of one of its subclasses. With the `Employee` superclass and the `Manager` and `CSR` subclasses, a reference variable `e` declared in the `Employee` class could refer to an `Employee` object, a `Manager` object, or a `CSR` object.

This feature is convenient and adds flexibility to your programs, because you can point to an object from a number of possible classes using one reference variable.

The following example uses two `Employee` type reference variables and assigns them, instantiated, to a `Manager` object and a `CSR` object.

Example

```
1 public class Employee
2 {
3     // class_code_block
4 }

1 public class Manager extends Employee
2 {
3     // class_code_block
4 }

1 public class CSR extends Employee
2 {
3     // class_code_block
4 }

1 public class PolymorphicExample
2 {
3     public static void main (String args[])
4     {
5         Employee e1 = new Manager();
6         Employee e2 = new CSR();
7     }
8 }
```



Polymorphic Methods

- A method in a superclass is inherited by subclasses.
- The method is the same even though the implementation is different.
- `work()` would be different for CSR and Manager

Polymorphic Methods

A superclass `Employee` would include in its method a `work()` method. `Employee` subclasses, such as `Manager` or `CSR`, would inherit the same method. It is still the same method in all three classes, even though the work each does is different.

A method defined in a superclass that is inherited in a subclass or subclasses is polymorphic. You can apply a polymorphic method to objects of different classes to achieve the same general result. Polymorphism is an important concept in object-oriented systems that support reuse and software maintenance.

The implementation of a polymorphic method depends on the object it is applied to. The manager and CSR both come to work, sit down at their desk, and start doing things that accomplish tasks for DirectClothing, Inc. However, the manager attends meetings and writes five-year plans, and the CSR talks with customers and places orders.



Overriding Methods

- Methods in the same inheritance hierarchy can be overridden.
- If several classes inherit the same method, each subclass can override it.

Overriding Methods

Methods in the same inheritance hierarchy can be overridden. If several classes inherit the same method, each subclass can override the specific implementation of the method.

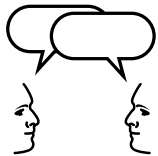
Polymorphism and Overriding

The following example shows a class `OrderTaker` with three different classes that inherit from it. Each is paid a different way. This demonstrates overriding and polymorphism.

```
1 abstract class OrderTaker
2 {
3     private String name;
4     private int basePay;
5     private int ordersTaken;
6
7     public void setName(String newValue)
8     {
9         name = newValue;
10    }
11
12    public void setBasePay(int newValue)
13    {
14        basePay = newValue;
15    }
16    public void incrementOrdersTaken()
17    {
18        ordersTaken++;
19    }
20    public String getName()
21    {
22        return name;
23    }
24    public int getBasePay()
25    {
26        return basePay;
27    }
28    public int getOrdersTaken()
29    {
30        return ordersTaken;
31    }
32    public double calculatePay()
33    {
34        //Generic order takers get a weekly portion of their salary
35        return getBasePay() / 52;
36    }
37 }
38
39 class CSR extends OrderTaker
```

```
40 {
41     public double calculatePay()
42     {
43         //CSRs get their weekly pay plus 10 cents per order they take
44         return getBasePay() / 52 + getOrdersTaken() * .1;
45     }
46 }
47
48 class OEC extends OrderTaker
49 {
50     public double calculatePay()
51     {
52         //OECs get their weekly pay plus 1 cent per order they take
53         return getBasePay() / 52 + getOrdersTaken() * .01;
54     }
55 }
56
57 class Telemarketer extends OrderTaker
58 {
59     private int peopleHarassed;
60
61     public void incrementPeopleHarassed()
62     {
63         peopleHarassed++;
64     }
65     public int getpeopleHarassed()
66     {
67         return peopleHarassed;
68     }
69     public double calculatePay()
70     {
71         //Telemarketers get 10 cents for every order they take and 1 cent per
person they harass (call)
72         return getBasePay() / 52 + peopleHarassed * .01 + getOrdersTaken() * .1;
73     }
74 }
75
76 class Clerks
77 {
78     public static void main(String args[])
79     {
80         OrderTaker first, second, third;
81         first = new OEC();
82         second = new CSR();
83         third = new Telemarketer();
84 }
```

```
85     first.setBasePay(52000);
86     second.setBasePay(52000);
87     third.setBasePay(52000);
88
89     first.incrementOrdersTaken();
90     first.incrementOrdersTaken();
91     second.incrementOrdersTaken();
92     third.incrementOrdersTaken();
93     ((Telemarketer) third).incrementPeopleHarassed();
94     System.out.println("First made " + first.calculatePay());
95     System.out.println("Second made " + second.calculatePay());
96     System.out.println("Third made " + third.calculatePay());
97 }
98 }
```



Discussion – What other operations could be polymorphic for DirectClothing, Inc.?



Arrays With Multiple Types

- References to `Manager` objects may not be stored in the `csrArray` object:

```
CSR [] csrArray = new CSR [10];
csrArray[0] = new CSR();
csrArray[1] = new CSR();
csrArray[2] = new CSR();
// and so on
```

- An array of `Employees` could store `Employees`, `Managers`, and `CSRs`.

```
Employee [] employeeArray = new Employee [10];
employeeArray[0] = new CSR();
employeeArray[1] = new Manager();
employeeArray[2] = new CSR();
// and so on
```

Arrays With Multiple Types

Up to now, you have learned that it is not possible to mix the types of value being stored in an array. For example, an array of `ints` can only hold `ints`.

Polymorphism allows some flexibility with this rule. Creating an array of class types means that each element of the array can be a reference to an instance of that class. That is not limited to the class itself, but to any of its subclasses. With the `Employee` superclass, you could create an array of `Employees` that include `Employee`, `CSR`, and `Manager` objects. An `Employee` reference can store the address of an instance of any subclass of `Employee`.

```
Employee [] employeeArray = new Employee [10];
employeeArray[0] = new CSR();
employeeArray[1] = new Manager();
employeeArray[2] = new CSR();
```

An array of `CSRs` or `Managers` can hold only `CSR` or `Managers`.

The following example demonstrates method overriding as well as creating a heterogeneous array of objects.

```
1 public class Pet
2 {
3     public void speak()
4     {
5         System.out.println("I am a generic pet.");
6     }
7 }
```

```
1 public class Dog extends Pet
2 {
3     public void speak()
4     {
5         System.out.println("Woof");
6     }
7 }
```

```
1 public class Cat extends Pet
2 {
3     public void speak()
4     {
5         System.out.println("Meow");
6     }
7 }
```

```
1 public class Duck extends Pet
2 {
3     public void speak()
4     {
5         System.out.println("Quack");
6     }
7 }
```

```
1 public class Animals
2 {
3     public static void main(String args[])
4     {
5         Pet myPets[] = new Pet[4];
6         myPets[0] = new Pet();
7         myPets[1] = new Cat();
8         myPets[2] = new Duck();
9         myPets[3] = new Dog();
10
11         for (int index=0; index<4; index++)
12             myPets[index].speak();
13     }
14 }
```



Abstract Methods

- Pet program example:
 - ▼ All pets need a speak method
 - ▼ *What* a generic pet says depends on what type of pet it is
 - ▼ An abstract method speak would enforce that all subclasses use a speak method
- If a subclass does not write a speak method, it becomes abstract

Abstract Methods

The previous example includes a generic pet class. In reality, of course, there is no such thing as a generic pet, nor is “I am a generic pet” what they say.

However, it is true that all pets communicate in one way or another, so you know that all of them will need a speak method. What they say depends on what kind of pet it is (dog, cat, gorilla, and so on). A method used this way is an abstract method

The following example uses an abstract method speak. It enforces that all subclasses of Pet, which the abstract speak method is in, must write their own speak to override it. If they do not, those classes in turn become abstract. (If a class has any abstract method (directly or inherited) it must be abstract. If a class is abstract you can not make an object of that class.)

Thus, the pet class defines that all pets must speak, but not how they speak.

```
1 abstract class Pet
2 {
```

```
3    abstract public void speak();
4 }

1 public class Dog extends Pet
2 {
3     public void speak()
4     {
5         System.out.println("Woof");
6     }
7 }

1 public class Cat extends Pet
2 {
3     public void speak()
4     {
5         System.out.println("Meow");
6     }
7 }

1 public class Duck extends Pet
2 {
3     public void speak()
4     {
5         System.out.println("Quack");
6     }
7 }

1 public class Animals
2 {
3     public static void main(String args[])
4     {
5         Pet myPets[] = new Pet[4];
6         myPets[0] = new Duck();
7         myPets[1] = new Cat();
8         myPets[2] = new Duck();
9         myPets[3] = new Dog();
10
11     for (int index=0; index<4; index++)
12         myPets[index].speak();
13 }
14 }
```

Interfaces

- A "public interface" is a contract between client code and the class that implements that interface
- A Java *interface* is a formal declaration of such a contract in which all methods contain no implementation
- Many, unrelated classes can implement the same interface
- A class can implement many, unrelated interfaces
- Syntax of a Java class:

```
<class_declaration> ::=  
    <modifier> class <name> [extends <superclass>]  
        [implements <interface> [,<interface>]* ] {  
        <declarations>*  
    }
```

Interfaces

Consider the pet example again. You can extend from only one class in the Java programming language because it does not allow multiple inheritance. If there were a superclass called `FourLegged`, a `Cat` and a `Dog` could not extend both `FourLegged` and `Pet`.

An abstract class that has only abstract methods and constants is called an interface instead. With an interface, a class *implements* it rather than extending it, so the result is like multiple inheritance.

A class can implement as many interfaces as you want, in addition to extending one class. `Cat` and `Dog` could implement `Pet`, if it were an interface, and they also could extend `FourLegged`.

An interface can also declare constants: `public`, `static`, and `final`

The following code shows the Pet example again, using interfaces.

```
1 interface Pet
2 {
3     abstract public void speak();
4 }

1 public class Dog implements Pet
2 {
3     public void speak()
4     {
5         System.out.println("Woof");
6     }
7 }

1 public class Cat implements Pet
2 {
3     public void speak()
4     {
5         System.out.println("Meow");
6     }
7 }

1 public class Duck implements Pet
2 {
3     public void speak()
4     {
5         System.out.println("Quack");
6     }
7 }

1 public class Animals
2 {
3     public static void main(String args[])
4     {
5         Pet myPets[] = new Pet[4];
6         myPets[0] = new Duck();
7         myPets[1] = new Cat();
8         myPets[2] = new Duck();
9         myPets[3] = new Dog();
10
11         for (int index=0; index<4; index++)
12             myPets[index].speak();
13     }
14 }
```

Uses of Interfaces

- Declaring methods that one or more classes are expected to implement
- Determining an object's programming interface without revealing the actual body of the class
- Capturing similarities between unrelated classes without forcing a class relationship
- Simulating multiple inheritance by declaring a class that implements several interfaces

Uses of Interfaces

Interfaces are useful for:

- Declaring methods that one or more classes are expected to implement.
- Revealing an object's programming interface without revealing the actual body of the class. (This can be useful when shipping a package of classes to other developers.)
- Capturing similarities between unrelated classes without forcing a class relationship.
- Simulating multiple inheritance by declaring a class that implements several interfaces.

Interface Example

Imagine a group of objects that all share the same ability: they fly. You can construct a public interface, called `Flyer`, that supports three operations: `takeOff`, `land`, and `fly`.

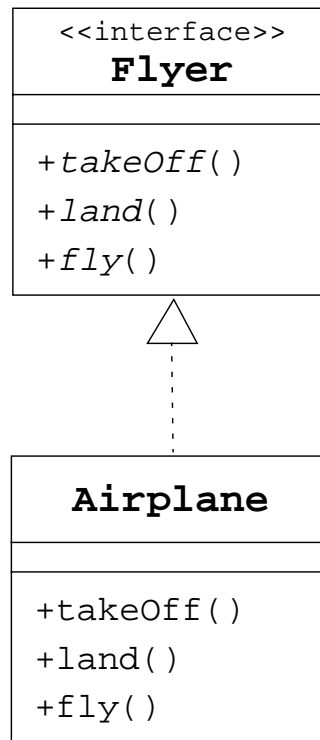


Figure A-4 The Flyer Interface and Airplane Implementation

The following code shows the general structure of how this might be implemented.

```
public interface Flyer
{
    public void takeOff();
    public void land();
    public void fly();
}

public class Airplane implements Flyer
{
    public void takeOff()
    {
        // accelerate until lift-off
        // raise landing gear
    }
    public void land()
    {
        // lower landing gear
        // decelerate and lower flaps until touch-down
        // apply breaks
    }
    public void takeOff()
    {
        // keep those engines running
    }
}
```


There can be multiple classes that implement the `Flyer` interface. We have seen that an airplane can fly, but a bird can also fly, Superman can fly, and so on.

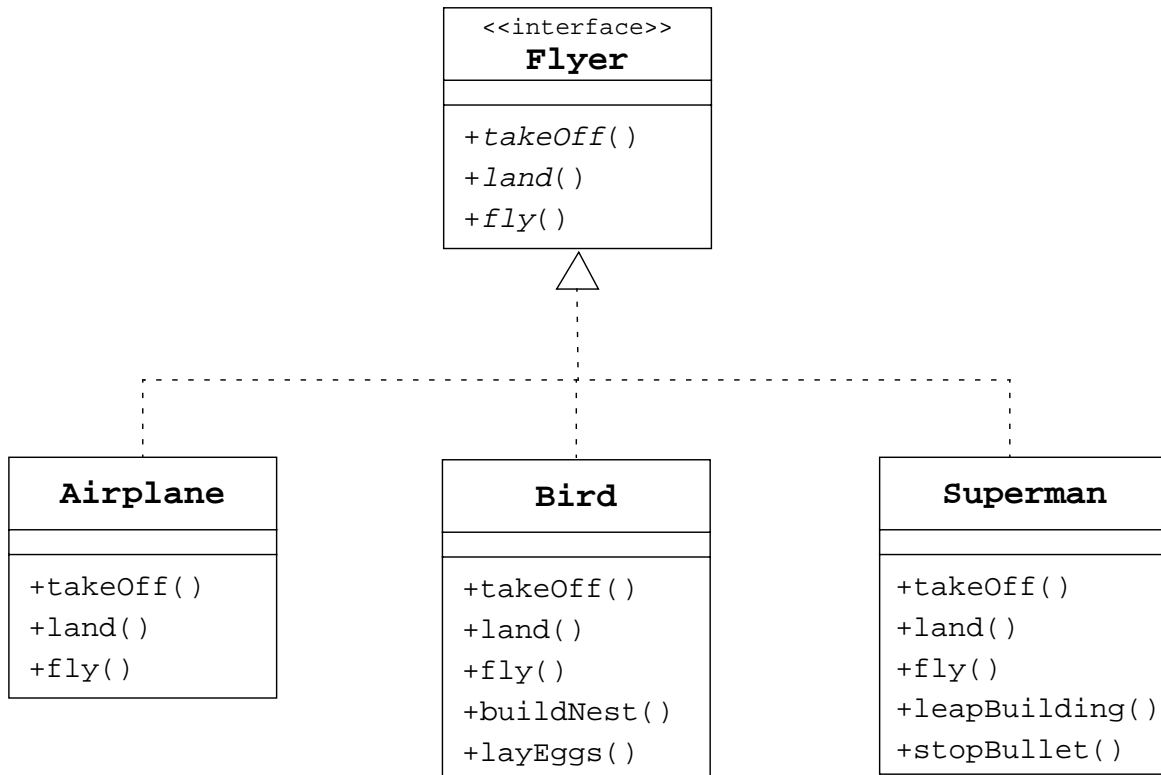


Figure A-5 Multiple Implementations of The `Flyer` Interface

An `Airplane` is a `Vehicle` and it can fly. A `Bird` is an `Animal` and it can fly. These examples show that a class can inherit from one class, but also implement some other interface.



The `this` Reference

- Storing members for each object instantiation can greatly increase memory requirements.
- `this` reference makes it unnecessary to store a separate copy of each object.
- `this` is included implicitly; you can specify it explicitly, as well.

The `this` Reference

When you start creating classes from which you instantiate objects, the classes can become large very quickly. In addition to data, each class can have multiple methods, including overloaded methods. When you instantiate objects, the instantiation stores a separate copy of each variable and method. With multiple objects, the memory requirements of a class can skyrocket.

The Java programming language provides a keyword, `this`, that lets you specify the current object, so that you do not have to store a separate copy of each variable and method for each instantiation of a class. `this` refers to the current object; you can use it explicitly, or let the Java programming language insert it implicitly.

When you invoke another object method from the same object, it is not necessary to put a reference in front:

```
1 public class Example
2 {
3     void method1()
4     {
5         method2();
6     }
7     void method2()
8     {
9         //method2 statements
10    }
11 }
```

method1 invokes method2 without a reference to an object. This is allowed because when the compiler finds an unreferenced method, it checks the local class for one that matches. If it finds one, it automatically adds the keyword `this` to the method call code. The `this` keyword means “reference to the same object.”

The previous code is equivalent to writing:

```
1 class Example2
2 {
3     void method1()
4     {
5         this.method2();
6     }
7     void method2()
8     {
9         // method2 statements
10    }
11 }
```

Typically, you do not want or need to refer to the `this` reference in your methods, but it is always there implicitly.

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ☐ Explain the advantages of polymorphism
- ☐ Describe how to implement polymorphic methods
- ☐ Explain how to create arrays containing multiple object types
- ☐ Define and give examples of containment
- ☐ Describe interfaces and how they are used

Think Beyond

Now that you can write the behind-the-scenes logic, what is left in order to let the users interact with the program?

What code would you need to write to allow this?

Graphical User Interface Development

B 

Objectives

Upon completion of this appendix, you should be able to:

- Describe the AWT package and its components
- Define the terms containers, components, and layout managers, and how they work together to build a graphical user interface
- Use the `FlowLayout` and `BorderLayout` managers to achieve a desired dynamic layout
- Add components to a container
- Use the `Frame` and `Panel` containers appropriately
- Describe how complex layouts with nested containers work
- Use the `Button`, `TextField`, `TextArea`, and `Label` components
- Describe how to handle events when a button is clicked

This appendix is an optional part of this course. It provides basic information about graphical user interfaces and how to create a simple window and corresponding functionality. This module is provided so that you can get an idea of the Java programming language's capabilities for graphical user interfaces, and understand a few of the main tools you can use to create them.

Relevance



Discussion – You know how to write programs to set and change values, make decisions, and perform other tasks. How would you create a graphical user interface (GUI) to allow users to interact with the program?

Assume you wanted to make a button that would save changes and close a window. List information you would need to supply to the computer to:

- Make a button
- Make a button click
- Make a button look like it clicks
- Tell your program when a button was clicked, and which one



Sun Educational Services

Extent of This Module

- Become familiar with the GUI tools
- Form a base for future learning
- Write a simple program
- Have fun!

Extent of This Module

This module does not attempt to teach you everything you will ever need to know about programming GUIs. This is only an introduction to GUI programming. You can learn more about GUI programming in SL-275: *Java™ Programming Language* or from other sources.



Sun Educational Services

The AWT

- Provides previously written classes to create a GUI
- In the `java.awt` package (Abstract Windowing Toolkit)
- Implements a comparable interface on all platforms

The AWT

In any programming language, writing a graphical user interface (GUI) from scratch is extremely time consuming. Thus, it is much better to use a set of previously written classes that will do the hard work for you.

The Java Development Kit (JDK) comes with a set of classes which allow you to easily create GUIs without writing the code from scratch. All of these classes are in a package called `java.awt`. AWT stands for Abstract Windowing Toolkit. The AWT classes are also designed to allow you to write generic GUI code that will implement a comparable GUI on any platform. (The GUI you create in this appendix will have a Windows look and feel under Microsoft Windows and a Solaris look and feel under the Solaris Operating Environment. However, the GUI will be essentially the same on all platforms.)



Sun Educational Services

Viewing Package Information

- In your browser, go to
`<JAVA BASE DIRECTORY>/api/docs/index.html`
- Locate the `java.awt` package

Viewing Package Information

Investigate the AWT package by viewing the Java API Documentation. As discussed in "Java Programming Language Overview," you can view this documentation by loading the `<JAVA BASE DIRECTORY>/api/docs/index.html` in a web browser. This will show you the documentation on all of the Java API classes. In Figure B-1, look for the package called `java.awt`. Click on the `java.awt` package in the upper left corner to get a list in the bottom left area of the figure showing all the classes in this package.

Throughout this chapter, you will use many of these classes. When using the API documentation, click on these links to get more information about how the classes work and how to use them.

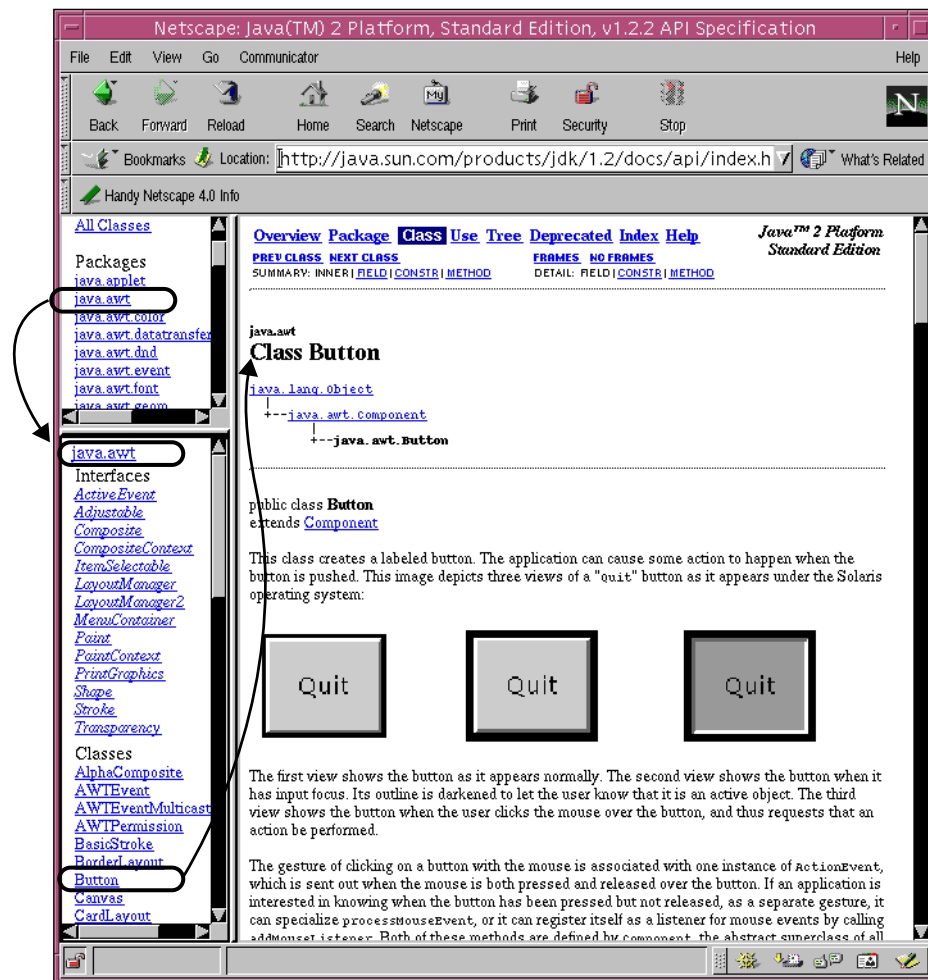


Figure B-1 Using the API Documentation to View Package and Class Information



Sun Educational Services

The java.awt Package Class Hierarchy

- Every GUI component is a subclass of `Component`.
- `Container` defines a kind of component that other components can be nested inside
- Two `Container` subclasses: `Frame` and `Panel`

The java.awt Package Class Hierarchy

Before you begin learning how to write GUIs, it is helpful to consider the hierarchy of classes in the AWT. Every GUI component that appears on the screen is a subclass of (inherits from) the abstract class `Component`.

The component class defines a number of methods and instance variables that are inherited by all components. This gives all components a set of common behaviors and attributes.

`Container` is an abstract subclass that defines a kind of component that allows other components to be nested inside of it. You will see two kinds of subclasses of container called `Frame` and `Panel`. `Container` is a subclass of component; a container “is a” component.

Figure B-2 is an overview of the AWT package. The classes in bold are the primary focus of this appendix.

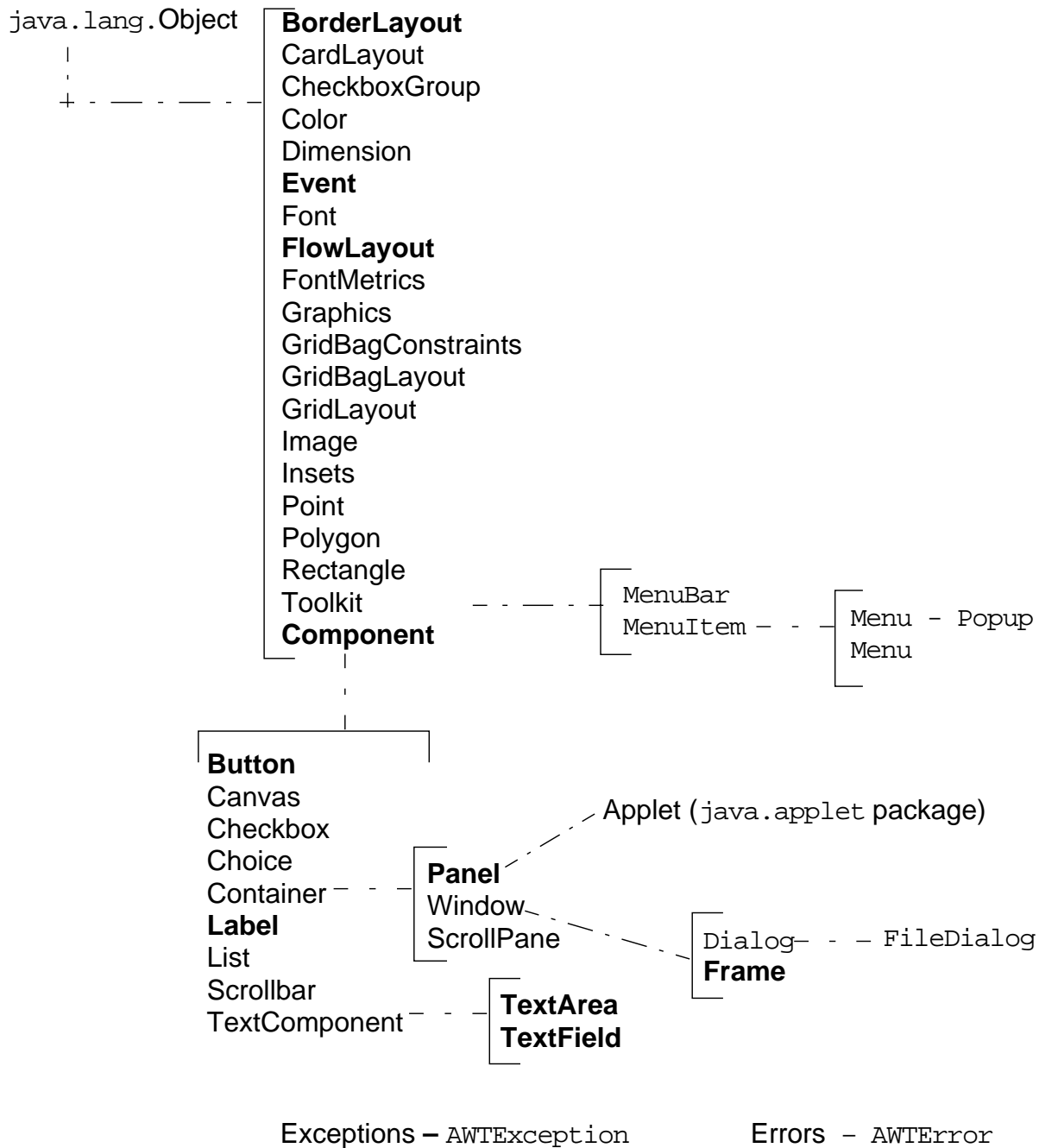


Figure B-2 `java.awt` Package

GUI Project

The concepts in this appendix will be presented with a project, developed incrementally. The project is an application called Sticky Pad. It allows you to leave a note on your screen when you leave the office and allows co-workers to leave notes back to you. They might then leave notes about what they needed to tell you.

Figure B-3 shows how the final product should look.

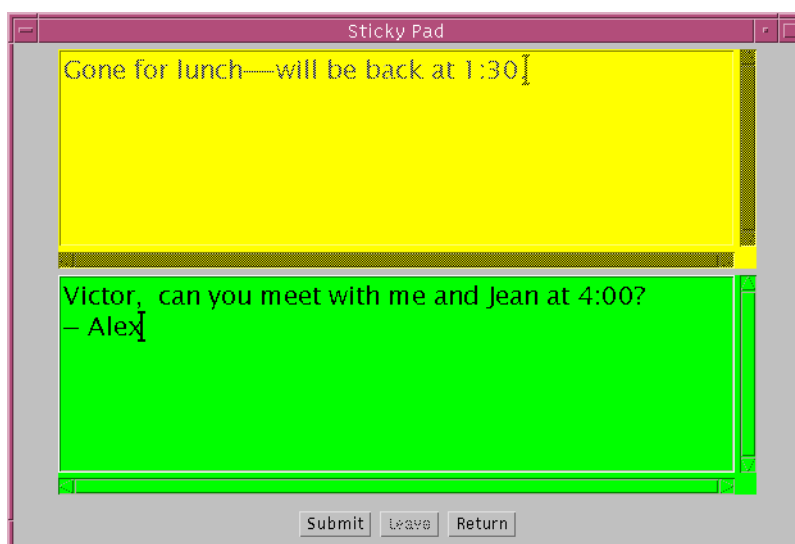
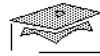


Figure B-3 Sticky Pad Application



Sun Educational Services

Frames

- The first thing you need is a `Frame`, a window with a title bar and resizing corners.
- Use `import java.awt.*` to access the graphics classes, including `Frame`.
- Set the size of the frame and make it visible when making a `Frame`.

Frames

To make a GUI in an application, first make a `Frame` to put the GUI in; it will stand on its own on your screen.

Start the program file with `import java.awt.*` to tell the compiler to look in the `java.awt` package to find the graphics classes you are using.

The class `Frame` in the `java.awt` package is a window with a title bar and resizing corners. Once you make the frame, you can add your components to the frame. The `Frame` class has a constructor that takes a `String` that appears in the title bar of the frame. When you make a frame, you must set the size of the frame and make it visible (call the `show` method of the frame). If you do not do this, you will not see your frame.

The following code makes the frame for the Sticky Pad.


```
1 import java.awt.*;
2 class StickyPad
3 {
4     private Frame frm;
5
6     public StickyPad()
7     {
8         frm = new Frame("Sticky Pad");
9         frm.setSize(500,420);
10        frm.show();
11    }
12
13    public static void main(String args[])
14    {
15        StickyPad sp = new StickyPad();
16    }
17 }
```

The code creates the GUI shown in Figure B-4. This illustration shows how it would appear in the Solaris Operating Environment.



Figure B-4 StickyPad Frame

If you run this program, you will not be able to close the frame. You will add that code later. For now, you must use Ctrl + C to terminate the program from the command line in the window from which you launched the application. Later in the appendix, you will learn how to close the frame.

Notice that this example is written in nine lines of code, not including lines only including braces. This frame can be drawn, minimized, maximized, resized, and moved. Consider how much code was written in the AWT classes to allow you to do this so easily. This is the beauty of object-oriented programming and code reuse. Someone else writes a set of classes with the complex logic encapsulated in private methods and variables and provides a simple public interface to the class for you to use.



Sun Educational Services

Adding a Button

- Make an object of the `Button` class.
- Always check the Java API documentation to see how a class works.
- Check for constructors.
- To add any component to your frame:
 - ▼ Call the `add` method of the frame.
 - ▼ Pass the component as the parameter.

Adding a Button

Once you have a frame to put your GUI in, begin to add components such as buttons, labels, and text fields to your frame. Each component has its own class in the `java.awt` package. For example, to make a button, simply make an object of the `Button` class.

Before you use any class for the first time, check the Java API documentation to study how it works and how it should be used. Look for the constructors available for the class. One of the constructors for a button takes a `String` representing the caption that will appear on the button. To add any component to your frame, call the `add` method of the frame, and pass the component as the argument.

The following code adds one button to the frame from the last example. The new code is in bold.

```
1 import java.awt.*;
2
3 class StickyPad
4 {
5     private Frame frm;
6
7     public StickyPad()
8     {
9         frm = new Frame("Sticky Pad");
10         Button submitBtn = new Button("Submit"); // specifies that
11                                                    // submit appears
12         frm.add(submitBtn);                      // on the button;
13
14         frm.setSize(500,420);
15         frm.show();
16     }
17
18     public static void main(String args[])
19     {
20         StickyPad sp = new StickyPad();
21     }
22 }
```

Figure B-5 shows what this code looks like onscreen in the Solaris operating environment.

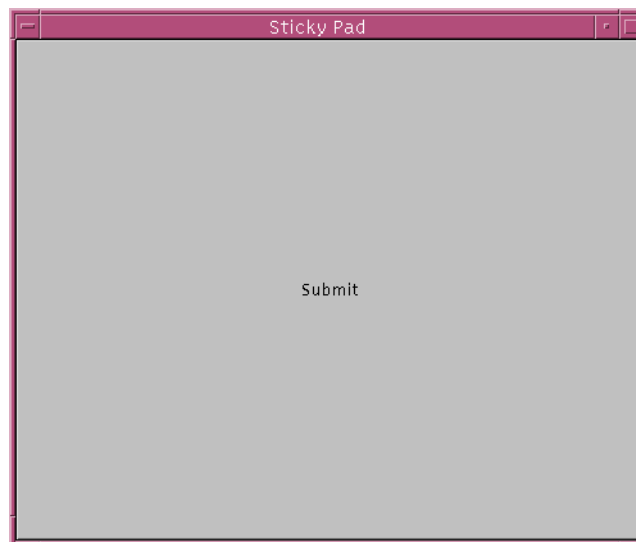


Figure B-5 Frame With Submit Button



Adding a Button

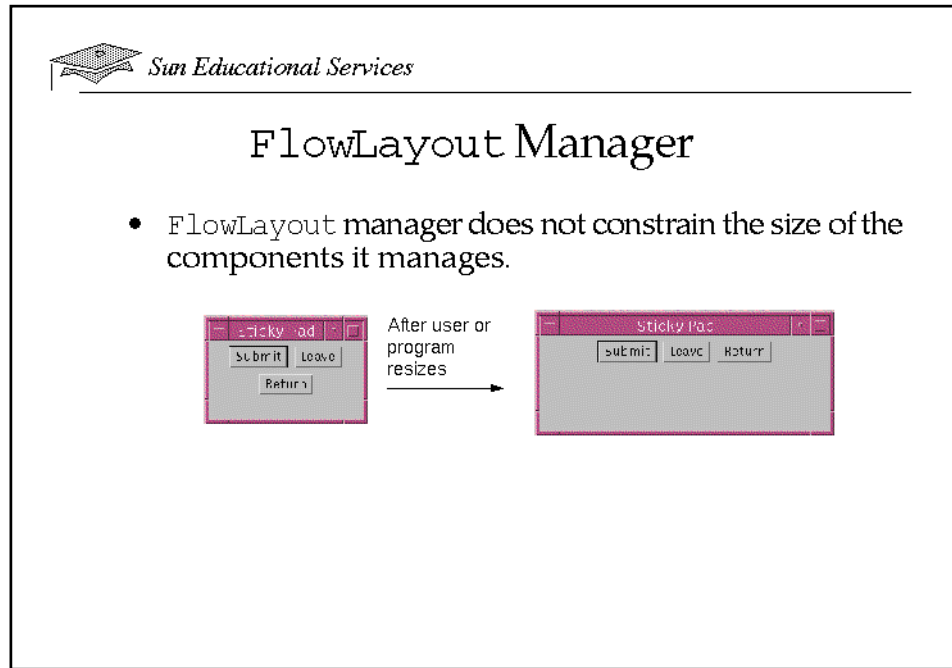
- Button expands to the size of the entire frame.
- Code does not specify the size or location of the button.
- What happens if you hard-code sizes and locations?
- AWT has `FlowLayout`, `BorderLayout`, `GridLayout`, `CardLayout`, and `GridBagLayout`.

This code creates a button; however, the button expands to the size of the entire frame. Notice that the code does not specify the size or location of the button. In most languages, you specify the specific location and size of all your components. However, this can be very time consuming and does not provide the kind of adaptability that Java applications should have.

Consider what happens if you hard code exact sizes and locations for each component. What happens when the user is running on a very small monitor with low resolution? What happens when the user has very high resolution and tries to stretch your GUI? What happens when the user is on a platform with different window systems and font sizes? In all these cases that GUI will probably not adapt well.

In Java technology, rather than hard-coding the sizes and locations, use layout managers to control the sizing and locations of the components. You can choose a predefined layout manager that organizes the layout in a way that you prefer. Then the layout manager adjusts the sizing and locations of the components based on the space available.

AWT has five layout managers: `FlowLayout`, `BorderLayout`, `GridLayout`, `CardLayout`, and `GridBagLayout`. This appendix covers the first two.



FlowLayout *Manager*

The FlowLayout manager is the simplest layout manager. It displays components in the order they were added from left to right. When it runs out of room on a line, it displays the next components on the next line.

Unlike other layout managers, the FlowLayout manager does not constrain the size of the components it manages, but instead allows them to have their preferred size.

When the area being managed by a FlowLayout is resized by the user, the layout can change. See Figure B-6.

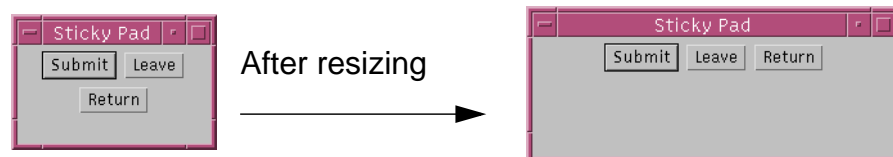


Figure B-6 FlowLayout Resizing

The following constructs and installs a new `FlowLayout` with centered alignment and a default five-unit horizontal and vertical gap:

```
frm.setLayout(new FlowLayout());
```

For the sticky pad, there should be three buttons with the captions of "Submit", "Leave", and "Return." The following code will make those buttons.

```
1 import java.awt.*;
2
3 class StickyPad
4 {
5     private Frame frm;
6     private Button submitBtn, leaveBtn, returnBtn;
7
8     public StickyPad()
9     {
10         frm = new Frame("Sticky Pad");
11         frm.setLayout(new FlowLayout());
12
13         submitBtn = new Button("Submit");
14         frm.add(submitBtn);
15         leaveBtn = new Button("Leave");
16         frm.add(leaveBtn);
17         returnBtn = new Button("Return");
18         frm.add(returnBtn);
19
20         frm.setSize(500,420);
21         frm.show();
22     }
23
24     public static void main(String args[])
25     {
26         StickyPad sp = new StickyPad();
27     }
28 }
```

Figure B-7 shows what the code looks like when run.



Figure B-7 Frame With Three Buttons

You have not yet specified how to make the buttons do something when you click them. You will add this functionality after you complete the layout.



Sun Educational Services

BorderLayout Manager

- Provides a more complex scheme for placing your components within a container
- Contains NORTH, SOUTH, EAST, WEST, and CENTER
- The following line constructs and installs a new BorderLayout with no gaps between the components:

```
frm.setLayout(new BorderLayout());
```

BorderLayout *Manager*

The BorderLayout manager provides a more complex scheme for placing your components within a container. It is the default layout manager for Frame. The BorderLayout manager contains five distinct areas: NORTH, SOUTH, EAST, WEST, and CENTER, indicated by BorderLayout.NORTH, and so on.

NORTH occupies the top of a frame, EAST occupies the right side, and so on. The CENTER area represents everything left over once the NORTH, SOUTH, EAST, and WEST areas are filled. When the Window is stretched vertically, the EAST, WEST and CENTER regions are stretched, whereas when the window is stretched horizontally, the NORTH, SOUTH, and CENTER regions are stretched.

Note – The relative positions of the buttons do not change as the window is resized, but the sizes of the buttons do change.



BorderLayout Manager

- Add components to named regions in the BorderLayout manager.
- NORTH, SOUTH, and CENTER regions change if the window is resized horizontally.
- EAST, WEST, and CENTER regions change if the window is resized vertically.
- An unused NSEW region behaves as if its preferred size were zero by zero.

The following line:

```
frm.setLayout(new BorderLayout());
```

constructs and installs a new BorderLayout with no gaps between the components.

You must add components to named regions in the BorderLayout manager; otherwise, they will be put in the center. You could add a component to the north region with the following line of code:

```
frm.add(myBtn, BorderLayout.NORTH);
```

Use a BorderLayout manager to produce layouts with elements that stretch in one direction, the other direction, or both, when resized.

Note – NORTH, SOUTH, and CENTER regions change if the window is resized horizontally, and EAST, WEST, and CENTER regions change if the window is resized vertically.

If you leave a NSEW region of a BorderLayout unused, it behaves as if its preferred size were zero by zero. The CENTER region still appears as background in the middle, even if it contains no components.

Only a single component can be added to each of the five regions of the BorderLayout manager. If you try to add more than one, only the last one added is visible. A later example shows how you can use intermediate containers to allow more than one component to be laid out in the space of a single BorderLayout manager region.

Note – The layout manager honors the preferred height of the NORTH and SOUTH components, but forces them to be as wide as the container. In the case of the EAST and WEST components, the preferred width is honored and the height is constrained.

This code adds one button to each region.

```
1  import java.awt.*;
2
3  class BorderEx
4  {
5      private Frame frm;
6      private Button bn, bs, bw, be, bc;
7
8      public BorderEx()
9      {
10         frm = new Frame("Border Layout");
11
12         bn = new Button("B1");
13         bs = new Button("B2");
14         bw = new Button("B3");
15         be = new Button("B4");
16         bc = new Button("B5");
17         frm.add(bn, BorderLayout.NORTH);
18         frm.add(bs, BorderLayout.SOUTH);
19         frm.add(bw, BorderLayout.WEST);
20         frm.add(be, BorderLayout.EAST);
21         frm.add(bc, BorderLayout.CENTER);
22
23         frm.setSize(200,200);
24         frm.show();
25     }
26
27     public static void main(String args[])
28     {
29         BorderEx guiWindow2 = new BorderEx();
30     }
31 }
```

Figure B-8 demonstrates how the code in the previous example looks, when resizing.

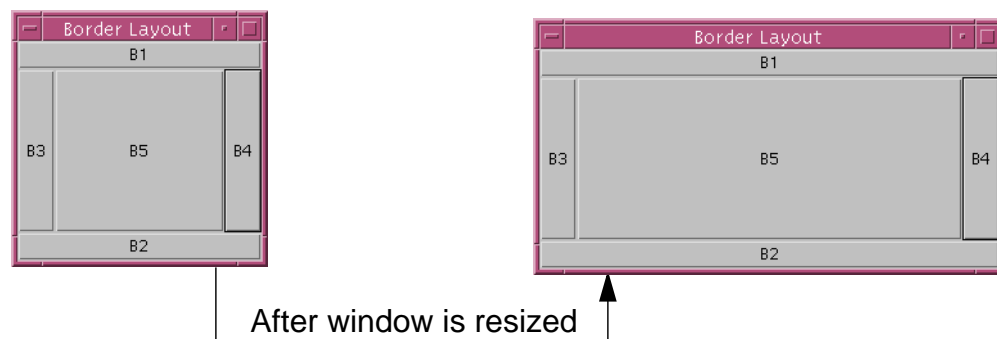


Figure B-8 Example of BorderLayout



Creating Panels and Complex Layouts

- Most GUI layouts are composed of several nested layouts.
- How do you get a text field to span the four columns?
- Stretch component to use the whole width but not stretch vertically.

Creating Panels and Complex Layouts

Layout managers are extremely powerful and effective tools. However, it would be difficult to design a very complex GUI layout with any one of the layout managers. Most GUI layouts are composed of several nested layouts. For instance, Figure B-3 on page B-9 shows what you would like the Sticky Pad GUI to look like. On the top is the text area where you can type your outgoing message. In the middle is the text area where your visitors can type messages to you. At the bottom, are the buttons to control the sticky pad. Using a simple flow layout, you could not keep the buttons nicely organized at the bottom of the frame. With a simple border layout you could not put multiple components in the South.

A component called a `Panel` solves the problem. `Panel`, like `Frame`, provides a space for you to attach any GUI component, including other panels. Each panel can have its own layout manager. Since, a panel is a component, it can be added to any other frame or panel. Thus, you can put the buttons onto a panel, then put that panel onto the southern region of the frame. You will put the text fields onto another panel and add that panel to the center region of the frame. The example on B-26 puts the whole GUI together.

Note – To make the text area, the example simply makes an object of the `TextArea` class and provides the width and height to its constructor.

Figure B-9 shows what the code on page B-26 looks like onscreen, with appropriate text added.

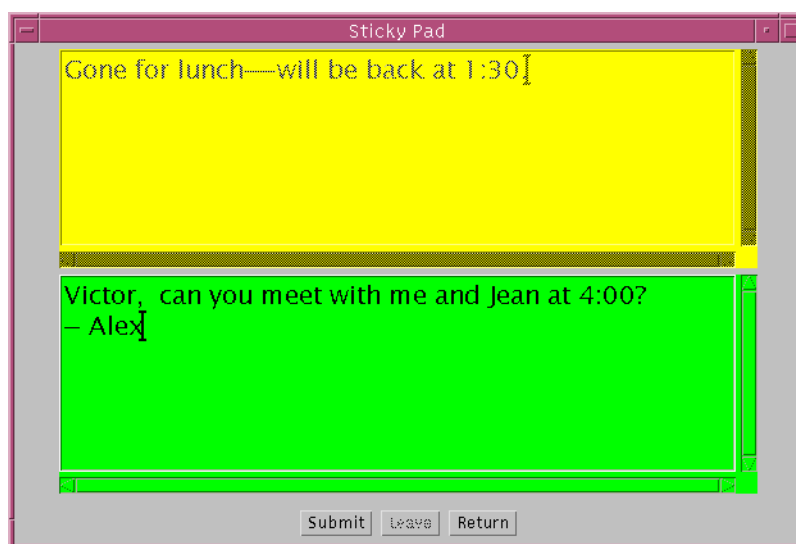


Figure B-9 Sticky Pad

```
1 import java.awt.*;
2
3 class StickyPad
4 {
5     private Frame frm;
6     private TextArea noteTA, responseTA;
7     private Button leaveBtn, returnBtn, submitBtn;
8
9     public StickyPad()
10    {
11        frm = new Frame("Sticky Pad");
12
13        Panel centerPnl = new Panel();
14        centerPnl.setLayout(new FlowLayout());
15
16        noteTA = new TextArea(6,40);
17        //set the color of the text area to yellow:
18        noteTA.setBackground(Color.yellow);
19        //set the font of the text area to a large 20 point font
20        noteTA.setFont(new Font("Dialog", Font.PLAIN, 20));
21        centerPnl.add(noteTA);
22
23        responseTA = new TextArea(6,40);
24        //set the color of the text area to green:
25        responseTA.setBackground(Color.green);
26        //set the font of the text area to a large 20 point font
27        responseTA.setFont(new Font("Dialog", Font.PLAIN, 20));
28        centerPnl.add(responseTA);
29
30        Panel southernPnl = new Panel();
31        southernPnl.setLayout(new FlowLayout());
32
33        submitBtn = new Button("Submit");
34        southernPnl.add(submitBtn);
35
36        leaveBtn = new Button("Leave");
37        southernPnl.add(leaveBtn);
38
39        returnBtn = new Button("Return");
40        southernPnl.add(returnBtn);
41
42        frm.add(centerPnl, BorderLayout.CENTER);
43        frm.add(southernPnl, BorderLayout.SOUTH);
44        frm.setSize(500, 420);
45        frm.show();
46    }
```



```
47
48 public static void main(String args[])
49 {
50     StickyPad sp = new StickyPad();
51 }
52 }
```



Event Handling

- Events are actions in the user interface.
- In event handling, program listens for these events in order to respond.
- An event source is an object that creates an event object.
- An event listener is an object that is registered with the button to be notified of certain kinds of events.
- Classes are in the `java.awt.event` package.

Event Handling

Now that you have successfully created the layout of the GUI, you need to learn how to listen to user input. Events are actions in the user interface, such as the mouse being clicked or a key being pressed. Event handling is when the program listens for these events in order to respond to the user.

An event source is an object that creates an event object. For instance, if you click a button on the GUI, that button is an event source that generates an object of the `ActionEvent` class. An event listener is an object that is registered with the button to be notified of certain kinds of events. When the event listener is notified that an event occurred, it executes appropriate code to respond. For example, you could make your application class an `ActionListener` that is registered to be notified when a certain button is clicked. You could then write code to perform whatever actions you desire when that button is pressed.

The classes used for event handling in AWT are in the `java.awt.event` package. There are numerous classes listed in the API documentation, used to handle different kinds of events. This section focuses on how to respond to a button being pressed and the window close button being selected. However, the pattern of responding to events is common to all kinds of events. Thus, if later you wanted to respond to keyboard input, scrollbars being moved, check boxes being checked, or any other event, you will have no problem learning it. You will already understand the concepts involved, you just have to study the classes.



Listening to a Button

To apply this event handling mechanism to listening to a button being clicked:

1. Implement the `ActionListener` interface.
2. Write the `actionPerformed(ActionEvent e)` method.
3. Tell the `Button` that you want to be told when the click takes place.

```
b.addActionListener(bh);
```

Listening to a Button

You will apply this event handling mechanism to listening to a button being pressed. There are three major steps to accomplish this:

1. Implement the `ActionListener` interface.
2. Write the `actionPerformed(ActionEvent e)` method, which is specified in the `ActionListener` interface. This method will be called by the button whenever it is clicked. Thus, you will write code in this method to perform the actions that should happen when the user clicks the button.
3. Tell the `Button` that you want to be told when the click takes place. To do so, register the object as an `ActionListener` for this button. Do this by passing a reference to this object to the `addActionListener` method of the button.

```
b.addActionListener(bh);
```

The following example implements the behavior for the submit button. When you click the submit button it will add the text in the second text area to a string variable and clears the text area for the next user. The other two buttons still do nothing.

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 class StickyPad implements ActionListener
5 {
6     private Frame frm;
7     private TextArea noteTA, responseTA;
8     private Button leaveBtn, returnBtn, submitBtn;
9     private String savedMessages;
10
11     public StickyPad()
12     {
13         savedMessages = "";
14         frm = new Frame("Sticky Pad");
15
16         Panel centerPnl = new Panel();
17         centerPnl.setLayout(new FlowLayout());
18
19         noteTA = new TextArea(6,40);
20         //set the color of the text area to yellow:
21         noteTA.setBackground(Color.yellow);
22         //set the font of the text area to a large 20 point font
23         noteTA.setFont(new Font("Dialog", Font.PLAIN, 20));
24         centerPnl.add(noteTA);
25
26         responseTA = new TextArea(6,40);
27         //set the color of the text area to green:
28         responseTA.setBackground(Color.green);
29         //set the font of the text area to a large 20 point font
30         responseTA.setFont(new Font("Dialog", Font.PLAIN, 20));
31         centerPnl.add(responseTA);
32
33         Panel southernPnl = new Panel();
34         southernPnl.setLayout(new FlowLayout());
35
36         submitBtn = new Button("Submit");
37         southernPnl.add(submitBtn);
38         submitBtn.addActionListener(this);
39
40         leaveBtn = new Button("Leave");
41         southernPnl.add(leaveBtn);
```

```
42
43     returnBtn = new Button("Return");
44     southernPnl.add(returnBtn);
45
46     frm.add(centerPnl, BorderLayout.CENTER);
47     frm.add(southernPnl, BorderLayout.SOUTH);
48     frm.setSize(500, 420);
49     frm.show();
50 }
51 public void actionPerformed(ActionEvent e)
52 {
53     //The submit button was pressed
54     savedMessages += responseTA.getText() + "\n";
55     responseTA.setText("");
56 }
57
58 public static void main(String args[])
59 {
60     StickyPad sp = new StickyPad();
61 }
62 }
```



Listening to Multiple Buttons

- With multiple buttons, how do you know which one the user clicks?
- The `ActionEvent` class has methods to give information about the event that occurred.
- One of its methods is `getActionCommand`.

Listening to Multiple Buttons

In the last example, you listened to only one button. Thus, when the `actionPerformed` method was executed, you knew that button had been pressed. However, what if you listened to all three buttons, how would you know which button was pressed? There are several approaches to solving this problem. Some solutions are more robust, but this section looks at one of the simplest solutions.

Notice that the parameter of the `actionPerformed` method is an `ActionEvent` object, called `e`. The `ActionEvent` class has methods to give information about the event that occurred. One of its methods is `getActionCommand`. This method tells you the caption of the button that was pressed. Thus, if you have several buttons on your GUI, in your `actionPerformed` method, you can compare the result of the `getActionCommand` method to strings of the button captions to decide which was pressed.

That `ActionEvent` code is added to the previous example and shown here; the new code is in bold.

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 class StickyPad implements ActionListener
5 {
6     private Frame frm;
7     private TextArea noteTA, responseTA;
8     private Button leaveBtn, returnBtn, submitBtn;
9     private String savedMessages;
10
11     public StickyPad()
12     {
13         savedMessages = "";
14         frm = new Frame("Sticky Pad");
15
16         Panel centerPnl = new Panel();
17         centerPnl.setLayout(new FlowLayout());
18
19         noteTA = new TextArea(6,40);
20         //set the color of the text area to yellow:
21         noteTA.setBackground(Color.yellow);
22         //set the font of the text area to a large 20 point font
23         noteTA.setFont(new Font("Dialog", Font.PLAIN, 20));
24         centerPnl.add(noteTA);
25
26         responseTA = new TextArea(6,40);
27         //set the color of the text area to green:
28         responseTA.setBackground(Color.green);
29         //set the font of the text area to a large 20 point font
30         responseTA.setFont(new Font("Dialog", Font.PLAIN, 20));
31         centerPnl.add(responseTA);
32
33         Panel southernPnl = new Panel();
34         southernPnl.setLayout(new FlowLayout());
35
36         submitBtn = new Button("Submit");
37         southernPnl.add(submitBtn);
38         submitBtn.addActionListener(this);
39         submitBtn.setEnabled(false);
40
41         leaveBtn = new Button("Leave");
42         southernPnl.add(leaveBtn);
43         leaveBtn.addActionListener(this);
```



```
44
45     returnBtn = new Button("Return");
46     southernPnl.add(returnBtn);
47     returnBtn.addActionListener(this);
48     returnBtn.setEnabled(false);
49
50     frm.add(centerPnl, BorderLayout.CENTER);
51     frm.add(southernPnl, BorderLayout.SOUTH);
52     frm.setSize(500, 420);
53     frm.show();
54 }
55 public void actionPerformed(ActionEvent e)
56 {
57     if (e.getActionCommand().equals("Leave"))
58     {
59         //Disable the outgoing text area so it can
60         //not be changed and enable the response text
61         //area and buttons
62         noteTA.setEnabled(false);
63         leaveBtn.setEnabled(false);
64         returnBtn.setEnabled(true);
65         submitBtn.setEnabled(true);
66     }
67     else if (e.getActionCommand().equals("Submit"))
68     {
69         //Save the message and clear the response text
70         //area for the next visitor
71         savedMessages += responseTA.getText() + "\n";
72         responseTA.setText("");
73     }
74     else if (e.getActionCommand().equals("Return"))
75     {
76         //Disable the response text area and buttons and
77         //display the save messages
78         responseTA.setText(savedMessages);
79         submitBtn.setEnabled(false);
80         returnBtn.setEnabled(false);
81     }
82 }
83
84 public static void main(String args[])
85 {
86     StickyPad sp = new StickyPad();
87 }
88 }
```



Closing the Window

1. Implement the `WindowListener` interface.
2. Call the `addWindowListener` method of the frame.
3. Write code for all of the methods in the `WindowListener` interface.

Closing the Window

If you have attempted to run any of the examples so far, you have noticed that the window close icon does not work. In order, to make it work, add event handling to listen to the close icon. This event handling follows the general pattern of all event handling. Do three things:

1. Implement the `WindowListener` interface.
2. Call the `addWindowListener` method of the frame.
3. Write code for all of the methods in the `WindowListener` interface including the `windowClosing` method.



Closing the Window

- Seven methods for the interface:
 - ▼ `public void windowActivated(WindowEvent e)`
 - ▼ `public void windowClosed(WindowEvent e)`
 - ▼ `public void windowClosing(WindowEvent e)`
 - ▼ `public void windowDeactivated(WindowEvent e)`
 - ▼ `public void windowDeiconified(WindowEvent e)`
 - ▼ `public void windowIconified(WindowEvent e)`
 - ▼ `public void windowOpened(WindowEvent e)`
- When you implement an interface, you must write all of the methods in that interface.

There is only one small catch: If you view the API documentation for the `WindowListener` interface, you will see that there are seven methods in this interface. The methods are:

```
public void windowActivated(WindowEvent e)
public void windowClosed(WindowEvent e)
public void windowClosing(WindowEvent e)
public void windowDeactivated(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowOpened(WindowEvent e)
```

Each of these methods is called when different events occur on the frame. Currently, the only window event that is important at this point is `windowClosing`. However, when you implement an interface, you must write all of the methods in that interface. Thus, write empty methods for the other six.

Table B-1 shows the parallels with listening to buttons.

Table B-1 Button and Window Parallels

	Button	Window
Implement	ActionListener	WindowListener
Add	addActionListener (this)	addWindowListener(this)
Methods	public void actionPerformed (ActionEvent e)	public void windowOpened(WindowEvent e) public void windowClosing(WindowEvent e) public void windowClosed(WindowEvent e) public void windowIconified(WindowEvent e) public void windowDeiconified(WindowEvent e) public void windowActivated(WindowEvent e) public void windowDeactivated(WindowEvent e)

This example adds the code to end the application when the close icon is pressed. The quickest way to end an application is with the line: `System.exit(0)`; The new code is in bold.

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 class StickyPad implements ActionListener, WindowListener
5 {
6     private Frame frm;
7     private TextArea noteTA, responseTA;
8     private Button leaveBtn, returnBtn, submitBtn;
9     private String savedMessages;
10
11     public StickyPad()
12     {
13         savedMessages = "";
14         frm = new Frame("Sticky Pad");
15
16         Panel centerPnl = new Panel();
17         centerPnl.setLayout(new FlowLayout());
18
19         noteTA = new TextArea(6,40);
20         //set the color of the text area to yellow:
21         noteTA.setBackground(Color.yellow);
22         //set the font of the text area to a large 20 point font
23         noteTA.setFont(new Font("Dialog", Font.PLAIN, 20));
24         centerPnl.add(noteTA);
25
26         responseTA = new TextArea(6,40);
27         //set the color of the text area to green:
28         responseTA.setBackground(Color.green);
29         //set the font of the text area to a large 20 point font
30         responseTA.setFont(new Font("Dialog", Font.PLAIN, 20));
31         centerPnl.add(responseTA);
32
33         Panel southernPnl = new Panel();
34         southernPnl.setLayout(new FlowLayout());
35
36         submitBtn = new Button("Submit");
37         southernPnl.add(submitBtn);
38         submitBtn.addActionListener(this);
39         submitBtn.setEnabled(false);
40
41         leaveBtn = new Button("Leave");
42         southernPnl.add(leaveBtn);
```

```
43     leaveBtn.addActionListener(this);
44
45     returnBtn = new Button("Return");
46     southernPnl.add(returnBtn);
47     returnBtn.addActionListener(this);
48     returnBtn.setEnabled(false);
49
50     frm.add(centerPnl, BorderLayout.CENTER);
51     frm.add(southernPnl, BorderLayout.SOUTH);
52     frm.setSize(500, 420);
53     frm.show();
54     frm.addWindowListener(this);
55 }
56 public void windowClosing(WindowEvent e)
57 {
58     System.exit(0);
59 }
60
61 public void windowOpened(WindowEvent e) {}
62 public void windowClosed(WindowEvent e) {}
63 public void windowIconified(WindowEvent e) {}
64 public void windowDeiconified(WindowEvent e) {}
65 public void windowActivated(WindowEvent e) {}
66 public void windowDeactivated(WindowEvent e) {}
67
68 public void actionPerformed(ActionEvent e)
69 {
70     if (e.getActionCommand().equals("Leave"))
71     {
72         //Disable the outgoing text area so it can
73         //not be changed and enable the response text
74         //area and buttons
75         noteTA.setEnabled(false);
76         leaveBtn.setEnabled(false);
77         returnBtn.setEnabled(true);
78         submitBtn.setEnabled(true);
79     }
80     else if (e.getActionCommand().equals("Submit"))
81     {
82         //Save the message and clear the response text
83         //area for the next visitor
84         savedMessages += responseTA.getText() + "\n";
85         responseTA.setText("");
86     }
87     else if (e.getActionCommand().equals("Return"))
88     {
```

```
89         //Disable the response text area and buttons and
90         //display the save messages
91         responseTA.setText(savedMessages);
92         submitBtn.setEnabled(false);
93         returnBtn.setEnabled(false);
94     }
95 }
96
97 public static void main(String args[])
98 {
99     StickyPad sp = new StickyPad();
100 }
101}
```



Sun Educational Services

More Components and Events

1. Read the API documentation for appropriate class in `java.awt` package.

Note constructors and methods.

2. Follow the same pattern used to create buttons and text fields.

Note `TextArea` and `Label` components.

3. To learn other kinds of event handling, study `java.awt.event` package.

More Components and Events

In this appendix, you have learned how to use buttons and text fields. More importantly, you have seen the patterns for how to make any GUI component. There are many other GUI components including text areas, labels, and check boxes. To use these and other components, do the following:

1. Read the API documentation for the appropriate class (they will all be in the `java.awt` package). Pay particular attention to the constructors and methods.
2. Follow the same pattern that you used to create buttons and text fields.
3. To learn other kinds of event handling, study the classes in the `java.awt.event` package.

You have also learned how to handle button and window events. More importantly, you have learned the pattern for handling events of any kind. The lab does not require any additional event handling beyond buttons and windows. However, if you want to learn other kinds of event handling, study the classes in `java.awt.event` package and follow the same pattern you used to respond to button and window events.

Exercise: Graphical User Interface Development



Exercise objective – Implement the layout and event concepts in this appendix by creating a simple GUI.

Tasks

Follow the instructions provided by your instructor to locate and complete the exercise for this appendix.

1. Write a program that allows students to enter a number in a text field. Create a button that increments the number by one each time the user clicks it. Create another button that decrements the number by one each time the user clicks it.
2. Create a program that allows a user to enter a `String` type and `int` ID for a shirt object. Create two text fields for data entry and a button that assigns the values to a shirt object.
3. Consider ways that you could enhance the application. You could investigate adding labels to the program that identify the purpose of the text areas and make the program more user friendly.

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Check that you are able to accomplish or answer the following:

- ☐ Describe the AWT package and its components
- ☐ Define the terms containers, components and layout managers, and how they work together to build a graphical user interface
- ☐ Use the `FlowLayout` and `BorderLayout` managers to achieve a desired dynamic layout
- ☐ Add components to a container
- ☐ Use the `Frame` and `Panel` containers appropriately
- ☐ Describe how complex layouts with nested containers work
- ☐ Use the `Button`, `TextField`, `TextArea`, and `Label` components
- ☐ Describe how to handle events when a button is clicked

Think Beyond

Are you prepared to continue your Java programming language learning? What resources do you need to obtain or learn about before you leave class?

Where Do You Go From Here?



This appendix provides guidance for you to continue your Java technology training and practice.

Getting Ready to Program

Use this section when you leave class in order to get all the files you need to continue programming.

You need to:

- Download Java technology
- Download the API specification
- Set up your computer so it knows where you installed Java technology
- Optional – Download or order a debugger or integrated development environment

Downloading Java Technology

Download the Java 2 Platform, Standard Edition.

<http://java.sun.com/products/>

(You can download any API from this site; it includes development tools, class libraries, and the JVM.)

Note – You can follow the explicit directions below; keep in mind that they might not currently be completely accurate if product names or URLs have changed since publication of this training guide.

1. Go to <http://java.sun.com>
2. Choose Products & APIs from the list on the left.
3. From the Product Shortcuts list, select the Java 2 SDK, Standard Edition option with the newest non-beta release number, such as 1.3.
4. In the Production Releases list, select your platform.
5. In the next page that is displayed, locate the link to download the Java 2 SDK, standard edition. (At print time for this guide, it was included in the Downloads section.)

6. If you are shown a list of different “dot rev” versions of the SDK, choose the most recent version, your platform, and the appropriate language.
7. If you are prompted to register, enter your information according to the instructions; otherwise, log in.
8. In the legal agreement page, click Agree, then click Continue.
9. Select the closest location to download from; click Change if you select a different location.
10. Select the item to download. You can typically download everything in one file, or choose from the list to download component items one at a time.

If you have a perpetual Internet connection, you can access the documentation online:

<http://java.sun.com/j2se/1.3/docs/api/index.html>

11. Enter the directory to download to.

Downloading the Java Technology API Specification

Repeat the previous steps until at step 5, a prompt is listed next to the Standard Edition download link that says “Download Java 2 SDK, v 1.3.0 Documentation” (or a later version). Click that link and follow the subsequent directions as prompted.

Setting up Your Computer to Run Java Programs

You need to set the class path to the location where you installed Java. During the download process, you will encounter a link to a Web site with comprehensive instructions on how to set up the classpath, the regular path, and other potential issues.

Downloading a Development Environment or Debugger

You can obtain *integrated development environments* and *debuggers* to help you with your programming.

- You will be doing all your programming in this class using a basic text file. This is the “purist” way to do programming, but also more difficult. You can get a lot of help from a development environment: it lists classes, color-codes different parts of your code, and a host of other features that vary from one development environment to the next.

There are many development environments, including Forte™ for Java™ which is free from Sun™ Microsystems.

<http://www.forte.com>

Links to other environments are also included on the Java technology download page.

Note – One disadvantage of development environments is that they can generate “messy” code that is not written well, even though it will do what you need it to. Be sure to research any development environment you decide to use; you might also choose to use the development environment without its code generation features.

- A debugger can help you figure out where the errors are in your program, providing line numbers, the ability to run a program line by line, and other features to help you.

References

Use the following resources to learn more about particular topics. In addition to the resources listed, we encourage you to continue to other Sun Educational Services training courses at the following URL:

<http://suned.sun.com/>

Suggested courses include the following:

- SL-275: *Java Programming Language*
- 00-226: *Object-Oriented Application Analysis and Design for Java Technology (UML)*
- SL-285: *Java Programming Language Workshop*

Basic Java Technology

- Farrell, Joyce. 1999. *Java Programming: Comprehensive*. Course Technology.

This is an excellent book for non-programmers; it explains concepts that are glossed over in more advanced books.

- Naughton, Patrick, Herbert Schildt. *Java 2: The Complete Reference*. Osborne McGraw-Hill. 1999.
- Eckel, Bruce. *Thinking in Java*. Prentice Hall Computer Books.
- Deitel and Deitel. *Java: How to Program*. Prentice Hall. 1999.

This is a more advanced book for those who want to “dive into” the Java programming language. It includes many code examples and exercises.

- Java Standard Edition Platform Documentation. Available: <http://java.sun.com/docs/index.html>
- Java API Specification. Available: <http://java.sun.com/j2se/1.3/docs/api/index.html>

Using Applets

- Pew, John. 1998. *Instant Java*. Prentice Hall.

Targeted at web designers, this book describes how to use existing Java applet code, in the book and on the included CD-ROM.

Online Tutorial

- <http://java.sun.com/docs/books/tutorial/>

An excellent self-paced tutorial on a wide variety of topics, this includes many code examples.

Technical White Papers

- Kramer, Douglas, Bill Joy, David Spenhoff. *The Java Platform* white paper. Available:
<http://www.javasoft.com/docs/whitePaper.Platform/CreditsPage.doc.html>

Provides detailed information about Java technology architecture.

- Gosling, James, Henry McGilton. *The Java Language Environment, A White Paper*. Available:
http://www.javasoft.com/docs/language_environment


Provides detailed information about how the Java programming language is related to other languages.

- *The Java Language: An Overview White Paper*. Available:
<http://www.javasoft.com/docs/Overviews/java/java-overview-1.html>

Provides a detailed overview of Java programming language features.

Includes a great deal of information about Java technology, including a tutorial, documentation search, information about products and APIs, and white papers on the Java virtual machine and other products.

Java Keywords

D 

This appendix lists Java keywords.

Keywords

Keywords are special reserved words in the Java programming language that give instructions to the compiler. Keywords make sense to the compiler when they are used in the correct order (called the *syntax* of the language) and can cause errors when used incorrectly.

Java source-code files are, like those of all programming languages, written using actual words and numbers. You cannot use just any words you want from the language when you program—the compiler only understands a few of them (keywords) and it is your responsibility to use those few correctly.


The Java programming language demands that all keywords be written *entirely* in lower case. For example, one of the few keywords is “class”. The compiler would not understand this if you wrote “Class” or “CLASS” or any other representation.

Table 10-1 contains all the Java keywords. While only a few of these will be described during this course, they must not be used as identifiers for classes, methods, variables, and so on.

Table 10-1 Java Technology Keywords

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

Java Naming Conventions

E 

This appendix consolidates the naming conventions noted throughout the course.

Class, Method, and Variable Identifiers

Identifiers are the names you assign to classes, variables, methods, and so on. Every class has an identifier, as does each method and variable. The following rules dictate the content and structure of identifiers:

- The first character of an identifier must be one of the following:
 - ▼ An uppercase letter (A–Z)
 - ▼ A lowercase letter (a–z)
 - ▼ The underscore character (`_`)
 - ▼ The dollar character (`$`)

Class names should start with an uppercase letter; variable and method names should start with a lowercase letter.

- The subsequent characters must be any of the following:
 - ▼ Any character from the previous list
 - ▼ Numeric characters (0–9)

Other characters can be added to this list, including any accented characters from other languages. These are dictated by the Java programming language.

- If you use two or more words in the identifier, begin each subsequent word with an uppercase letter. Do not separate words with underscores, dashes, or other characters.
- Do not use Java keywords. See Appendix D, “Java Keywords.”

Good examples of identifiers include:

- `custID` (variable identifier)
- `isClosed` (boolean variable identifier)
- `play` (method identifier)
- `playMusic` (method identifier)
- `Order` (class identifier)

Java is a *case-sensitive* programming language. Case sensitivity means distinguishing between the upper- and lowercase representations of each alphabetical character. The Java programming language considers two identifiers to be different if only their capitalization differs; if you create a variable called “order” then you cannot refer to it later as “Order.”

Constant Identifiers

Use all capital letters for constant identifiers, and separate words with underscores, as in this constant: `SALES_TAX`.

Navigating the Operating Environment

F 

This appendix lists common commands for navigating the Solaris Operating Environment or DOS, from a character-based command window.

Table F-1 shows commands for each operating environment.

Table F-1 Basic Terminal Window Commands

Command	Solaris Operating Environment	Microsoft Windows
Display the name of the current directory	<code>pwd</code>	<code>cd</code>
List the contents of the current directory	<code>ls</code> <code>ls -l</code> for more detail	<code>dir /w</code> <code>dir</code> for more detail
Change to one directory above current directory	<code>cd ..</code>	<code>cd ..</code>
Change to more than one directory above current directory	<code>cd ../..</code> (repeat ../ as necessary)	<code>cd ../../</code>
Change to a directory below current directory	<code>cd directory</code>	<code>cd directory</code>
Change to a specific directory	<code>cd full_path</code>	<code>cd full_path</code>
Copy a file	<code>cp file destination</code>	<code>copy file destination</code>
Move a file	<code>mv file destination</code>	<code>move file destination</code>
Copy a directory	<code>cp -R directory destination</code>	<code>xcopy /s /e directory destination</code>
Move a directory	<code>mv directory destination</code>	<code>move directory destination</code>
Create a new file	<code>touch name</code>	N/A (use a text editor)
Delete a file	<code>rm file</code>	<code>del file</code>
Create a new directory	<code>mkdir directory</code>	<code>mkdir directory</code>
Delete a directory	<code>rmdir directory</code> (empty directories only) <code>rm -r directory</code> (directories with contents)	<code>del directory</code> <code>deltree directory</code> (directory and all contents below it)
Rename a file	<code>mv oldname newname</code>	<code>ren oldname newname</code>
Rename a directory	<code>mv oldname newname</code>	<code>ren oldname newname</code>
Compile a Java application	<code>javac filename.class</code>	<code>javac filename.class</code>
Run a Java application	<code>java filename</code>	<code>java filename</code>

Solaris Operating Environment Tips

You also can use File Manager (on the task bar). This provides a simple graphical user interface for all of the tasks in the table.

You can create a new program this way:

1. In File Manager, choose File > New.
2. Enter an appropriate name such as `Shirt.java`.
3. Close the file and save it.
4. Double-click the file in File Manager to reopen it.
5. Write the program.

You can even compile and run it by right-clicking the `.java` file and choosing compile, and by double-clicking the `.class` file.

If you choose File > Open Terminal in File Manager, the terminal window will open in the same directory displayed in File Manager.

Copyright 2000 Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, California 94303, Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley 4.3 BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

Sun, Sun Microsystems, the Sun Logo, Solaris, JVM, JDK, Forte, Write Once Run Anywhere, et Java sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays.

Netscape est une marque de Netscape Communications Corporation.

Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

L'accord du gouvernement américain est requis avant l'exportation du produit.

Le système X Window est un produit de X Consortium, Inc.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.



Please
Recycle



Adobe PostScript

