Java SE 7 Fundamentals

Student Guide • Volume 1

D67234GC10
Edition 1.0
September 2011
D74170

ORACLE

Authors

Jill Moritz

Kenneth Somerville

Cindy Church

Technical Contributors and Reviewers

Mike Williams

Tom McGinn

Matt Heimer

Joe Darcy

Brian Goetz

Alex Buckley

Adam Messenger

Steve Watts

Editors

Smita Kommini

Richard Wallis

Graphic Designers

Rajiv Chandrabhanu

Maheshwari Krishnamurthy

Publishers

Revathi Ramamoorthy

Jayanthy Keshavamurthy

Contents

1 Java SE 7 Fundamentals
Course Objectives 1-2
Schedule 1-5
Facilities in Your Location 1-7
Quiz 1-8

2 Introducing the Java Technology
Objectives 2-2
Topics 2-4
Java's Place in the World 2-5
Java Desktops 2-6
Java Mobile Phones 2-7
Java TV and Card 2-8
The Story of Java 2-9
Key Concepts of the Java Programming Language 2-10
Procedural Programming 2-11
Object-Oriented 2-12
Distributed 2-13
Simple 2-14
Multi-Threaded 2-15
Secure 2-16
Platform-Dependent Programs 2-17
Platform-Independent Programs 2-20
Quiz 2-22
Topics 2-23
Identifying Java Technology Product Groups 2-24
Java SE 2-25
Java EE 2-26
Java ME 2-27
Java Card 2-28
Setting Up the Java Development Environment 2-29
Downloading and Installing the Java JDK 2-30
Examining the Installed Java Development Kit 2-31
Topics 2-32
Using an Integrated Development Environment 2-33

NetBeans IDE Download 2-34
NetBeans IDE and New Project Wizard 2-35
Quiz 2-36
Topics 2-37
Product Life Cycle (PLC) Stages 2-38
Summary 2-40
Practice 2-1: Running a Java Program by Using the Command Line 2-42
Practice 2-2: Running a Java Program by Using the NetBeans IDE 2-43

3 Thinking in Objects
Objectives 3-2
Relevance 3-3
Topics 3-4
Analyzing a Problem by Using Object-Oriented Analysis (OOA) 3-5
Duke's Choice Order Process 3-6
Topics 3-7
Identifying a Problem Domain 3-8
Topics 3-9
Identifying Objects 3-10
Topics 3-13
Additional Criteria for Recognizing Objects 3-14
Possible Objects in the Duke's Choice Case Study 3-16
Topics 3-17
Identifying Object Attributes and Operations 3-18
Object with Another Object as an Attribute 3-19
Possible Attributes and Operations for Objects in the Duke's Choice Case Study 3-20
Topics 3-21
Case Study Solution: Classes 3-22
Case Study Solution: Attributes 3-23
Case Study Solution: Behaviors 3-25
Topics 3-27
Designing Classes 3-28
Class and Resulting Objects 3-29
Modeling Classes 3-30
Using UML-like Modeling 3-32
Quiz 3-33
Summary 3-35
Practice 3.1 Overview: Analyzing a Problem by Using Object-Oriented Analysis 3-36
Practice 3.2 Overview: Designing a Solution 3-37

4 Introducing the Java Language
Objectives 4-2
Topics 4-3
Relevance 4-4
Identifying the Components of a Class 4-5
Structuring Classes 4-6
Symbols Used in Defining a Java Source 4-8
Putting It All Together 4-9
Quiz 4-11
Field Declarations and Assignments 4-12
Comments 4-13
Topics 4-15
Methods 4-16
Topics 4-18
Keywords 4-19
Topics 4-20
Creating and Using a Test Class 4-21
The main Method 4-22
Compiling a Program 4-23
Executing (Testing) a Program 4-24
Compiling and Running a Program by Using an IDE 4-25
Topics 4-26
Avoiding Syntax Problems 4-27
Topics 4-28
Working with an IDE Debugger 4-29
Summary 4-31
Practice 4-1: Viewing and Adding Code to an Existing Java Program 4-32
Practice 4-2: Creating and Compiling a Java Program 4-33
Practice 4-3: Exploring the Debugger 4-34

5 Declaring, Initializing, and Using Variables
Objectives 5-2
Relevance 5-3
Topics 5-4
Identifying Variable Use and Syntax 5-5
Uses of Variables 5-7
Variable Declaration and Initialization 5-8
Topics 5-10
Describing Primitive Data Types 5-11
Integral Primitive Types 5-12

Floating Point Primitive Types 5-14
Textual Primitive Type 5-15
Logical Primitive Type 5-17
Topics 5-18
Naming a Variable 5-19
Assigning a Value to a Variable 5-21
Declaring and Initializing Several Variables in One Line of Code 5-22
Additional Ways to Declare Variables and Assign Values to Variables 5-23
Constants 5-25
Storing Primitives and Constants in Memory 5-26
Quiz 5-27
Topics 5-28
Standard Mathematical Operators 5-29
Increment and Decrement Operators (++ and --) 5-31
Increment and Decrement Operators (++ and —) 5-34
Operator Precedence 5-35
Using Parentheses 5-38
Topics 5-39
Using Promotion and Type Casting 5-40
Promotion 5-42
Type Casting 5-44
Compiler Assumptions for Integral and Floating Point Data Types 5-47
Floating Point Data Types and Assignment 5-49
Example 5-50
Quiz 5-51
Summary 5-52
Practice 5-1: Declaring Field Variables in a Class 5-53
Practice 5-2: Using Operators and Performing Type Casting to Prevent Data Loss 5-54

6 Working with Objects
Objectives 6-2
Topics 6-3
Working with Objects: Introduction 6-4
Accessing Objects by Using a Reference 6-5
The Shirt Class 6-6
Topics 6-7
Working with Object Reference Variables 6-8
Declare and Initialize: Example 6-9
Work with Object References 6-10
Working with Object References 6-11

References to Different Objects 6-12
Different Objects and Their References 6-13
References and Objects In Memory 6-14
Assigning a Reference to Another Reference 6-15
Two References, One Object 6-16
Assigning a Reference to Another Reference 6-17
Quiz 6-18
Topics 6-19
The String Class 6-20
Concatenating Strings 6-21
String Concatenation 6-22
String Method Calls with Primitive Return Values 6-25
String Method Calls with Object Return Values 6-26
Method Calls Requiring Arguments 6-27
Topics 6-28
Java API Documentation 6-29
Java Platform SE 7 Documentation 6-30
Java Platform SE 7: Method Summary 6-32
Java Platform SE 7: Method Detail 6-33
System.out Methods 6-34
Documentation on System.out.println() 6-35
Using print() and println() Methods 6-36
Topics 6-37
StringBuilder Class 6-38
StringBuilder Advantages Over String for Concatenation (or Appending) 6-39
StringBuilder: Declare and Instantiate 6-40
StringBuilder Append 6-41
Quiz 6-42
Summary 6-43
Practice 6-1 Overview: Creating and Manipulating Java Objects 6-44
Practice 6-2 Overview: Using the String and StringBuilder Classes 6-45
Practice 6-3 Overview: Using the Java API Documentation 6-46

7 Using Operators and Decision Constructs
Objectives 7-2
Relevance 7-3
Topics 7-4
Using Relational and Conditional Operators 7-5
Elevator Example 7-6
The ElevatorTest.java File 7-8
Relational Operators 7-9

Testing Equality Between Strings 7-10
Common Conditional Operators 7-11
Topics 7-12
Creating if and if/else Constructs 7-13
The if Construct 7-14
The if Construct: Example 7-15
The if Construct: Output 7-17
Nested if Statements 7-18
The if/else Construct 7-20
The if/else Construct: Example 7-21
The if/else Construct 7-23
Topics 7-24
Chaining if/else Constructs 7-25
Topics 7-27
Using the switch Construct 7-28
Using the switch Construct: Example 7-30
When To Use switch Constructs 7-32
Quiz 7-33
Summary 7-35
Practice 7.1 Overview: Writing a Class That Uses the if/else Statement 7-36
Practice 7.2 Overview: Writing a Class That Uses the switch Statement 7-37

8 Creating and Using Arrays
Objectives 8-2
Topics 8-3
Introduction to Arrays 8-4
One-Dimensional Arrays 8-5
Creating One-Dimensional Arrays 8-6
Array Indices and Length 8-7
Topics 8-8
Declaring a One-Dimensional Array 8-9
Instantiating a One-Dimensional Array 8-10
Initializing a One-Dimensional Array 8-11
Declaring, Instantiating, and Initializing One-Dimensional Arrays 8-12
Accessing a Value Within an Array 8-13
Storing Arrays in Memory 8-14
Storing Arrays of References in Memory 8-15
Quiz 8-16
Topics 8-18
Using the args Array in the main Method 8-19
Converting String Arguments to Other Types 8-20

Topics 8-21
Describing Two-Dimensional Arrays 8-22
Declaring a Two-Dimensional Array 8-23
Instantiating a Two-Dimensional Array 8-24
Initializing a Two-Dimensional Array 8-25
Topics 8-26
The ArrayList Class 8-27
Class Names and the Import Statement 8-28
Working with an ArrayList 8-29
Quiz 8-30
Summary 8-31
Practice 8-1: Creating a Class with a One-Dimensional Array of Primitive Types 8-32
Practice 8-2: Create and Work with an ArrayList 8-33
Practice 8-3: Use Runtime Arguments and Parse the args Array 8-34

9 Using Loop Constructs
Objectives 9-2
Topics 9-3
Loops 9-4
Repeating Behavior 9-5
Creating while Loops 9-6
while Loop in Elevator 9-7
Types of Variables 9-8
while Loop: Example 1 9-9
while Loop: Example 2 9-10
while Loop with Counter 9-11
Topics 9-12
for Loop 9-13
Developing a for Loop 9-14
Topics 9-15
Nested for Loop 9-16
Nested while Loop 9-17
Topics 9-18
Loops and Arrays 9-19
for Loop with Arrays 9-20
Setting Values in an Array 9-21
Enhanced for Loop with Arrays 9-22
Enhanced for Loop with ArrayLists 9-23
Using break with Loops 9-24
Using continue with Loops 9-25

Topics 9-26
Coding a do/while Loop 9-27
Topics 9-29
Comparing Loop Constructs 9-30
Quiz 9-31
Summary 9-33
Practice 9.1 Overview: Writing a Class That Uses a for Loop 9-34
Practice 9.2 Overview: Writing a Class That Uses a while Loop 9-35
Practice 9.3 Overview: Using for Loops to Process an ArrayList 9-36
Practice 9.4 Overview: Writing a Class That Uses a Nested for Loop to Process a Two-Dimensional Array 9-37

10 Working with Methods and Method Overloading
Objectives 10-2
Topics 10-3
Creating and Invoking Methods 10-4
Basic Form of a Method 10-5
Invoking a Method in a Different Class 10-6
Calling and Worker Methods 10-7
Passing Arguments and Returning Values 10-8
Creating a Method with a Parameter 10-9
Creating a Method with a Return Value 10-10
Invoking a Method in the Same Class 10-11
Advantages of Method Use 10-12
Quiz 10-13
Invoking Methods: Summary 10-14
Topics 10-15
Math Utilities 10-16
Static Methods in Math 10-17
Creating static Methods and Variables 10-18
static Variables 10-20
Static Methods and Variables in the Java API 10-21
Topics 10-23
Method Signature 10-24
Method Overloading 10-25
Using Method Overloading 10-26
Method Overloading and the Java API 10-28
Quiz 10-29
Summary 10-30
Practice 10-1: Writing a Method with Arguments and Return Values 10-31
Challenge Practice 10-2: Writing a Class Containing an Overloaded Method 10-32

11 Using Encapsulation and Constructors
Objectives 11-2
Topics 11-3
Overview 11-4
The public Modifier 11-5
Dangers of Accessing a public field 11-6
The private Modifier 11-7
Dangers of Accessing a public field 11-8
The private Modifier on Methods 11-9
Interface and Implementation 11-10
Get and Set Methods 11-11
Using Setter and Getter Methods 11-12
Setter Method with Checking 11-13
Using Setter and Getter Methods 11-14
Topics 11-15
Initializing a Shirt Object 11-16
Constructors 11-17
Creating Constructors 11-18
Initializing a Shirt Object by Using a Constructor 11-20
Default Constructor 11-21
Quiz 11-22
Summary 11-23
Practice 11-1: Implementing Encapsulation in a Class 11-24
Challenge Practice 11-2: Adding Validation to the DateThree Class 11-25
Practice 11-3: Creating Constructors to Initialize Objects 11-26

12 Using Advanced Object-Oriented Concepts
Objectives 12-2
Topics 12-3
Class Hierarchies 12-4
Topics 12-5
Common Behaviors 12-6
Code Duplication 12-7
Inheritance 12-8
Overriding Superclass Methods 12-9
The Clothing Superclass: 1 12-10
The Clothing Superclass: 2 12-11
The Clothing Superclass: 3 12-12
Declaring a Subclass 12-13
Declaring a Subclass (extends, super, and this keywords) 12-14

Declaring a Subclass: 2 12-15
Abstract Classes 12-16
The Abstract Clothing Superclass: 1 12-17
The Abstract Clothing Superclass: 2 12-18
Superclass and Subclass Relationships 12-19
Another Inheritance Example 12-20
Topics 12-21
Superclass Reference Types 12-22
Access to Object Functionality 12-23
Accessing Class Methods from Superclass 12-24
Casting the Reference Type 12-25
Casting 12-26
Polymorphic Method Calls 12-27
Quiz 12-28
Topics 12-29
Multiple Hierarchies 12-30
Interfaces 12-31
Implementing the Returnable Interface 12-32
Access to Object Methods from Interface 12-33
ArrayList 12-34
List Interface 12-35
Topics 12-36
The Object Class 12-37
Calling the toString() Method 12-38
Quiz 12-39
Summary 12-40
Practice 12-1: Creating and Using Superclasses and Subclasses 12-41
Practice 12-2: Using a Java Interface 12-42

13 Handling Errors
Objectives 13-2
Topics 13-3
Reporting Exceptions 13-4
How Exceptions Are Thrown 13-6
Types of Exceptions 13-7
OutOfMemoryError 13-8
Topics 13-9
Method Stack 13-10
Call Stack: Example 13-11
Throwing Throwables 13-12
Working with Exceptions in NetBeans 13-14

Catching an Exception 13-15
Uncaught Exception 13-16
Exception Printed to Console 13-17
Summary of Exception Types 13-18
Quiz 13-19
Topics 13-21
Exceptions in the Java API Documentation 13-22
Calling a Method That Throws an Exception 13-23
Working with a Checked Exception 13-24
Best Practices 13-25
Bad Practices 13-26
Topics 13-28
Multiple Exceptions 13-29
Catching IOException 13-30
Catching IllegalArgumentException 13-31
Catching Remaining Exceptions 13-32
Summary 13-33
Practice 13-1 Overview: Using a try/catch Block to Handle an Exception 13-34
Practice 13-2 Overview: Catching and Throwing a Custom Exception 13-35

14 The Big Picture
Objectives 14-2
Topics 14-3
Packages 14-4
Packages Directory Structure 14-5
Packages in NetBeans 14-6
Packages in Source Code 14-7
Topics 14-8
DukesChoice.jar 14-9
Set Main Class of Project 14-10
Creating the JAR File with NetBeans 14-11
Topics 14-13
Client/Server Two-Tier Architecture 14-14
Client/Server Three-Tier Architecture 14-15
Topics 14-16
The Duke's Choice Application 14-17
The Clothing Class 14-18
The Tiers of Duke's Choice 14-20
Running the JAR File from the Command Line 14-21
Listing Items from the Command Line 14-22
Listing Items in Duke's Choice Web Application 14-23

Topics 14-25
Enhancing the Application 14-26
Adding a New Item for Sale 14-27
Implement Returnable 14-29
Implement Constructor 14-30
The Suit Class: Overriding getDisplay() 14-31
Implement Getters and Setters 14-32
Updating the Applications with the Suit Class 14-33
Testing the Suit Class: Command Line 14-34
Testing the Suit Class: Web Application 14-35
Adding the Suit Class to the Web Application 14-36
Summary 14-37
No Practice for This Lesson 14-38
Course Summary 14-39

Appendix A Java Language Quick Reference

Appendix B UMLet Tips
UML Default Interface B-2

Appendix C Resources
Java on Oracle Technology Network (OTN) C-2
Java SE Downloads C-3
Java Documentation C-4
Java Community C-5
Java Community: Expansive Reach C-6
Java Community: Java.net C-7
Java Technologies C-8
Java Training C-9
Oracle Learning Library C-10

1

Java SE 7 Fundamentals

Course Objectives

After completing this course, you should be able to:

- List and describe several key features of the Java technology, such as that it is object-oriented, multi-threaded, distributed, simple, and secure
- Identify different Java technology groups
- Describe examples of how Java is used in applications, as well as consumer products
- Describe the benefits of using an integrated development environment (IDE)
- Develop classes and describe how to declare a class
- Analyze a business problem in order to recognize objects and operations that form the building blocks of the Java program design

ORACLE

Course Objectives

- Define the term *object* and its relationship to a class
- Demonstrate Java programming syntax
- Write a simple Java program that compiles and runs successfully
- Declare and initialize variables
- List several primitive data types
- Instantiate an object and effectively use object reference variables
- Use operators, loops, and decision constructs
- Declare and instantiate arrays and ArrayLists and be able to iterate through them

ORACLE

Course Objectives

- Use Javadocs to look up Java foundation classes
- Declare a method with arguments and return values
- Use inheritance to declare and define a subclass of an existing superclass
- Describe how errors are handled in a Java program
- Describe how to deploy a simple Java application by using the NetBeans IDE

Schedule

- Day One
  - Lesson 1: Introduction
  - Lesson 2: Introducing the Java Technology
  - Lesson 3: Thinking in Objects
  - Lesson 4: Introducing the Java Language
- Day Two
  - Lesson 5: Working with Primitive Variables
  - Lesson 6: Working with Objects
  - Lesson 7: Using Operators and Decision Constructs (lesson only)

ORACLE

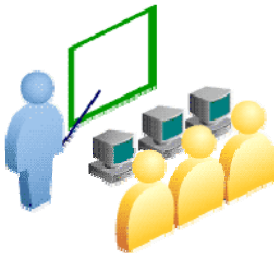
Schedule

- Day Three
  - Lesson 7: Using Operators and Decision Constructs (practices only)
  - Lesson 8: Creating and Using Arrays
  - Lesson 9: Using Loop Constructs
- Day Four
  - Lesson 10: Working with Methods and Method Overloading
  - Lesson 11: Using Encapsulation and Constructors
  - Lesson 12: Introducing Advanced Object-Oriented Concepts
- Day Five
  - Lesson 13: Handling Errors
  - Lesson 14: The Big Picture

ORACLE

Facilities in Your Location

- Enrollment, registration, sign-in
- Badges
- Parking
- Phones
- Internet
- Restrooms
- Labs
- Lunch
- Kitchen/snacks
- Hours
- Materials (paper, pens, and markers)



Quiz

a. What is your name?
b. What do you do for a living, and where do you work?
c. What is the most interesting place you have visited?
d. Why are you interested in Java?

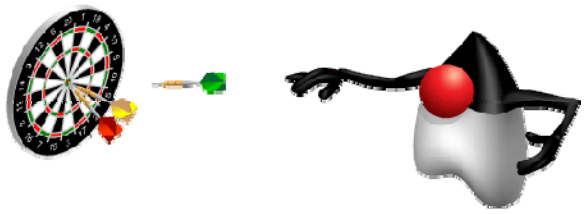
2

Introducing the Java Technology

Objectives

After completing this lesson, you should be able to:

- Describe key features of the Java technology
- Describe and identify features of object-oriented programming
- Discuss the difference between compiled and interpreted languages
- Describe how to download and install the Java Platform
- Describe how to run a Java application by using the command line
- Identify the different Java technologies



ORACLE

Objectives

- Relate Java with other languages
- Discuss the different IDEs that support the Java language
- Describe how to download and install an IDE
- Describe each phase of the product life cycle

Topics

- Describing key features of Java and object-oriented programming
- Describing the Java technology and development environment
- Working with IDEs
- Describing the product life cycle

Java's Place in the World

To put things in perspective, Java is the single most widely used development language in the world today, with over 9 million developers saying they spend at least some of their time developing in Java, according to a recent Evans Data study. That's out of a world population of about 14 million developers.

Java Desktops

- 1.1 billion desktops run Java (Nielsen Online, Gartner 2010).
- 930 million JRE downloads a year (August 2009–2010): The JRE (Java Runtime) is used by end users.
- 9.5 million JDK downloads a year (August 2009–2010): The JDK (Java Development Kit) is used by Java developers

Java Mobile Phones

- 3 billion mobile phones run on Java.
- 31 times more Java phones ship every year than Android and Apple combined (Gartner 2009).
- 5 of the top 5 manufacturers ship Java.

Java TV and Card

- 100% of Blu-ray players run Java.
- 71.2 million people connect to the web on Java-powered devices (InStat 2010).
- 1.4 billion Java Cards are manufactured every year (InStat 2010).

The Story of Java

Once upon a time…

The Java programming language (formerly Oak) originated in 1991 as part of a research project to develop a programming language that would bridge the communication gap between many consumer devices, such as video cassette recorders (VCRs) and televisions. Specifically, a team of highly skilled software developers at Sun (the Green team, under the leadership of James Gosling) wanted to create a programming language that enabled consumer devices with different central processing units (CPUs) to share the same software enhancements.

This initial concept failed after several deals with consumer device companies were unsuccessful. The Green team was forced to find another market for their new programming language. Fortunately, the World Wide Web was becoming popular and the Green team recognized that the Oak language was perfect for developing web multimedia components to enhance web pages. These small applications, called applets, became the initial use of the Oak language, and programmers using the Internet adopted what became the Java programming language.

The big break for Java came in 1995, when Netscape incorporated Java into its browser.

Did You Know? The character on the slide is Duke, Java's mascot. The original Duke was created by the Green team's graphic artist, Joe Palrang.

Key Concepts of the Java Programming Language

- Object-oriented
- Distributed
- Simple
- Multi-threaded
- Secure
- Platform-independent

The terms listed in the slide represent object-oriented concepts. We will discuss these terms in-depth, and this discussion will help you build a foundation for understanding the Java technology.

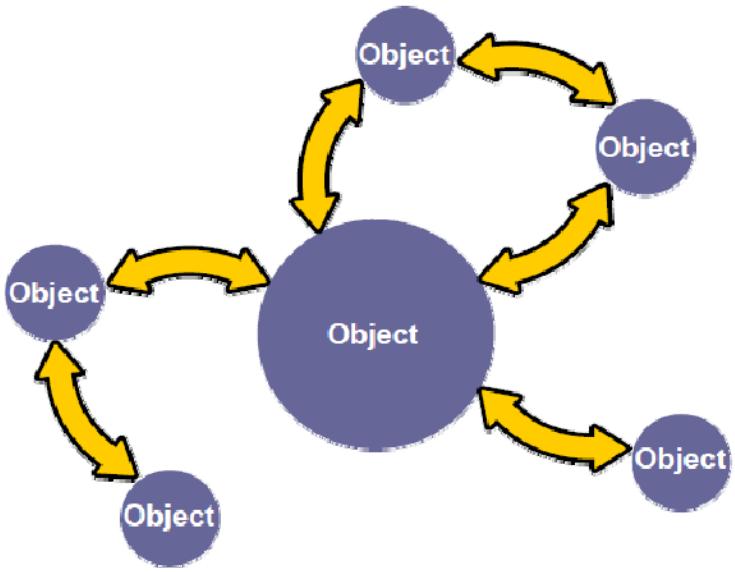
Procedural Programming

Procedural programming focuses on sequence.

The Java programming language is an object-oriented programming (OOP) language because one of the main goals of the Java technology programmer is to create objects, pieces of autonomous code, that can interact with other objects to solve a problem. OOP started with the SIMULA-67 programming language in 1967, and has led to popular programming languages such as C++, upon which the Java programming language is loosely based.

The diagram demonstrates a procedural program's focus on sequence.

Object-Oriented



OOP differs from procedural programming because procedural programming stresses the sequence of coding steps required to solve a problem, whereas OOP stresses the creation and interaction of objects.

- **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. After it is created, an object can be easily passed around inside the system.
- **Information-hiding:** By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- **Code re-use:** If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
- **Pluggability and debugging ease:** If a particular object is found to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace the bolt, not the entire machine.

The diagram illustrates an object-oriented program's focus on objects and object interactions.

Distributed

The Java programming language is a distributed language because the language provides support for distributed network technologies, such as Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), and Universal Resource Locator (URL). Additionally, the dynamic class loading capabilities of Java technology allow pieces of code to be downloaded over the Internet and executed on a personal computer.

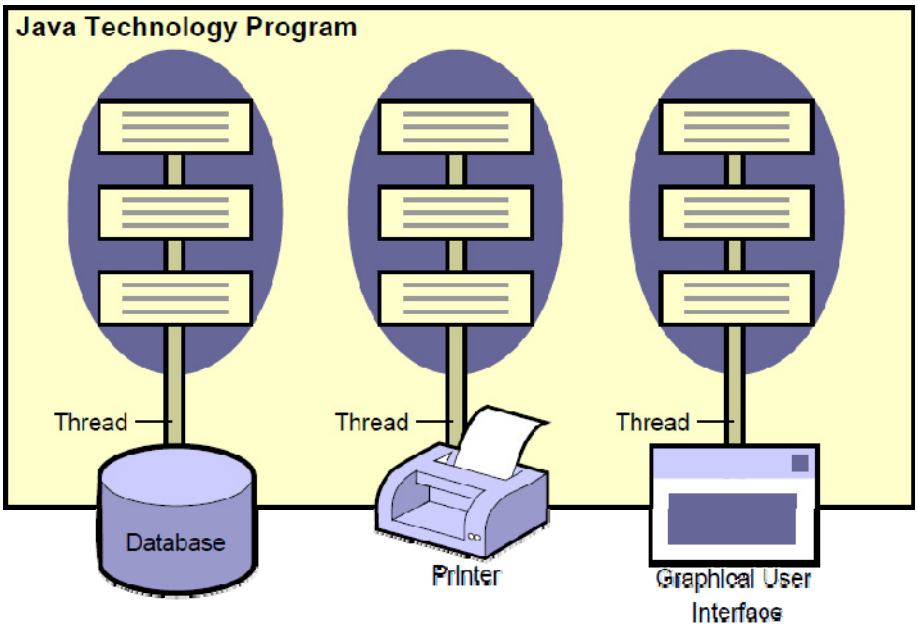
Note: The terms *Java technology* and *Java programming language* do not refer to the same thing. *Java technology* refers to a family of Java technology products of which the programming language is only one part.

Simple

- References are used instead of memory pointers.
- A `boolean` data type can have a value of either `true` or `false`.
- Memory management is automatic.

The Java programming language is simple because the designers removed some of the complex or obscure programming constructs found in other popular programming languages. As an example, the Java programming language does not allow programmers to directly manipulate pointers to memory locations (a complex feature of the C and C++ programming languages). Instead, the Java programming language only allows programmers to manipulate objects using object references. The programming language also uses a feature called a garbage collector to monitor and to remove objects that are no longer referred to. Another feature that makes the Java programming language simple is that a Java Boolean can only have a true or false value, unlike some other languages where Boolean is represented by 0 and 1.

Multi-Threaded



The Java programming language supports multithreading. This allows several tasks to run concurrently (at the same time), such as querying a database, performing long-running and complex calculations, and displaying a user interface. Multithreading allows a Java technology program to be very efficient in its use of system resources. The image illustrates how the Java programming language is multi-threaded.

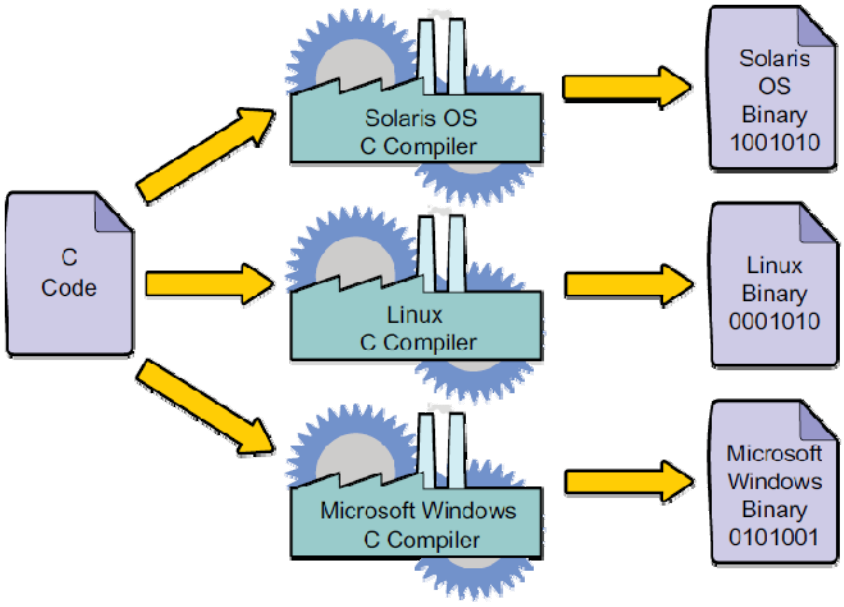
Secure

Java technology programs are secure because the Java programming language, with the environment in which Java technology programs run, uses security measures to protect programs from attacks. These measures include:

- Prohibiting distributed programs, such as applets, from reading and writing to a hard disk of a computer
- Verifying that all Java technology programs contain valid code
- Supporting digital signatures. Java technology code can be “signed” by a company or person in a way that another person receiving the code can verify the legitimacy of the code.
- Prohibiting the manipulation of memory by using pointers

The image illustrates how Java technology programs are secured by not allowing invalid code to execute on a computer.

Platform-Dependent Programs

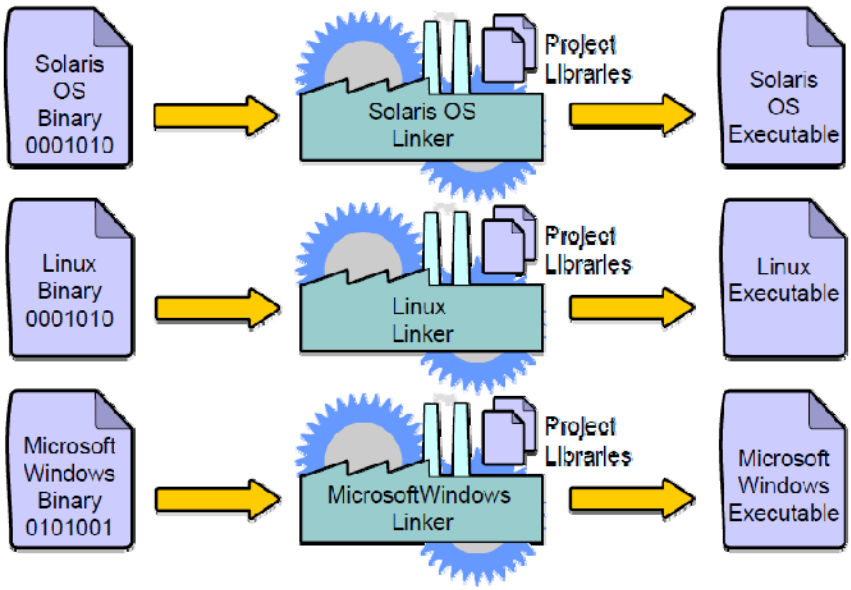


ORACLE

Programs written in most languages usually require numerous modifications to run on more than one type of computing platform, a combination of a CPU and operating system. This platform dependence is because most languages require you to write code specific to the underlying platform. Popular programming languages like C and C++ require programmers to compile and link their programs, resulting in an executable program unique to a platform. A compiler is an application that converts a program that you write into a CPU-specific code called *machine code*. These platform-specific files (binary files) are often combined with other files, such as libraries of prewritten code, using a linker to create a platform-dependent program, called an *executable*, that can be executed by an end user. Unlike C and C++, the Java programming language is platform-independent.

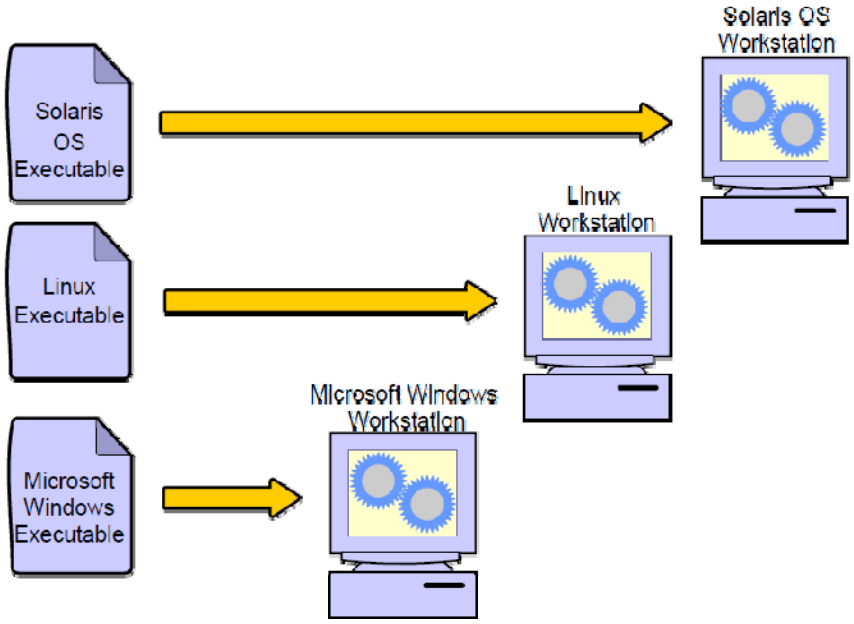
The image illustrates how a compiler creates a binary file.

Platform-Dependent Programs



The image illustrates how a binary file is linked with libraries to create a platform-dependent executable.

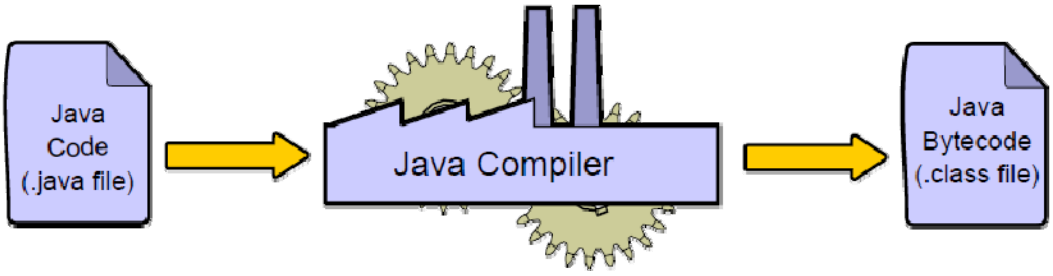
Platform-Dependent Programs



ORACLE

The image illustrates how platform-dependent executables can execute only on one platform.

Platform-Independent Programs

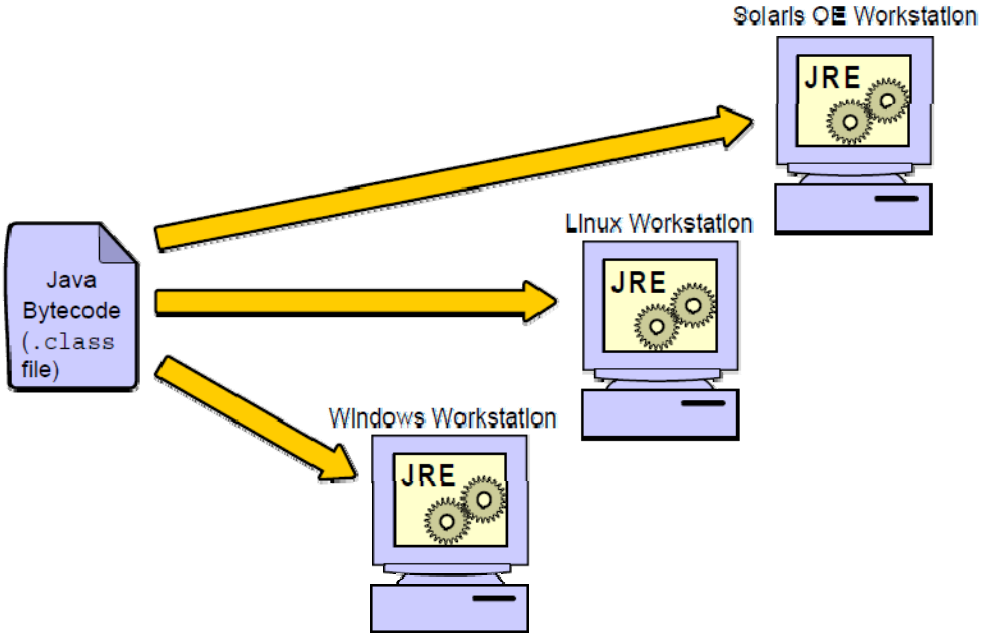


The image illustrates the Java technology compiler (Java compiler) creating Java bytecode.

A Java technology program can run on several different CPUs and operating system combinations, such as the Solaris OS on a SPARC chip, MacOS X on an Intel chip, and Microsoft Windows on an Intel chip, usually with few or no modifications.

Java technology programs are compiled using a Java technology compiler. The resulting format of a compiled Java technology program is platform-independent Java technology bytecode instead of CPU-specific machine code. After the bytecode is created, it is interpreted (executed) by a bytecode interpreter called the virtual machine or VM. A virtual machine is a platform-specific program that understands platform-independent bytecode and can execute it on a particular platform. For this reason, the Java programming language is often referred to as an interpreted language, and Java technology programs are said to be portable or executable on any platform. Other interpreted languages include Perl.

Platform-Independent Programs



The image illustrates a Java technology bytecode (Java bytecode) file executing on several platforms where a Java runtime environment exists.

A virtual machine gets its name because it is a piece of software that runs code, a task usually accomplished by the CPU or hardware machine. For Java technology programs to be platform-independent, a virtual machine called the Java Virtual Machine (JVM) is required on every platform where your programming will run. The Java Virtual Machine is responsible for interpreting Java technology code, loading Java classes, and executing Java technology programs.

However, a Java technology program needs more than just a Java Virtual Machine to execute. A Java technology program also needs a set of standard Java class libraries for the platform. Java class libraries are libraries of prewritten code that can be combined with the code that you write to create robust applications.

Combined, the JVM software and Java class libraries are referred to as the Java runtime environment (JRE). Java runtime environments are available from Oracle for many common platforms.

Note: Some modifications might be required to make a Java technology program platform-independent. For example, directory names might need to be altered so that they use the appropriate delimiters (forward and backward slashes) for the underlying operating system.

Quiz

The Java programming language is said be platform-independent because:

a. Compiled code runs on multiple platforms with few or no modifications
b. It does not allow pointers to be used to manipulate memory
c. The format of a compiled Java program is CPU-specific code
d. It is multi-threaded

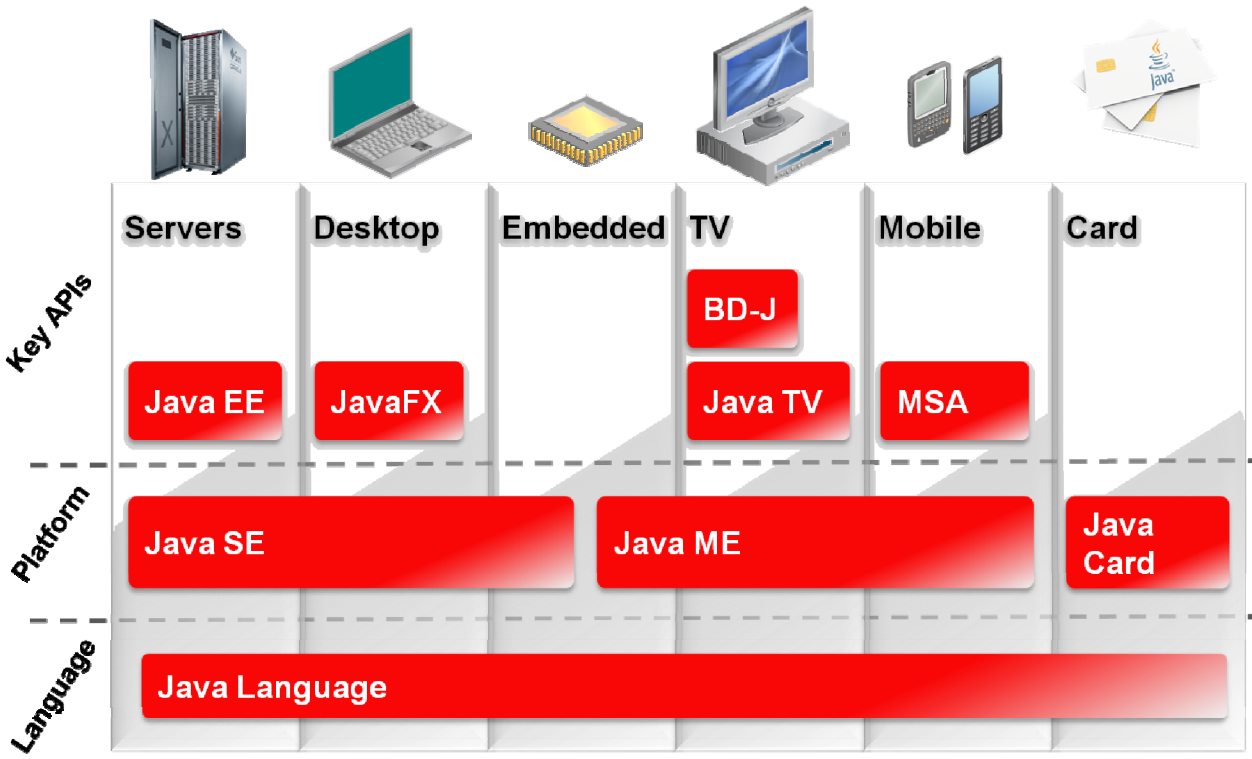
Answer: a

- b is a correct statement but an incorrect answer because it does not relate to platform independence.
- c is incorrect because a compiled Java program is not CPU-specific. It is interpreted by the virtual machine that resides on the system.
- d is a correct statement because Java is multi-threaded, but that is not why it is called platform-independent.

Topics

- Describing key features of Java and object-oriented programming
- Describing the Java technology and development environment
- Working with IDEs
- Describing the product life cycle

Identifying Java Technology Product Groups



Oracle provides a complete line of Java technology products ranging from kits that create Java technology programs to emulation (testing) environments for consumer devices, such as cellular phones. As indicated in the graphic, all Java technology products share the foundation of the Java language. Java technologies, such as the Java Virtual Machine, are included (in different forms) in three different groups of products, each designed to fulfill the needs of a particular target market. The figure illustrates the three Java technology product groups and their target device types. Each edition includes a Java Development Kit (JDK) [also known as a Software Development Kit (SDK)] that allows programmers to create, compile, and execute Java technology programs on a particular platform.

Note: The JavaFX API is a rich client for creating user interfaces for your Java program. The MSA API is the mobile software application used to create user interfaces on portable devices.

Java SE

Is used to develop applets that run within web browsers and applications that run on desktop computers

Java Platform, Standard Edition (Java SE): Used to develop applets and applications that run within web browsers and on desktop computers. For example, you can use the Java SE JDK to create a word processing program for a personal computer.

We are using two Java desktop applications in this course: NetBeans and UMLet.

Note: Applets and applications differ in several ways. Primarily, applets are launched inside a web browser, while applications are launched within an operating system. While this course focuses mainly on application development, most of the information in this course can be applied to applet development.

Java EE

Is used to create large enterprise, server-side, and client-side distributed applications

Java Platform, Enterprise Edition (Java EE): Used to create large enterprise, server-side, and client-side distributed applications. For example, you can use the Java EE JDK to create a web shopping (eCommerce) application for a retail company's website.

Java EE is built on top of the Java SE Platform, extending it with additional APIs supporting the needs of large-scale, high-performance enterprise software. The APIs are packaged and grouped to support different kinds of containers, such as a Web container for Web-based applications, a client container for thick clients, and the EJB container to run workhorse Java components. Some of the kinds of functionality supported by the different APIs include objects, UI, integration, persistence, transactions, and security.

Java ME

Is used to create applications for resource-constrained consumer devices

ORACLE

Java Platform, Micro Edition (Java ME): Used to create applications for resource-constrained consumer devices. For example, you can use the Java ME JDK to create a game that runs on a cellular phone. Blu-ray Disc Java applications and Java TV use the same SDK as Java ME.

Java Card

Java Card is typically used for:

- Identity
- Security
- Transactions
- Mobile phone SIMs
- Much more…

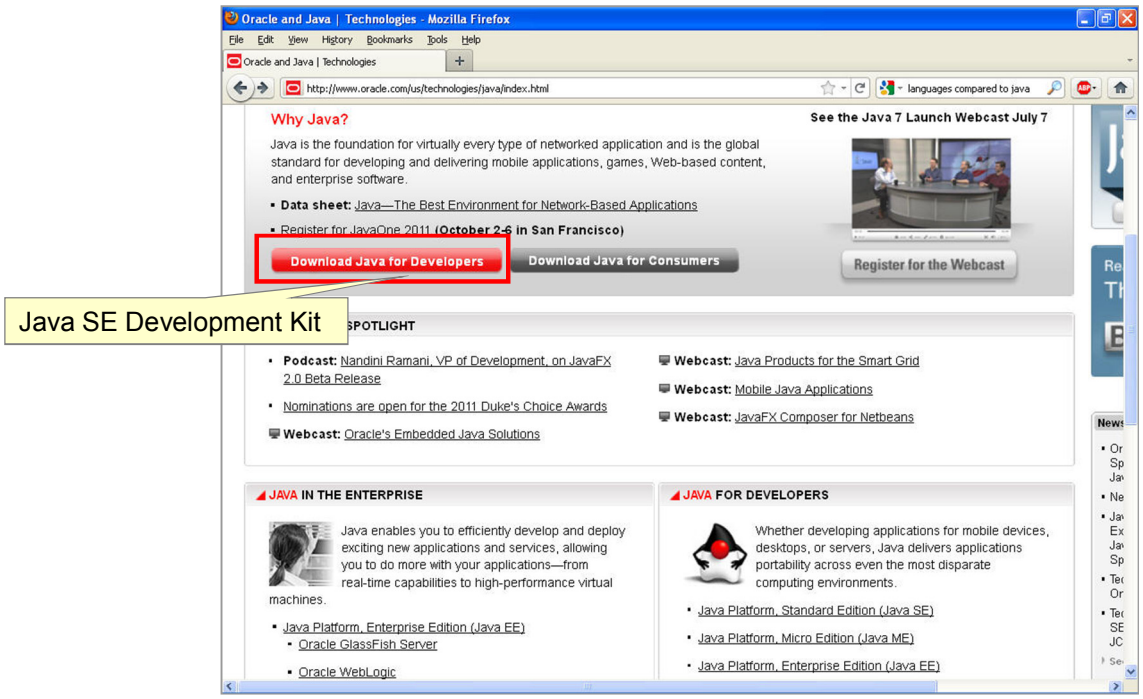
Setting Up the Java Development Environment

It is easy to set up your Java development environment.

- Download and install the Java Development Kit (JDK) from oracle.com/java.
- Set your `PATH` to the installed JDK.
- Compile and run a Java application by using the command line

Setting up your Java development environment is a simple task. The JDK is available for free from the Oracle Java website.

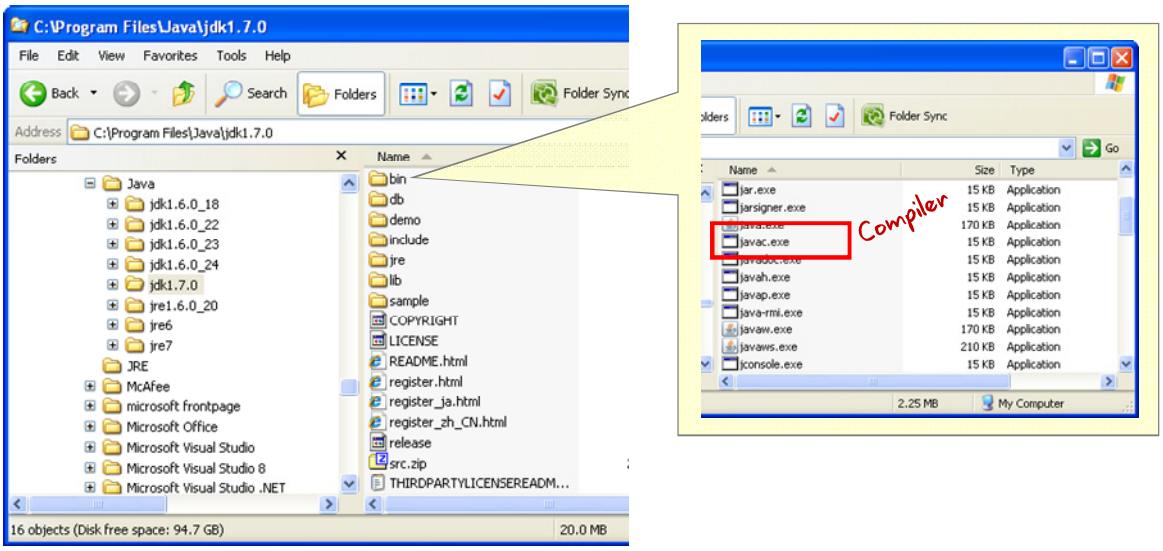
Downloading and Installing the Java JDK



1. Go to oracle.com/java.
2. Choose the Java Platform, Standard Edition (Java SE) link.
3. Download the version for your platform.
4. Follow the installation instructions.
5. Set your Java `PATH`.
6. Compile and run a sample Java application.

Note: The activities for this lesson will show you how to complete Steps 5 and 6.

Examining the Installed Java Development Kit



Java SE Development Kit

When you download and install the Java SE Development Kit, the following items are installed:

- Java runtime environment (JRE):
- A Java Virtual Machine (JVM) for the platform you choose
- Java class libraries for the platform you choose
- A Java technology compiler
- Additional utilities, such as utilities for creating Java archive files (JAR files) and for debugging Java technology programs
- Examples of Java technology programs

Java class library (API) documentation is a separate download.

Note: The compiler (`javac`) is located in the `../jdk<version>/bin` directory.

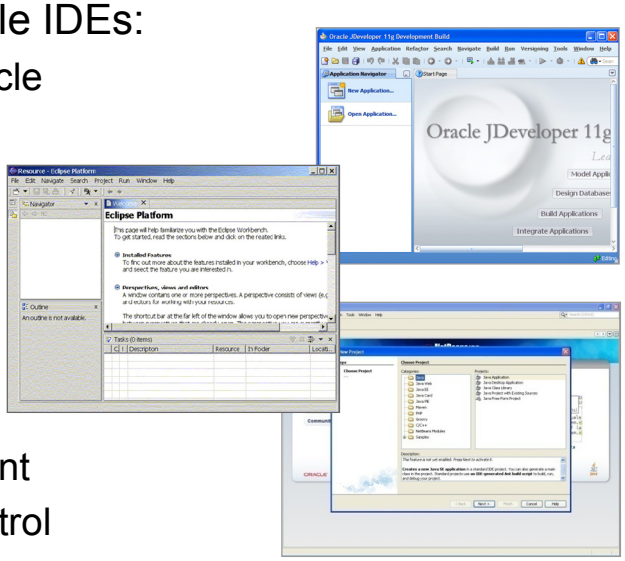
Topics

- Describing key features of Java and object-oriented programming
- Describing the Java technology and development environment
- Working with IDEs
- Describing the product life cycle

Using an Integrated Development Environment

An integrated development environment (IDE) is a tool that can assist you with your Java application development.

- There are several available IDEs:
  - NetBeans IDE from Oracle
  - JDeveloper from Oracle
  - Eclipse from IBM
- Features include:
  - Full integration
  - Easy deployment
  - Smart editor
  - Easy project development
  - Built-in source code control

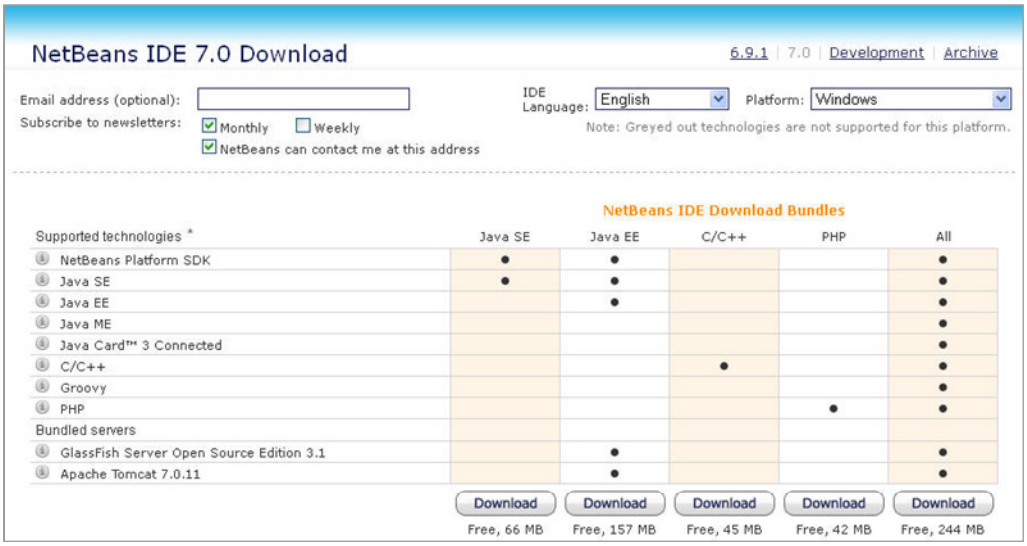


Integrated development environments help speed development time by:

- Automating simple tasks
- Using code auto-completion
- Integrating debugging
- Facilitating the compilation and deployment of applications

NetBeans IDE Download

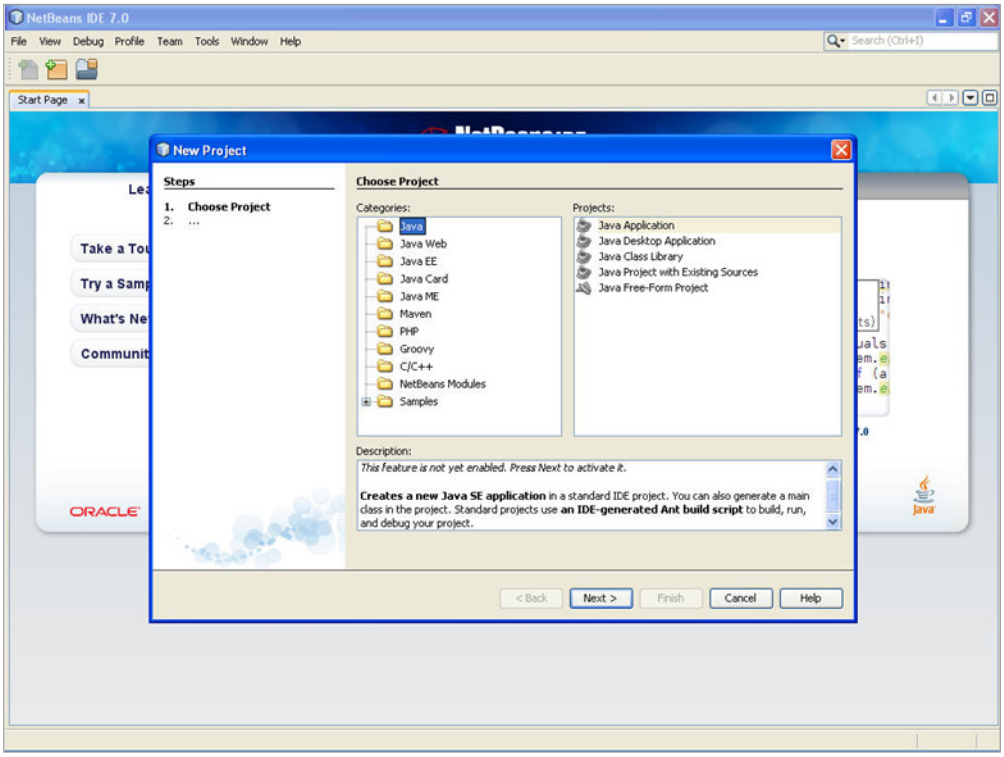
- Available free from NetBeans.org
- Sets the application's Java properties automatically
- Several available bundles



We will use the NetBeans IDE for the course activities. When you are familiar with an IDE, you can easily transfer your skills to any similar IDE of your choice.

The NetBeans IDE 7.0 is compatible with and supports Java SE 7. The IDE's installer requires a JDK to install the IDE on your system, because NetBeans is a Java application. However, you can add additional JDK versions after NetBeans is in place, and you can choose which JDK version to use when you create a NetBeans project. The IDE is available with specific downloads that support various Java technologies, as shown in the graphic. For example, you might choose to download and install the Java SE download only, or you can choose the NetBeans **All** bundle.

NetBeans IDE and New Project Wizard



In NetBeans, you work within the context of a project, which consists of an organized group of source files and associated metadata; project-specific properties files; an Ant build script and run settings; and all the tools you'll need to write, compile, test, and debug your application. You can create a main project with subprojects, and you can link projects through dependencies. So getting started is as easy as giving your project a name. After you tell NetBeans the name of a new project, it then:

- Creates a source tree with an optional skeleton class inside
- Creates a folder for unit tests
- Sets classpaths for compiling, running, and testing
- Sets the Java platform on which the project runs
- Creates an Ant build script (`build.xml`), which contains instructions that the IDE uses when you perform commands on your project, such as compile or run

You will get to explore these features during the activities for this lesson.

Quiz

The Java technology product group that is designed for developing applications for consumer devices is _____.

Choose one from the list to fill in the blank:

a. Java SE JDK
b. Java ES SDK
c. Java EE SDK
d. Java ME SDK

Answer: d

Topics

- Describing key features of Java and object-oriented programming
- Describing the Java technology and development environment
- Working with IDEs
- Describing the product life cycle

Product Life Cycle (PLC) Stages

1. Analysis
2. Design
3. Development
4. Testing
5. Implementation
6. Maintenance
7. End-of-life (EOL)

The product life cycle is an iterative process used to develop new products by solving problems.

- **Analysis:** The process of investigating a problem that you want to solve with your product. Among other tasks, analysis consists of:
  - Clearly defining the problem you want to solve, the market niche you want to fill, or the system you want to create. The boundary of a problem is also known as the *scope* of the project.
  - Identifying the key subcomponents that compose your overall product

  Note: Good analysis of the problem leads to a good design of the solution and to decreased development and testing time.
- **Design:** The process of applying the findings you made during the analysis stage to the actual design of your product. The primary task during the design stage is to develop blueprints or specifications for the products or components in your system.
- **Development:** Consists of using the blueprints created during the design stage to create actual components
- **Testing:** Consists of ensuring that the individual components, or the product as a whole, meet the requirements of the specification created during the design stage

  Note: Testing is usually performed by a team consisting of people other than those who actually developed the product. A team ensures that the product is tested without any bias on behalf of the developer.

- **Implementation:** Consists of making the product available to customers
- **Maintenance:** Consists of fixing problems with the product and rereleasing the product as a new version or revision
- **End-of-life (EOL):** Although the PLC does not have a separate stage for the start of a concept or project, it does have a stage for the end of a project. EOL consists of carrying out all of the necessary tasks to ensure that the customers and employees are aware that a product is no longer being sold and supported, and that a new product is available.

The PLC is an important part of product development because it helps to ensure that products are created and delivered so that time-to-market is reduced, the quality of a product is high, and the return on investment is maximized. Developers who do not follow the PLC often encounter problems with their products that are costly to fix and that could have been avoided.

Summary

In this lesson, you should have learned how to:

- Describe key features of the Java technology
- Describe and identify features of object-oriented programming
- Discuss the difference between compiled and interpreted languages
- Describe how to download and install the Java Platform
- Describe how to run a Java application by using the command line
- Identify the different Java technologies

ORACLE

Summary

- Relate Java with other languages
- Discuss the different IDEs that support the Java language
- Describe how to download and install an IDE
- Describe each phase of the product life cycle

Practice 2-1: Running a Java Program by Using the Command Line

This practice covers compiling and executing a Java program.

Practice 2-2: Running a Java Program by Using the NetBeans IDE

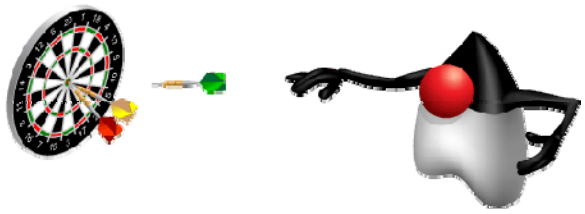
This practice covers compiling and executing a Java program by using the NetBeans IDE.

Thinking in Objects

Objectives

After completing this lesson, you should be able to:

- Analyze a problem by using object-oriented analysis (OOA)
- Identify a problem domain
- Identify the objects
- Define additional criteria for recognizing objects
- Define attributes and operations
- Discuss a case study solution
- Design a class
- Model a class



ORACLE

Relevance

- How do you decide what components are needed for something that you are going to build, such as a house or a piece of furniture?
- What is a taxonomy?
- How are organisms in a taxonomy related?
- What is the difference between attributes and values?

Generally, you determine the scope of the item first (the outer dimensions of the item [height, width, depth], the way the item must "fit" into its environment [lot size], and so on). After that, you can begin to divide the item into its major components, which are usually recognized as nouns or "things," such as floor, roof, or kitchen.

A taxonomy is a classification of related organisms that have similar characteristics (or features) called attributes, such as:

- Fins or gills
- Operations
- The ability to swim
- The ability to walk on two feet

Attributes are distinguishing characteristics or features of an organism from a similar taxonomy (for example, a dorsal fin is an attribute for a whale).

Values represent the current state of an attribute. For example, one whale, the blue whale, has a small dorsal fin while another whale, the orca or killer whale, has a large dorsal fin. Large and small are values for the fin attribute in the whale taxonomy.

Topics

- Analyze a problem by using object-oriented analysis (OOA)
- Identify a problem domain
- Identify the objects
- Define additional criteria for recognizing objects
- Define attributes and operations
- Discuss a case study solution
- Design and model a class

Analyzing a Problem by Using Object-Oriented Analysis (OOA)

Duke's Choice sells clothing from their catalog. Business is growing 30 percent per year, and they need a new order entry system.

ORACLE

Duke's Choice produces an online catalog of clothing every three months and emails it to subscribers. Each shirt in the catalog has an item identifier (ID), one or more colors (each with a color code), one or more sizes, a description, and a price.

Duke's Choice accepts all credit cards. Customers can call Duke's Choice to order directly from a customer service representative (CSR), or customers can complete an online order form on the Duke's Choice website.

Duke's Choice Order Process

As the customer enters the order online, each item's availability (quantity-on-hand) is verified. If one or more items are not currently available (in Duke's Choice's warehouse), then the item is marked as backordered until the items arrive at the warehouse. After all of the items are available, payment is verified and the order is submitted to the warehouse for assembly and for shipping to the customer's address. When the order is received, the customer is given an order ID, which is used to track the order throughout the process. Orders that are phoned in are entered by a CSR.

Note: In a true analysis, you would work side-by-side with a company getting details about each aspect of how the company does its business. This case study outlines only a small portion of the information needed to create a system for Duke's Choice.

Topics

- Analyze a problem by using object-oriented analysis (OOA)
- Identify a problem domain
- Identify the objects
- Define additional criteria for recognizing objects
- Define attributes and operations
- Discuss a case study solution
- Design and model a class

Identifying a Problem Domain

- A problem domain is the scope of the problem that you will solve.
- Example: “Create a system allowing the online order entry method to accept and verify payment for an order.”

Because Java is an object-oriented programming language, one of the main goals of the Java technology programmer is to create objects to build a system or, more specifically, to solve a problem.

The scope of the problem you will solve is referred to as a problem domain. Most projects start by defining the problem domain, by gathering customer requirements, and by writing a statement of scope that briefly states what you, the developer, want to achieve. For example, a scope statement for the Duke’s Choice project might be: “Create a system allowing the online order entry method to accept and verify payment for an order.” After you have determined the scope of the project, you can begin to identify the objects that will interact to solve the problem.

Topics

- Analyze a problem by using object-oriented analysis (OOA)
- Identify a problem domain
- Identify the objects
- Define additional criteria for recognizing objects
- Define attributes and operations
- Discuss a case study solution
- Design and model a class

Identifying Objects

- Objects can be physical or conceptual.
- Objects have *attributes* (characteristics) such as size, name, shape, and so on.
- Objects have *operations* (the things they can do) such as setting a value, displaying a screen, or increasing speed.

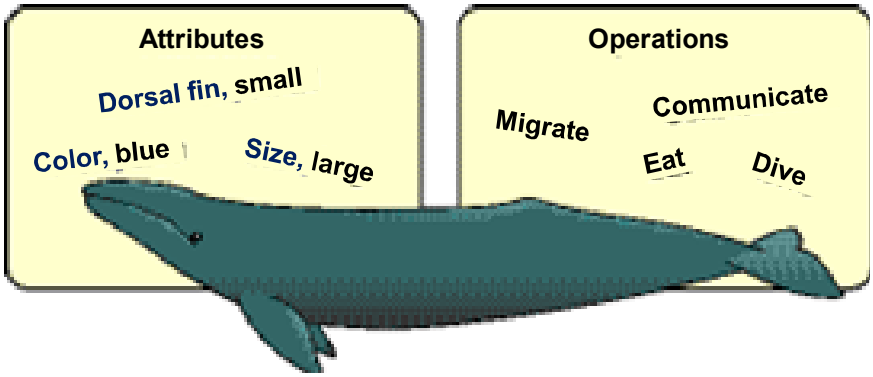
To validate objects in a problem domain, you must first identify the properties of all objects:

- Objects can be physical or conceptual. A customer's account is an example of a conceptual object, because it is not something you can physically touch. An automated teller machine (ATM) is something many people touch every day and is an example of a physical object.
- Objects have attributes (characteristics) such as size, name, shape, and so on, that represent the state of the object. For example, an object might have a color attribute. The value of all of an object's attributes is often referred to as the object's current state. An object might have a color attribute with the value of red and a size attribute with a value of large.

- Objects have operations (the things they can do) such as setting a value, displaying a screen, or increasing speed, which represent the behavior through which the state of the object can be altered. Operations usually affect an object's attributes. The operations that an object performs are often referred to as its behavior. For example, an object might have an operation allowing other objects to change the object's color attribute from one state to another, such as from red to blue.

Did You Know? Object names are often nouns, such as "account" or "shirt." Object attributes are often nouns too, such as "color" or "size." Object operations are usually verbs or noun-verb combinations, such as "display" or "submit order." Your ability to recognize objects in the world around you will help you to define objects better when you approach a problem using object-oriented analysis.

Identifying Objects



The figure illustrates the characteristics of a whale that make it an object.

Discussion: Look around the room. What objects are in the room that you are sitting in at this moment? For example, a door can be an object within the problem domain of “build a house.” A door has at least one attribute which has a value (open or closed) and an operation such as “close door” or “open door” allowing you to change the state of a door.

Did You Know? An attribute with only two states is referred to as a Boolean attribute.

A clock can be an object. A clock has at least one attribute (current time) that has a value (the current Hours:Minutes:Seconds) and a dial that allows you to set the value of the current time (an operation).

A chair can be an object. A chair has at least one attribute (height) that has a value (height in inches), and it can have a lever allowing another object, such as a person, to change the value of the height (an operation). An instructor can be an object. A student can be an object.

Topics

- Analyze a problem by using object-oriented analysis (OOA)
- Identify a problem domain
- Identify the objects
- Define additional criteria for recognizing objects
- Define attributes and operations
- Discuss a case study solution
- Design and model a class

Additional Criteria for Recognizing Objects

- Relevance to the problem domain:
  - Does the object exist within the boundaries of the problem domain?
  - Is the object required for the solution to be complete?
  - Is the object required as part of an interaction between a user and the system?
- Independent existence

Use the following criteria to further test whether something should be considered an object in a problem domain:

- Relevance to the problem domain
- Independent existence

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

- Does the object exist within the boundaries of the problem domain?
- Is the object required for the solution to be complete?
- Is the object required as part of an interaction between a user and the solution?

Note: Some items in a problem domain can be attributes of objects or can be objects themselves. For example, temperature can be an attribute of an object in a medical system, or it can be an object in a scientific system that tracks weather patterns.

For an item to be an object and not an attribute of another object, it must exist independently in the context of the problem domain. Objects can be connected and still have independent existence. In the Duke's Choice case study, a customer and an order are connected but are independent of each other, so both would be objects.

When evaluating potential objects, ask yourself if the object needs to exist independently, rather than being an attribute of another object. Identifying objects in a problem domain is an art, not a science. Any object could be a valid object if it has relevance to the domain of a problem and has the characteristics of an object, but this does not mean that it is a good object. Regardless, the person who models the system or solution must understand the entire system.

Possible Objects in the Duke's Choice Case Study

The figure illustrates three objects in the problem domain for the Duke's Choice order entry system. This list is not an exhaustive, authoritative answer. This list is just a first analysis of the system.

Some nouns that are probably not appropriate objects for this system are:

- Fax
- Verification
- Payment

Topics

- Analyze a problem by using object-oriented analysis (OOA)
- Identify a problem domain
- Identify the objects
- Define additional criteria for recognizing objects
- Define attributes and operations
- Discuss a case study solution
- Design and model a class

Identifying Object Attributes and Operations

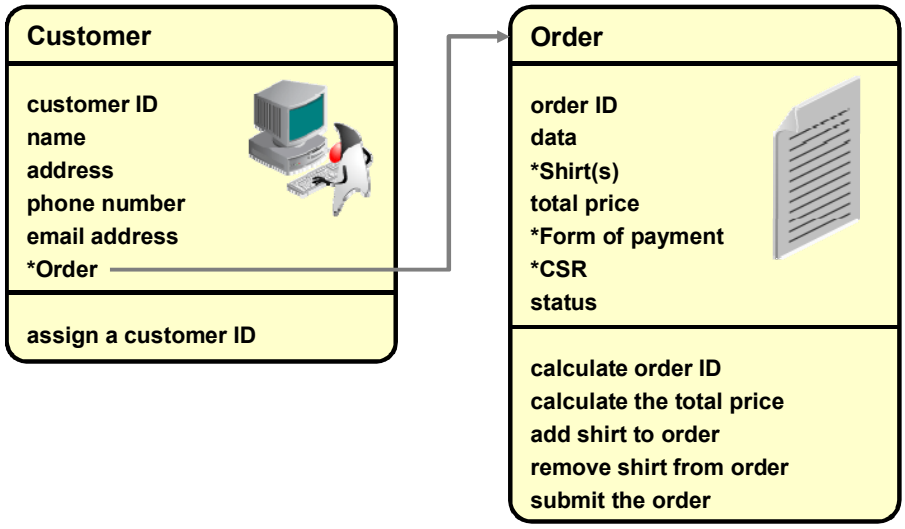
- Attributes are data, such as:
  - ID
  - Order object
- Operations are actions, such as:
  - Delete item
  - Change ID

After you identify the objects, you then specify their attributes and operations.

As described previously, attributes define the state of an object. Attributes can be data, such as order ID and customer ID for an Order object, or they can be another object, such as the customer having an entire Order object as an attribute rather than just the order ID.

As described previously, operations are behaviors that usually modify the state of an attribute. For example, an order can be printed, can have an item added or deleted, and so on. (The customer or CSR would be initiating those actions in real life, but the operations belong to the Order object.)

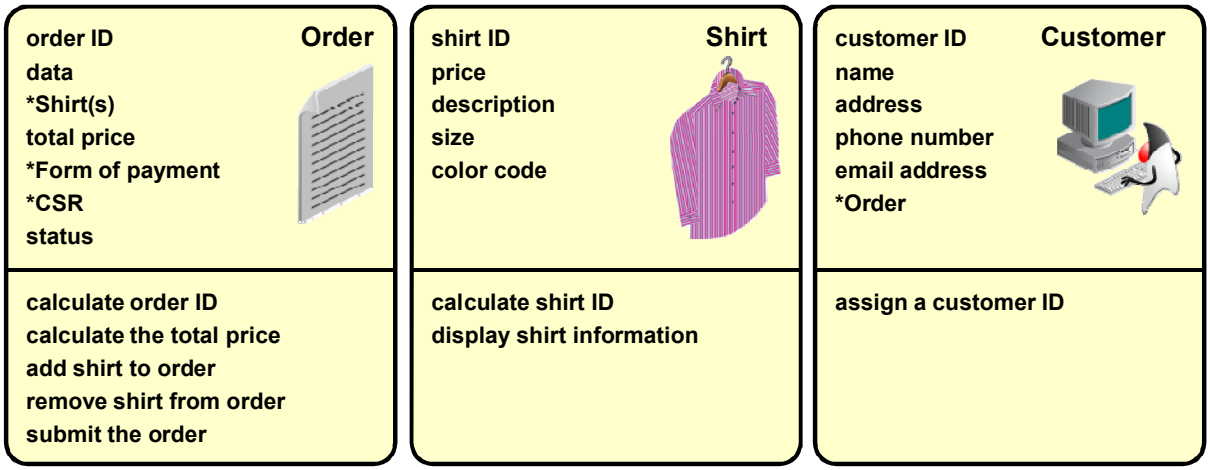
Object with Another Object as an Attribute



An attribute can be a reference to another object. For example, the Customer object can have an attribute that is an Order object. This association might or might not be necessary, depending on the problem you are trying to solve.

Note: Use attribute and operation names that clearly describe the attribute or operation. The figure illustrates the Customer object containing an Order attribute. The asterisks (*) denote attributes that are other objects.

Possible Attributes and Operations for Objects in the Duke's Choice Case Study



The figure illustrates some possible attributes and operations for Order, Shirt, and Customer objects.

Topics

- Analyze a problem by using object-oriented analysis (OOA)
- Identify a problem domain
- Identify the objects
- Define additional criteria for recognizing objects
- Define attributes and operations
- Discuss a case study solution
- Design and model a class

Case Study Solution: *Classes*

Class	Order	Shirt	Customer	Form of Payment	Catalog	CSR

Case Study Solution: *Attributes*

Class	`Order`	`Shirt`	`Customer`
Attributes	order ID date *Shirt(s) total price *Form of payment *CSR status	shirt ID price description size color code	customer ID name address phone number email address *Order

Case Study Solution: *Attributes*

Class	`Form of Payment`	`Catalog`	`CSR`
Attributes	customer ID name address phone number email address *Order	*Shirt(s)	name extension

Case Study Solution: *Behaviors*

Class	`Order`	`Shirt`	`Customer`
Attributes	order ID date *Shirt(s) total price *Form of payment *CSR status	shirt ID price description size color code	customer ID name address phone number email address *Order
Behaviors	calculate order ID calculate the total price add shirt to order remove shirt from order submit the order	calculate shirt ID display shirt information	assign a customer ID

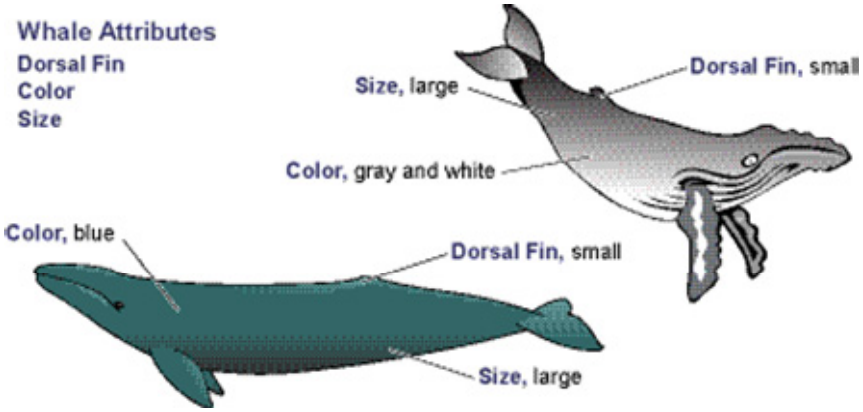
Case Study Solution: *Behaviors*

Class	`Form of Payment`	`Catalog`	`CSR`
Attributes	customer ID name address phone number email address *Order	*Shirt(s)	name extension
Behaviors	verify credit card number verify check payment	add a shirt remove a shirt	process order

Topics

- Analyze a problem by using object-oriented analysis (OOA)
- Identify a problem domain
- Identify the objects
- Define additional criteria for recognizing objects
- Define attributes and operations
- Discuss a case study solution
- Design and model a class

Designing Classes



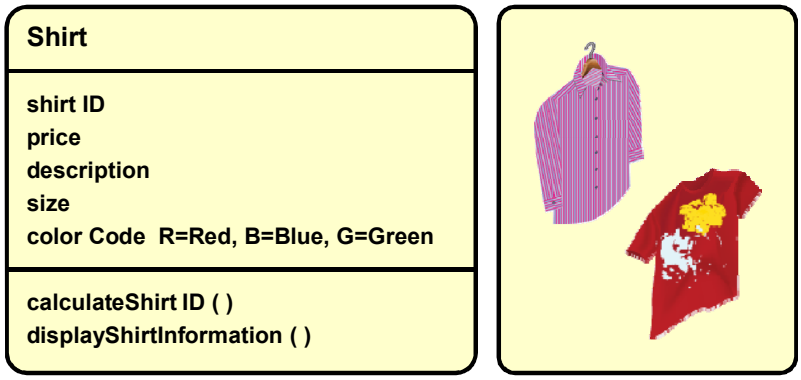
Identifying objects helps you design the class or blueprint for each of the objects in a system. For example, window manufacturers often create a single blueprint for each of the styles of windows they create. These blueprints define a range of colors and styles that can be selected when the window is purchased.

These blueprints are then the basis for any number of windows with any number of combinations of color and style. In object-oriented design terms, each object (window) created using the class (generic blueprint) is called an instance of a class. More specifically, each object created from a class can have a specific state (values) for each of its attributes, but will have the same attributes and operations.

Note: The *American Heritage Dictionary* defines the word *class* to be “a group whose members have certain attributes in common.”

Classes and objects are often used in the field of biology. For example, a marine biologist studying sea creatures is often asked to categorize sea creatures in a family, or class, of sea creatures. In OOA terms, each animal (such as a blue whale) in a family (such as whales) can be considered an object instance of the whale class.

Class and Resulting Objects



Shirt Class **Shirt Objects**

Referring to the Duke's Choice case study:

- A class is the way you define an object. Classes are descriptive categories, templates, or blueprints. A Shirt could be a class defining all shirts to have a shirt ID, size, color code, description, and a price.
- Objects are unique instances of classes. The large blue polo shirt that costs $29.99 with shirt ID 62467-B is an instance of the Shirt class, as is the small green shirt with the same price and shirt ID 66889-C, or the patterned shirt for $39.99, ID 09988-A. You can also have two Shirt objects in memory with exactly the same attribute values.

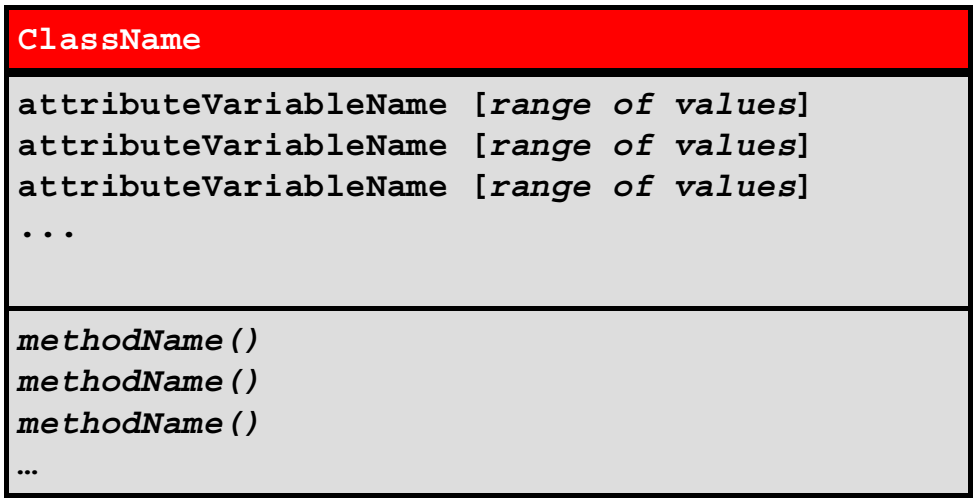
The graphic illustrates a class and several objects based on the class.

Note: You will revisit the Shirt class throughout this course.

In the Java programming language, attributes are represented using variables, and operations are represented using methods. Variables are the Java programming language mechanism for holding data. Methods are the Java programming language mechanism for performing an operation.

Modeling Classes

Syntax:



The first phase of the design stage consists of visually organizing or modeling a program and its classes. Each class in a design should be modeled so that it is enclosed in a box with the class name at the top, followed by a list of the attribute variables (including the range of possible values) and a list of methods.

The syntax for modeling a class is shown in the figure. The syntax for modeling a class is:

- The `ClassName` is the name of the class.
- The `attributeVariableName` is the name of the variable for an attribute.
- The `range of values` is an optional range of values that the attribute can contain.
- The `methodName` is the name of a method.

Modeling Classes

Example:

For example, the figure contains a modeled Shirt object.

Note: This modeling technique is loosely based on a light version of the Unified Modeling Language (UML), which is a tool to aid in the modeling process (some of the details have been removed for newer programmers).

Variable and method names are written in a special short-hand called "camelCase." Camel case specifies that a variable or method, representing any multiple-word attribute or operation, starts with a lowercase letter, and subsequent words are capitalized. For example, an operation such as “calculate the total price” is written calcTotalPrice(). Furthermore, a set of closed parentheses indicates a method.

Note: Modeling classes are similar to modeling database structures. In fact, your object data can be stored in a database using the Java Database Connectivity (JDBC) API. The JDBC API allows you to read and write records by using structured query language (SQL) statements within your Java technology programs.

Using UML-like Modeling

UML: Universal Modeling Language

- UML is used to:
  - Model the objects, attributes, operations, and relationships in object-oriented programs
  - Model the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects
- There are many courses available that teach UML.

UML is used to model object-oriented programs. There are many courses available that teach UML. We will not be teaching UML in our Java courses, but we can touch on a few features of UML that you can use to solve your case study.

- Choose nouns for all your objects.
- Choose verbs for all your methods.
- Choose adjectives for all your attributes.

You can use a simple text editor to complete Practice 3. We want you to get into the habit of finding the objects that make up your classes. UML is a good method for identifying the classes, objects, and methods that comprise your case study.

Quiz

Which of the following terms represents two different properties of an object?

a. Methods and operations
b. The problem domain
c. Attributes and operations
d. Variables and data

Answer: c

Quiz

Which of the following statements is true?

a. An object is a blueprint for a class.
b. An object and a class are exactly the same.
c. An object is an instance of a class.
d. An attribute cannot be a reference to another object.

Answer: c

- a is false because a class is a blueprint for an object.
- b is false because an object is simply an instantiation of a class, and a class serves as a blueprint for the object.
- c is correct.
- d is false because an attribute can be a reference to another object.

Summary

In this lesson, you should have learned how to:

- Analyze a problem by using object-oriented analysis (OOA)
- Identify a problem domain
- Identify the objects
- Define additional criteria for recognizing objects
- Define attributes and operations
- Discuss a case study solution
- Design a class
- Model a class

ORACLE

Practice 3.1 Overview: Analyzing a Problem by Using Object-Oriented Analysis

This practice covers performing an analysis of the case study.

Practice 3.2 Overview: Designing a Solution

This practice covers producing a design by using UML-like notation.

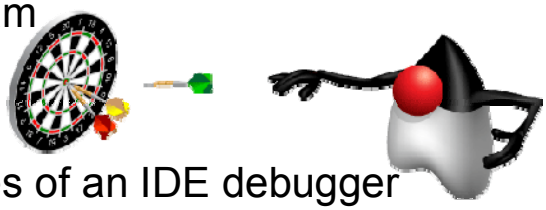
4

Introducing the Java Language

Objectives

After completing this lesson, you should be able to:

- Define a class
- Identify the components of a class
- Explain the term *object*
- Describe the purpose of a variable
- Discuss methods and describe how to use a `main` method
- Describe the elements that comprise a Java class, such as declarations, return values, and the proper use of brackets and braces
- Identify keywords and describe their purpose
- Test and execute a simple program
- Describe some common causes of syntax errors
- Describe the purpose and features of an IDE debugger



ORACLE

Topics

- Define a class, identify class components, and use variables
- Discuss methods and use of a `main` method
- Identify keywords
- Test and execute a simple Java program
- Describe some common causes of syntax errors
- Describe the purpose and features of an IDE debugger

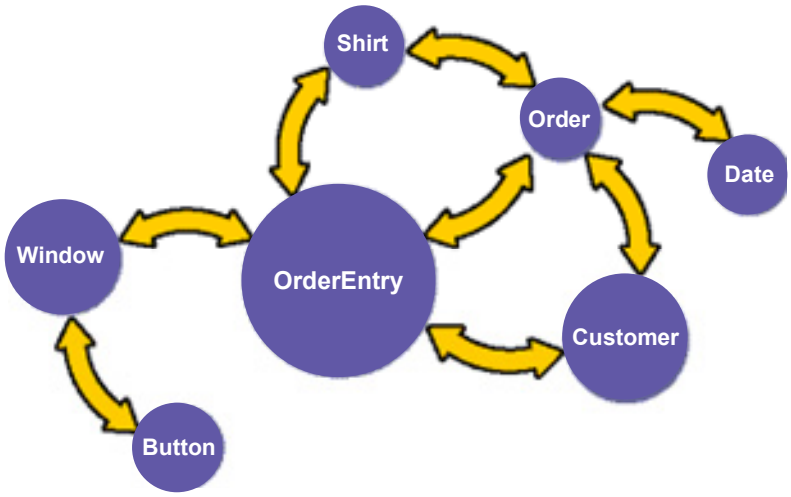
Relevance

How do you test something that you have built, such as a house, a piece of furniture, or a program?

This lesson provides an overview of the components of a class. It also describes how to compile and execute a Java technology program that consists of multiple classes. We need to understand what developing and testing classes is all about.

There are several ways to test a program. You can test different components (unit test), you can test the entire item by seeing if it “fits” into its environment, and so on.

Identifying the Components of a Class



Classes are the blueprints that you create to define the objects in a program. For example, the figure illustrates some of the objects that might exist in the order entry program for Duke's Choice.

A desktop application usually consists of one object, often called the controller object, main object, or test object that is the starting point for your program. In the previous figure, an OrderEntry object might interact with one or more Window objects, Customer objects, Order objects, and so on while your program runs. Every object in this illustration is an instance of a blueprint or class. For example, all Window objects are instances of `Window` classes. Some classes, such as the `Window` class (used for creating graphical user interface [GUI] windows), are general-purpose classes and are provided to you as part of the Java technology API. Other classes, such as the `Shirt` class, might be unique to your particular program, so you must create them yourself. This course describes how to use existing classes and how to create and use your own classes.

Structuring Classes

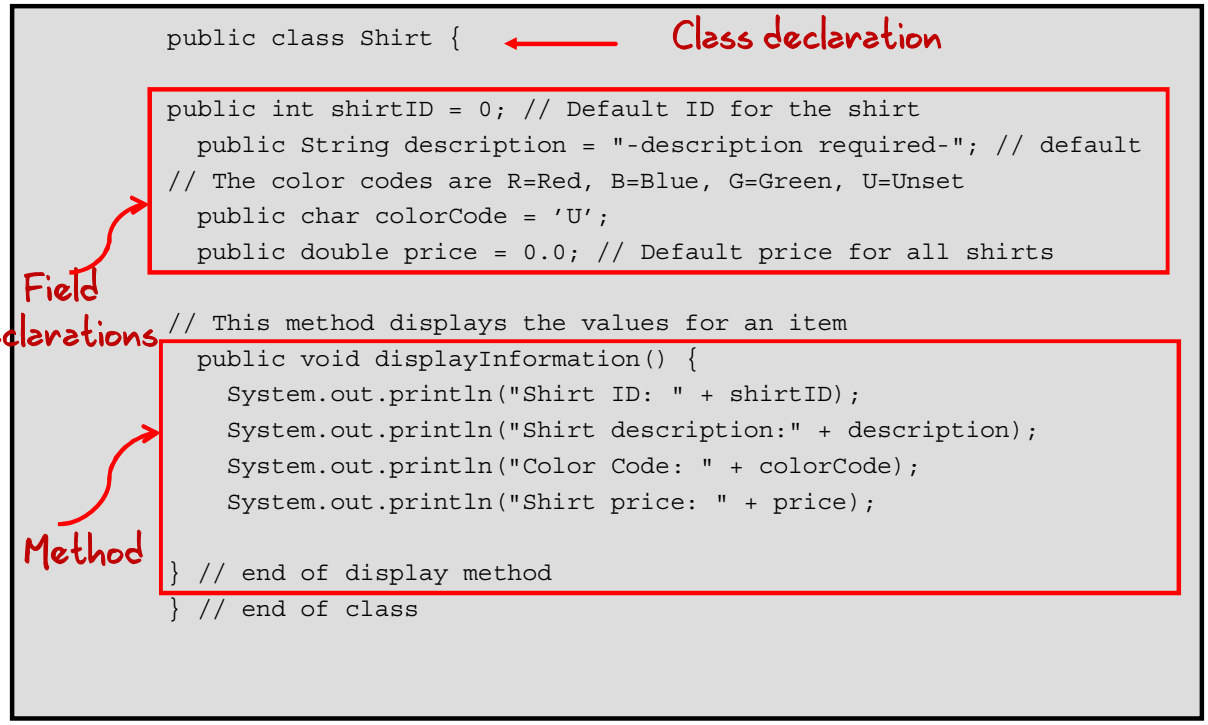
- The class declaration
- Field declarations (class attributes are called “fields”)
  - Fields may also be initialized at declaration time.
- Methods (optional)
- Comments (optional)

Classes are composed of the Java technology code necessary to instantiate objects, such as Shirt objects. This course divides the code in a Java class file into four separate sections:

- The class declaration
- Field declarations (class attributes are called “fields”). Variables hold values, and the values can change during the course of the application. Fields are one type of variable. Local variables are another type of variable. Variables may also be initialized at declaration time.
- Methods (optional)
- Comments (optional)

Did You Know? A class does not have to contain both methods and attributes.

Structuring Classes



The programming code for a class is contained within a text file that must adhere to a certain structure. The example shows a `Shirt` class for all shirts that will appear in the Duke's Choice catalog. The `Shirt` class has several fields and one method, `displayInformation`, for printing the values of the fields.

Symbols Used in Defining a Java Source

- Braces { }
- Parentheses ( )
- Semicolons ;
- Commas ,
- Single quotation marks ' '
- Double quotation marks " "
- Single-line comment //

- Braces { } signify a block of code. The curly braces enclose the code for a specific method or the code for an entire class. (These are just some of the uses for curly braces.)
- Parentheses ( ) are used to indicate input data (also called “arguments”) that can be passed into a method.
- Semicolons (;) signify the end of a statement.
- Commas (,) can separate multiple arguments and values.
- Single quotation marks (' ') define single characters.
- Double quotation marks (" ") define a string of multiple characters.
- Double forward slashes (//) indicate a single-line comment.

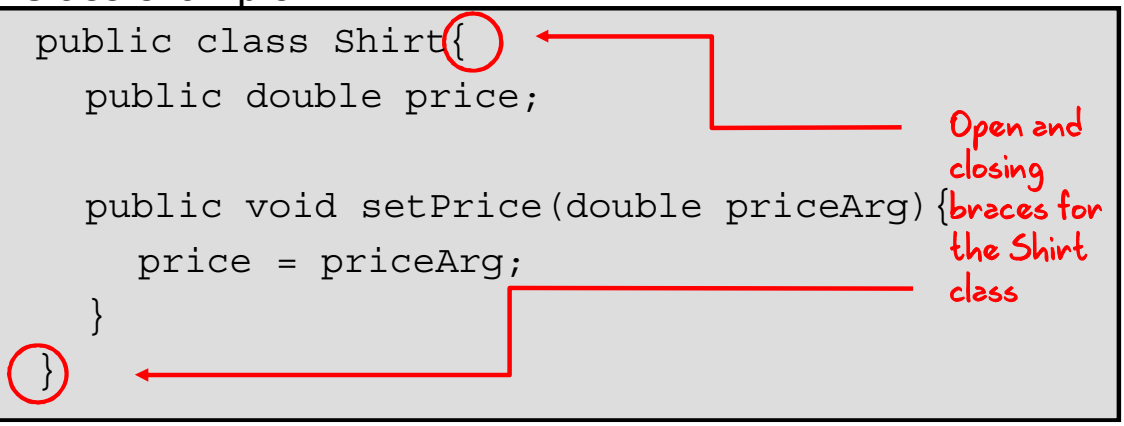
Putting It All Together

- Syntax for declaring a class:

```
[modifiers] class class_identifier
```

- Class example:

```
public class Shirt{
   public double price;

   public void setPrice(double priceArg){
      price = priceArg;
   }
}
```



You must declare a class for each class that you designed for the problem domain. For each class, you must write a class declaration. The syntax for declaring a class is:

```
[modifiers] class class_identifier
```

- The `[modifiers]` variable determines the accessibility that other classes have to this class. Modifiers are presented in detail later in this course. The `[modifiers]` variable is optional (indicated by the square brackets) and can be public, abstract, or final. For now, use the public modifier.
- The `class` keyword tells the compiler that the code block is a class declaration. Keywords are words that are reserved by the Java programming language for certain constructs.
- The `class identifier` is the name that you give to the class. Class naming guidelines are as follows:
  - Class names should be nouns, in mixed case, with the first letter of each word capitalized (for example, `MyClass`).
  - Class names should contain whole words. Avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as JVM or UML).

- The class example shown at the bottom of the slide is described as follows:
  - The `Shirt` class uses a class modifier of `public`, followed by the `class` keyword, followed by a class name of `Shirt`.
  - Curly braces are used to enclose the entire body of code for the Shirt class and also to enclose the body of code for the `setPrice` method.
  - Parentheses are used to enclose the argument passed into the `setPrice` method. (You will see more of the method syntax in a later slide.)
  - A semicolon is used at the end of the declaration of the field, `price`.

Requirements for Your Source File

In this course, you will be developing your classes so that the Java technology programming code that you write for each class is in its own text file or source code file. In the Java programming language, source code file names must match the public class name in the source code file, and must have a `.java` extension. For example, the `Shirt` class must be saved in a file called `Shirt.java`.

Quiz

Select the class declaration that adheres to the class-naming guidelines.

a. `class Shirt`

b. `public Class 501Pants`

c. `public Shirt`

d. `public Class Pants`

Answer: a

- The class definition is followed by an open curly brace ({) indicating the beginning of the `class_body`, the attribute variables, and the methods that compose the class. The braces { } around the `class_body` define where the class starts and ends.
- b is incorrect because the word `class` is capitalized.
- c is incorrect because class is not used in the class name.
- d is incorrect because the word `class` is capitalized.

Field Declarations and Assignments

```
public int shirtID = 0;
public String description = "-description required-";
public char colorCode = 'U';
public double price = 0.0;
public int quantityInStock = 0;
```


The field declarations and assignments block follows the first open curly brace ({). Generally, you set up all of the attribute variables for the class after this curly brace. Notice the semicolon at the end of each line of code in this example.

Comments

- Single-line:

```
public int shirtID = 0; // Default ID for the shirt
public double price = 0.0; // Default price for all shirts

// The color codes are R=Red, B=Blue, G=Green
```

- Traditional:

```
/*****************************************
* Attribute Variable Declaration Section *
*****************************************/
```


You should put comments in every class that you create to make it easier to determine what the program is doing. Commenting is particularly important in longer programs developed by large teams where several programmers need to read the code. Commenting helps with the maintenance of a program when new programmers need to determine what the code is doing.

Two main styles of comments can be used:

- **Single-line comments:** A `//` marker tells the compiler to ignore everything till the end of the current line. Many programmers also make their programs easier to read by using single-line comments to comment the first and last line of every class and method. For example, the `Shirt` class contains an end-of-line comment to indicate the end of the display method (Line 18):
  - } // end of display method

- **Traditional comments:** A /* character combination tells the compiler to ignore everything on all lines up to, and including, a comment termination marker (*/).
  - /***
  - * Attribute Variable Declaration Section *
  - ***/
  - Programmers often use traditional comments to provide details for a large block of code. In long programs, it can be very difficult to find the ending braces of the class. Commenting the structure that each ending brace belongs to makes reading and fixing errors much easier.

Did You Know? There is a third type of comment called a documentation comment. You can use a Java technology tool, the Javadoc tool, to create documentation for any of your classes that will be used by other programmers. In fact, all of the class library documentation that comes with the Java SE JDK was created using the Javadoc tool. Documentation comments must begin with a forward slash and two asterisks (/**) and end with an asterisk and a forward slash (*/). The previous example of a traditional comment would also be a valid documentation comment.

Topics

- Define a class, identify class components, and use variables
- Discuss methods and use of a `main` method
- Identify keywords
- Test and execute a simple Java program
- Describe some common causes of syntax errors
- Describe the purpose and features of an IDE debugger

Methods

- Syntax:

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

- Example:

```
public void displayInformation() {

    System.out.println("Shirt ID: " + shirtID);
    System.out.println("Shirt description:" + description);
    System.out.println("Color Code: " + colorCode);
    System.out.println("Shirt price: " + price);
    System.out.println("Quantity in stock: " + quantityInStock);

  } // end of display method
```

ORACLE

Methods follow the attribute variable declarations in a class. The syntax for methods is:

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

where:

- The `[modifiers]` represent several unique Java technology keywords that modify the way methods are accessed. Modifiers are optional (indicated by the square brackets).
- The `return type` indicates the type of value (if any) that the method returns. If the method returns a value, the type of the value must be declared. Returned values can be used by the calling method. Any method can return at most one value. If the method returns nothing, the keyword `void` must be used as the `return type`.

- The `method_identifier` is the name of the method.
- The `([arguments])` represent a list of variables whose values are passed to the method for use by the method. Arguments are optional (as indicated by the square brackets) because methods are not required to accept arguments. Also note that the parentheses are not optional. A method that does not accept arguments is declared with an empty set of parentheses.
- The `method_code_block` is a sequence of statements that the method performs. A wide variety of tasks can take place in the code block or body of the method. In the code sample, the `Shirt` class contains one method, the `displayInformation` method, which displays the values for the attributes of a shirt.

Within the `displayInformation` method, you see several lines of code that invoke the `System.out.println` method. This method is used to print out a specific string of data. You will use this method in the upcoming practice.

Topics

- Define a class, identify class components, and use variables
- Discuss methods and use of a main method
- Identify keywords
- Test and execute a simple Java program
- Describe some common causes of syntax errors
- Describe the purpose and features of an IDE debugger

Keywords

abstract	default	for	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	imports	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
continue	float	new	switch	while

ORACLE

Keywords are special, reserved words in the Java programming language that give instructions to the compiler. Keywords must not be used as identifiers for classes, methods, variables, and so on. The table contains all the Java technology keywords. `true`, `false`, and `null` might seem like keywords, but they are actually literals; you cannot use them as identifiers in your programs.

Topics

- Define a class, identify class components, and use variables
- Discuss methods and use of a `main` method
- Identify keywords
- Test and execute a simple Java program
- Describe some common causes of syntax errors
- Describe the purpose and features of an IDE debugger

Creating and Using a Test Class

Example:

```
class ShirtTest {

     public static void main (String args[]) {

       Shirt myShirt;
       myShirt= new Shirt();

       myShirt.displayInformation();


    }
   }
```


Most of the classes that you create throughout this course cannot be used (executed and tested) by themselves. Instead, you must execute another class to create an object instance of your class before your class can be tested. Throughout this course, you use a test or main class to test each of your classes. The code in the slide is an example of a test class for the `Shirt` class.

Every test class within this course should be named so that it can be recognized as the test class for a particular class you have written. Specifically, each test class name should consist of the name of the class you are testing, followed by the word "`Test`." For example, the class designed to test the `Shirt` class is called `ShirtTest`. Test classes have two distinct tasks to perform:

- Providing a starting point, called the `main` method, for your program
- Creating an object instance of your class and testing its methods

The `main` Method

- A special method that the JVM recognizes as the starting point for every Java technology program that runs from a command line
- Syntax:

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


The `main` method is a special method that the Java Virtual Machine recognizes as the starting point for every Java technology program that runs from the command line or from a prompt. Any program that you want to run from a command line or a prompt must have a `main` method.

Did You Know? Many of the Java technology classes that engineers create do not run within an operating system (OS). Remember applets? Applets run within a web browser and have their own unique starting method.

The syntax for the `main` method is:

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

The `main` method adheres to the syntax for all methods described earlier.

Specifically:

- The `main` method contains two required modifiers, `public` and `static`.
- The `main` method does not return any values, so it has a return type of `void`.
- The `main` method has a method identifier (name) of “main.”
- The `main` method accepts zero or more objects of type `String (String args[])`. This syntax allows you to type in values on the command line to be used by your program while it is running.

Compiling a Program

1. Go to the directory where the source code files are stored.
2. Enter the following command for each `.java` file you want to compile.

- Syntax:

```
javac filename
```

- Example:

```
javac Shirt.java
```


This is a review from the Lesson 2 activity. Compiling converts the class files that you write into bytecode that can be executed by a Java Virtual Machine. Remember the rules for naming your Java source files. If a source file contains a public class, the source file must use the same name as the public class, with a `.java` extension. For example, the class `Shirt` must be saved in a file called `Shirt.java`. To compile the `Shirt` and `ShirtTest` source code files:

1. Go to the directory where the source code files are stored.
2. Enter the following command for each `.java` file that you want to compile:

```
javac filename
```

Example:

```
javac Shirt.java
```

After the compilation has finished, and assuming no compilation errors have occurred, you should have a new file called *`classname.class`* in your directory for each source code file that you compiled. If you compile a class that references other objects, the classes for those objects are also compiled (if they have not been compiled already). For example, if you compiled the `ShirtTest.java` file, (which references a Shirt object), you have a `Shirt.class` and `ShirtTest.class` file.

Executing (Testing) a Program

1. Go to the directory where the class files are stored.
2. Enter the following for the class file that contains the `main` method:

- Syntax:

```
java classname
```

- Example:

```
java ShirtTest
```

- Output:

```
Shirt ID: 0
Shirt description:-description required-
Color Code: U
Shirt price: 0.0
Quantity in stock: 0
```

ORACLE

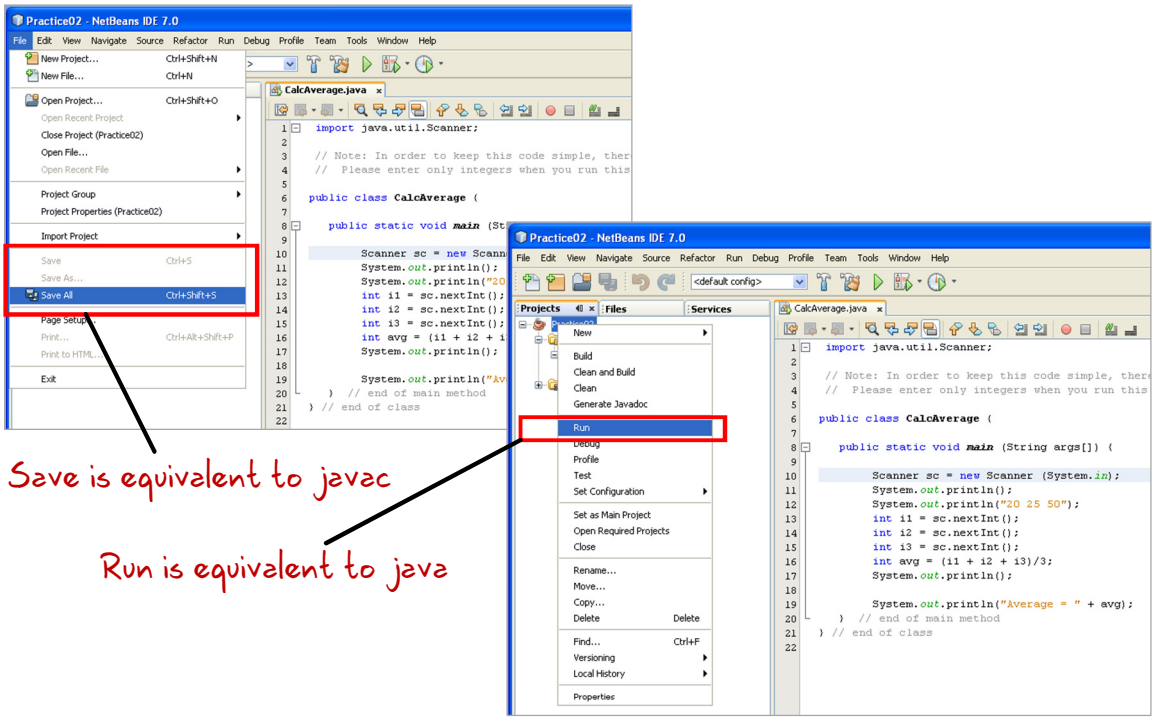
- When you have successfully compiled your source code files, you can execute and test them using the Java Virtual Machine. To execute and test your program:
  1. Go to the directory where the class files are stored.
  2. Enter the following command for the class file that contains the `main` method:

     ```
     java classname
     ```
- Example:

  ```
  java ShirtTest
  ```
- This command runs the `ShirtTest` class. As mentioned previously, the `ShirtTest` class creates an instance of the Shirt object, using the `Shirt` class. All Shirt objects have one method, the `display` method, which prints the values of their attribute variables, as in this example:

  ```
  Shirt ID: 0
  Shirt description:-description required-
  Color Code: U
  Shirt price: 0.0
  Quantity in stock: 0
  ```

Compiling and Running a Program by Using an IDE

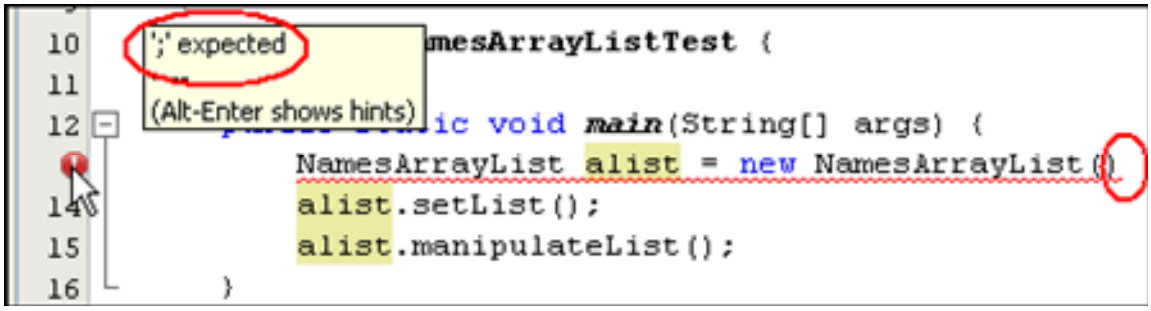


Save does the `javac <classname(s)>` for all `.java` files in the project. Either Run File (to run a single file) or the Run button (to run the "main" class for the entire project) does the `java <classname>`. Be sure to observe any red bubble indicators in the code editor to locate syntax errors.

Topics

- Define a class, identify class components, and use variables
- Discuss methods and use of a `main` method
- Identify keywords
- Test and execute a simple Java program
- Describe some common causes of syntax errors
- Describe the purpose and features of an IDE debugger

Avoiding Syntax Problems



ORACLE

Most Java editors check the code syntax and show alerts by using icons and red underlines where there are errors in the code. To avoid syntax problems, do the following:

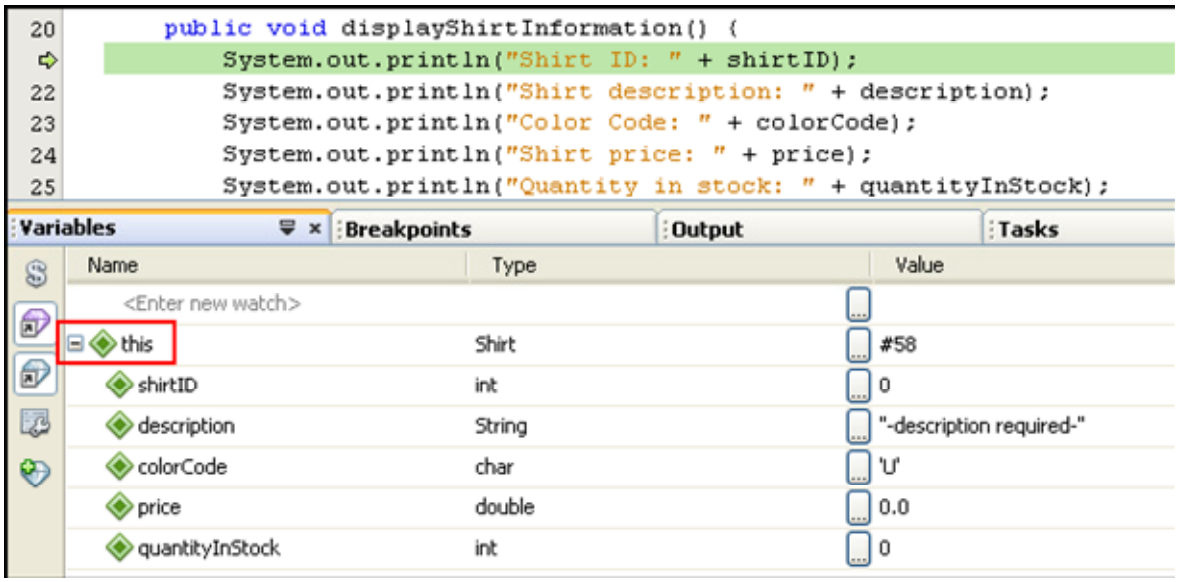
- Be sure to observe any red bubble indicators in the code editor to locate syntax errors.
- Be sure that you have a semicolon at the end of every line where one is required.
- Be sure that you have an even number of symbols such as curly braces, brackets, and quotation marks.

The screenshot shows an error in Line 13, in which there is a missing semicolon. If you hover the cursor over the red bubble, the editor offers a suggestion for fixing the error.

Topics

- Define a class, identify class components, and use variables
- Discuss methods and use of a `main` method
- Identify keywords
- Test and execute a simple Java program
- Describe some common causes of syntax errors
- Describe the purpose and features of an IDE debugger

Working with an IDE Debugger



A debugger lets you place breakpoints in your source code, add field watches, step through your code, run into methods, take snapshots, and monitor execution as it occurs. You can also attach the NetBeans debugger to an already running process. Other features include:

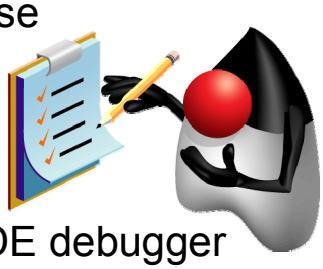
- **Configurable debugger:** You can configure breaking/suspending behavior, specify Variable Formatters, and skip methods and packages by using Step Filters.
- **Debugging window:** The Debugging window integrates the Sessions, Threads, and Call Stack views.
- **Configurable breakpoints:** Configure these custom breakpoints to be triggered by conditions and events such as uncaught exceptions, class loading, or variable access.
- **Expression evaluation:** Evaluate Java-syntax expressions assigned to watches and conditional breakpoints “live” while stepping through your code.
- **Expression stepping:** Step over individual expressions within a statement.
- **Multi-session debugging:** Debug several processes at the same time.
- **HeapWalker:** Watch references to objects while debugging a program.

In the screenshot, you see a program in the middle of a debug session. A breakpoint has been set on the highlighted line and the execution has stopped at that line. Notice that in the Variables window at the bottom of the screen, you see the fields of the class that is currently being executed (referenced by the keyword `this`). During a debug session, you can change the values of these fields to try out different “What If” scenarios. This is helpful in solving logic problems.

Summary

In this lesson, you should have learned how to:

- Define a class
- Identify the components of a class
- Explain the term *object*
- Describe the purpose of a variable
- Discuss methods and describe how to use a `main` method
- Describe the elements that comprise a Java class, such as declarations, return values, and the proper use of brackets and braces
- Identify keywords and describe their purpose
- Test and execute a simple program
- Describe some common causes of syntax errors
- Describe the purpose and features of an IDE debugger



ORACLE

Practice 4-1: Viewing and Adding Code to an Existing Java Program

This practice covers creating a new project.

Practice 4-2: Creating and Compiling a Java Program

This practice covers creating a new project by using the `Shirt` class.

Practice 4-3: Exploring the Debugger

This practice covers the following topics:

- Setting a breakpoint
- Using a step over

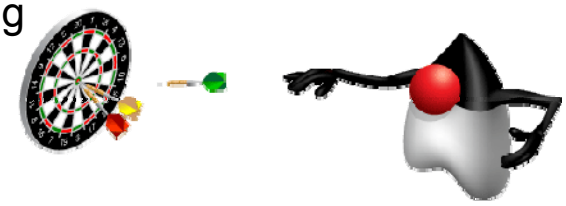
5

Declaring, Initializing, and Using Variables

Objectives

After completing this lesson, you should be able to:

- Identify the uses of variables and define the syntax for a variable
- List the eight Java programming language primitive data types
- Declare, initialize, and use variables and constants according to Java programming language guidelines and coding standards
- Modify variable values by using operators
- Use promotion and type casting



ORACLE

Relevance

- A variable refers to something that can change. Variables can contain one of a set of values. Where have you seen variables before?
- What types of data do you think variables can hold?

Topics

- Identify the uses of variables and define the syntax for a variable
- List the eight Java programming language primitive data types
- Declare, initialize, and use variables and constants
- Modify variable values by using operators
- Use promotion and type casting

Identifying Variable Use and Syntax

Example:

```
public class Shirt {

 public int shirtID = 0; // Default ID for the shirt

 public String description = "-description required-"; // default
 // The color codes are R=Red, B=Blue, G=Green, U=Unset
public char colorCode = 'U';

 public double price = 0.0; // Default price for all shirts

 public int quantityInStock = 0; // Default quantity for all shirts

// This method displays the values for an item
  public void displayInformation() {

System.out.println("Shirt ID: " + shirtID);
```


You use variables for storing and retrieving data for your program. Objects store their individual states in fields. Fields are also called "instance variables" because their values are unique to each individual instance of a class. The code example shows a `Shirt` class that declares several non-static fields (such as `price`, `shirtID`, and `colorCode` in the `Shirt` class). When an object is instantiated from a class, these variables contain data specific to a particular object instance of the class. For example, one instance of the `Shirt` class might have the value of `7` assigned to the `quantityInStock` non-static field, while another instance of the `Shirt` class might have the value of `100` assigned to the `quantityInStock` non-static field.

Identifying Variable Use and Syntax

Example:

```
public void displayDescription {
        String displayString = "";
        displayString = "Shirt description: " + description;
        System.out.println(displayString);
  }
```


Your programs can also have variables defined within methods. These variables are called local variables because they are available only locally within the method in which they are declared.

Note: Throughout this course, we might use the terms “variables” or “fields” to refer to variables. If the situation requires, we will refer to “local variable” when it applies.

Uses of Variables

- Holding unique data for an object instance
- Assigning the value of one variable to another
- Representing values within a mathematical expression
- Printing the values to the screen
- Holding references to other objects

Variables are used extensively in the Java programming language for tasks such as:

- Holding unique attribute data for an object instance (as you have seen with the price and ID variables)
- Assigning the value of one variable to another
- Representing values within a mathematical expression
- Printing the values to the screen. For example, the `Shirt` class uses the `price` and `ID` variables to print the price and ID values for the shirt:

```
System.out.println("Shirt price: " + price);
System.out.println("Shirt ID: " + shirtID);
```

- Holding references to other objects

Variable Declaration and Initialization

- Syntax (fields):

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

- Syntax (local variables):

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

- Examples:

```
public int shirtID = 0;
public String description = "-description required-";
public char colorCode = 'U';
public double price = 0.0;
public int quantityInStock = 0;
```

Attribute variable declaration and initialization follow the same general syntax. The syntax for field declaration and initialization is:

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

The syntax for initializing a variable inside of a method is:

```
identifier = value;
```

The syntax for declaring and initializing a variable inside of a method is:

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

where:

- The `[modifiers]` represent several special Java technology keywords, such as `public` and `private`, that modify the access that other code has to a field. Modifiers are optional (as indicated by the square brackets). For now, all of the fields you create should have a `public` modifier.
- The `type` represents the type of information or data held by the variable. Some variables contain characters, some contain numbers, and some are Boolean and can contain only one of two values. All variables must be assigned a type to indicate the type of information that they contain.

Note: Do not use modifiers with local variables (variables declared within methods).

- The `identifier` is the name you assign to the variable that is of type `type`.
- The `value` is the value you want to assign to the variable. The value is optional because you do not need to assign a value to a variable at the time that you declare the variable.

The following are the declarations for the fields in the `Shirt` class:

```
public int shirtID = 0;
public String description = "-description required-";
public char colorCode = 'U';
public double price = 0.0;
public int quantityInStock = 0;
```

Topics

- Identify the uses of variables and define the syntax for a variable
- List the eight Java programming language primitive data types
- Declare, initialize, and use variables and constants
- Modify variable values by using operators
- Use promotion and type casting

Describing Primitive Data Types

- Integral types (`byte`, `short`, `int`, and `long`)
- Floating point types (`float` and `double`)
- Textual type (`char`)
- Logical type (`boolean`)

Many of the values in Java technology programs are stored as primitive data types. The slide lists the eight primitive types built in to the Java programming language.

Integral Primitive 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 0b10 (binary number)
`short`	16 bits	-2^{15} to 2^{15} -1 (-32,768 to 32,767, or 65,535 possible values)	2 -32699
`int` (default type for integral literals)	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 147334778 123_456_678

ORACLE

There are four integral primitive types in the Java programming language, identified by the keywords `byte`, `short`, `int`, and `long`. These types store numbers that do not have decimal points. If you need to store people's ages, for example, a variable of type `byte` would work because `byte` types can accept values in that range. When you specify a literal value for a `long` type, put a capital `L` to the right of the value to explicitly state that it is a `long` type. Integer literals are assumed by the compiler to be of type `int` unless you specify otherwise using an `L` indicating `long` type.

A new SE 7 feature allows you to express any of the integral types as a binary (0s and 1s). For instance, a binary expression of the number 2 is shown as an allowed value of the `byte` integral type. The binary value is `0b10`. Notice that this value starts with `0b` (that is, zero followed by either a lowercase or uppercase letter *B*). This indicates to the compiler that a binary value follows.

Another new feature of SE 7 is seen in the `int` row. The ability to include underscores in a lengthy `int` number helps with readability of the code. For instance, you might use this to make it easier to read a large integral number, substituting underscores for commas. The use of the underscore has no effect on the numerical value of the `int`, nor does it appear if the variable is printed to the screen.

Integral Primitive Types

Type	Length	Range	Examples of Allowed Literal Values
`long`	64 bits	-2^{63} to $2^{63}-1$ (-9,223,372,036854,775,808 to 9,223,372,036854,775,807, or 18,446,744,073,709,551,616 possible values)	2 -2036854775808L 1L

The `Shirt` class contains two attributes of type `int` to hold the values for a `shirtID` and the quantity in stock, and literal values are used to supply a default starting value of zero (`0`) for each.

```
public int shirtID = 0; // Default ID for the shirt
public int quantityInStock = 0; // Default quantity for all shirts
```

Note: The only reason to use the `byte` and `short` types in programs is to save memory consumption. Because most modern desktop computers contain an abundance of memory, most desktop application programmers do not use `byte` and `short` types. This course uses primarily `int` and `long` types in the examples.

Floating Point Primitive Types

Type	Float Length	Examples of Allowed Literal Values
`float`	32 bits	99F -327456,99.01F 4.2E6F (engineering notation for $4.2 * 10^6$)
`double` (default type for floating point literals)	64 bits	-1111 2.1E12 99970132745699.999

```
public double price = 0.0; // Default price for all shirts
```


There are two types for floating point numbers, `float` and `double`. These types are used to store numbers with values to the right of the decimal point, such as 12.24 or 3.14159. When you specify a literal value for a `float` type, put a capital `F` (`float`) to the right of the value to explicitly state that it is a `float` type, not a `double` type.

Literal values for floating point types are assumed to be of type `double` unless you specify otherwise, using the `F` indicating `float` type. The `Shirt` class shows the use of one `double` literal value to specify the default value for the price:

```
public double price = 0.0; // Default price for all shirts
```

Note: Use the `double` type when a greater range or higher accuracy is needed.

Textual Primitive Type

- The only primitive textual data type is `char`.
- It is used for a single character (16 bits).
- Example:
  - `public char colorCode = 'U';`

Another data type you use for storing and manipulating data is single-character information. The primitive type used for storing a single character (such as a `y`) is `char`, which is 16 bits in size. The `Shirt` class shows the use of one textual literal value to specify the default value for a `colorCode`:

```
public char colorCode = 'U';
```

When you assign a literal value to a `char` variable, such as `t`, you must use single quotation marks around the character: `'t'` Using single quotation marks around the character clarifies for the compiler that the `t` is just the literal value `t`, rather than a variable `t` that represents another value.

The `char` type 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 that the programming language uses.

Did You Know: Many computer languages use American Standard Code for Information Interchange (ASCII), an 8-bit character set that has an entry for every English character, punctuation mark, number, 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. Therefore, your programs can be written so that they work correctly and display the correct language for most countries. Unicode contains a subset of ASCII (the first 128 characters).

Logical Primitive Type

- The only data type is `boolean`.
- It can store only `true` or `false`.
- It holds the result of an expression that evaluates to either `true` or `false`.

Computer programs must often make decisions. The result of a decision, whether the statement in the program is true or false, can be saved in Boolean variables. Variables of type `boolean` can store only:

- The Java programming language literals `true` or `false`
- The results of an expression that only evaluates to `true` or `false`. For example, if the variable answer is equal to 42, then the expression "`if answer < 42`" evaluates to a `false` result.

Topics

- Identify the uses of variables and define the syntax for a variable
- List the eight Java programming language primitive data types
- Declare, initialize, and use variables and constants
- Modify variable values by using operators
- Use promotion and type casting

Naming a Variable

Rules:

- Variable identifiers must start with either an uppercase or lowercase letter, an underscore (_), or a dollar sign ($).
- Variable identifiers cannot contain punctuation, spaces, or dashes.
- Java technology keywords cannot be used.

As with a class or method, you must assign an identifier or a name to each variable in your program. Remember, the purpose of the variable is to act as a mechanism for storing and retrieving values. Therefore, you should make variable identifiers simple but descriptive. For example, if you store the value of an item ID, you might name the variable `myID`, `itemID`, `itemNumber`, or anything else that clarifies the use of the variable to yourself and to others reading your program.

Did You Know? Many programmers follow the convention of using the first letter of the type as the identifier: `int i`, `float f`, and so on. This convention is acceptable for small programs that are easy to decipher, but generally you should use more descriptive identifiers.

Naming a Variable

Guidelines:

- Begin each variable with a lowercase letter. Subsequent words should be capitalized (for example, `myVariable`).
- Choose names that are mnemonic and that indicate to the casual observer the intent of the variable.

Assigning a Value to a Variable

- Example:
  - `double price = 12.99;`
- Example (Boolean):
  - `boolean isOpen = false;`

You can assign a value to a variable at the time when the variable is declared, or you can assign the variable later. To assign a value to a variable during declaration, add an equal sign (=) after the declaration, followed by the value to be assigned. For example, the price field in the `Shirt` class could be assigned the value `12.99` as the price for a Shirt object.

```
double price = 12.99;
```

An example of Boolean variable declaration and assignment is:

```
boolean isOpen = false;
```

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 the preceding example, you could say, "12.99 is assigned to price." Operators such as the assignment operator (=) are presented later in this course.

Note: Fields are automatically initialized: integral types are set to `0`, floating point types are set to `0.0`, the `char` type is set to `\u0000`, and the `boolean` type is set to `false`. However, you should explicitly initialize your fields so that other people can read your code. Local variables (declared within a method) must be explicitly initialized before being used.

Declaring and Initializing Several Variables in One Line of Code

- Syntax:
  - *type identifier = value [, identifier = value];*
- Example:
  - `double price = 0.0, wholesalePrice = 0.0;`

You can declare one or more variables on the same line of code, but only if they are all of the same type. The syntax for declaring several variables in one line of code is:

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

Therefore, if there are separate retail and wholesale prices in the `Shirt` class, they might be declared as follows:

```
double price = 0.0, wholesalePrice = 0.0;
```

Additional Ways to Declare Variables and Assign Values to Variables

- Assigning literal values:
  - `int ID = 0;`
  - `float pi = 3.14F;`
  - `char myChar = 'G';`
  - `boolean isOpen = false;`
- Assigning the value of one variable to another variable:
  - `int ID = 0;`
  - `int saleID = ID;`

You can assign values to variables by using several different approaches:

- Assigning literal values directly to variables (as has been described throughout this lesson):

```
int ID = 0;
float pi = 3.14F;
char myChar = 'G';
boolean isOpen = false;
```

- Assigning the value of one variable to another variable:

```
int ID = 0;
int saleID = ID;
```

The first line of code creates an integer called `ID` and uses it to store the number 0. The second line of code creates another integer called `saleID` and uses it to store the same value as `ID` (0). If the contents of `ID` are changed later, the contents of `saleID` do not automatically change. Even though the two integers currently have the same value, they can be independently changed later in a program.

Additional Ways to Declare Variables and Assign Values to Variables

- Assigning the result of an expression to integral, floating point, or Boolean variables:
  - `float numberOrdered = 908.5F;`
  - `float casePrice = 19.99F;`
  - `float price = (casePrice * numberOrdered);`
  - `int hour = 12;`
  - `boolean isOpen = (hour > 8);`
- Assigning the return value of a method call to a variable

Assigning the result of an expression to integral, floating point, or Boolean type variables:
In 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);
```

Assigning the return value of a method call to a variable: This approach is described later in this course.

Constants

- Variable (can change):
  - `double salesTax = 6.25;`
- Constant (cannot change):
  - `final int NUMBER_OF_MONTHS = 12;`
- Guideline: Constants should be capitalized, with words separated by an underscore (_).

In this lesson, you have learned about variables that have values that you can change. In this section, you learn how to use constants to represent values that cannot change.

Assume that you are writing part of a scheduling application, and you need to refer to the number of months in a year. Make the variable a constant by using the `final` keyword to inform the compiler that you do not want the value of the variable to be changed after it has been initialized. Also, by convention, name the constant identifier using all capital letters, with underscores separating words, so that it is easy to determine that it is a constant:

```
final int NUMBER_OF_MONTHS = 12;
```

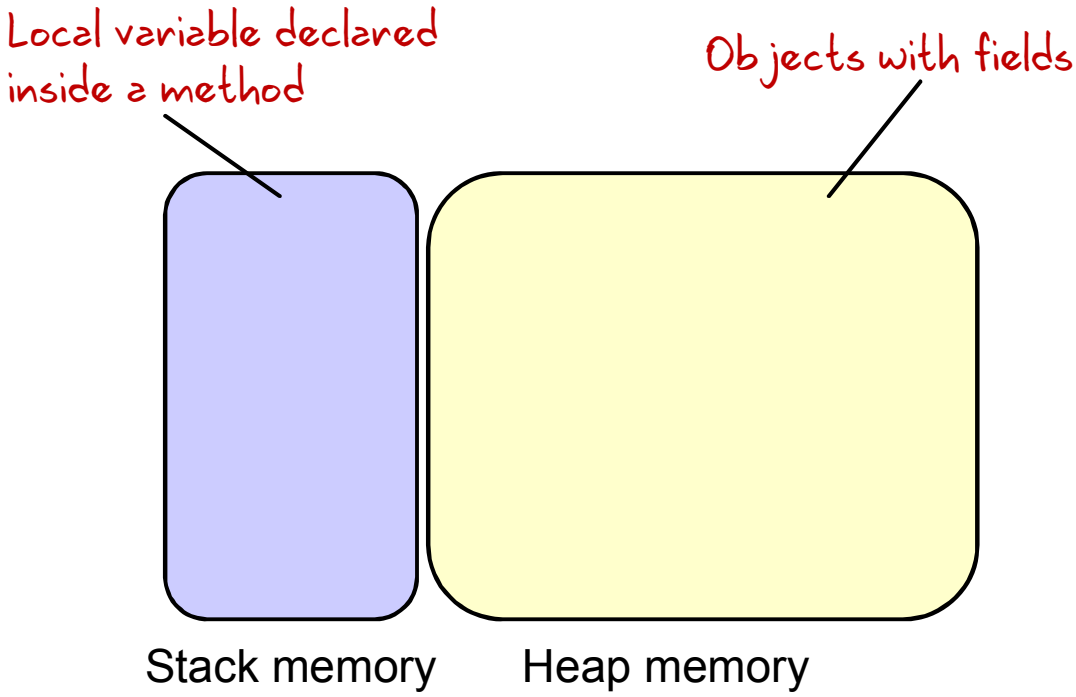
Any values that tend to change rarely, if ever, are good candidates for a constant variable (for example, `MAX_COUNT`, `PI`, and so on).

If someone attempts to change the value of a constant after it has already been assigned a value, the compiler gives an error message. If you modify your code to provide a different value for the constant, you need to recompile your program.

Guidelines for Naming Constants

You should name constants so that they can be easily identified. Generally, constants should be capitalized, with words separated by an underscore (_).

Storing Primitives and Constants in Memory



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

The figure shows that local variables are stored separately (on the stack) from fields (on the heap). Objects and their fields and methods are usually stored in heap memory. Heap memory consists of dynamically allocated memory chunks containing information used to hold objects (including their fields and methods) while they are needed by your program. Other variables are usually stored in stack memory. Stack memory stores items that are used for only a brief period of time (shorter than the life of an object), such as variables declared inside of a method.

Quiz

The variable declaration `public int myInteger=10;` adheres to the variable declaration and initialization syntax.

a. True
b. False

Answer: a

Topics

- Identify the uses of variables and define the syntax for a variable
- List the eight Java programming language primitive data types
- Declare, initialize, and use variables and constants
- Modify variable values by using operators
- Use promotion and type casting

Standard Mathematical Operators

Purpose	Operator	Example	Comments
Addition	+	`sum = num1 + num2;` If `num1` is 10 and `num2` is 2, `sum` is 12.	
Subtraction	-	`diff = num1 - num2;` If `num1` is 10 and `num2` is 2, `diff` is 8.	
Multiplication	*	`prod = num1 * num2;` If `num1` is 10 and `num2` is 2, `prod` is 20.	
Division	/	`quot = num1 / num2;` If `num1` is 31 and `num2` is 6, `quot` is 5.	Division returns an integer value (with no remainder).

ORACLE

Standard Mathematical Operators

Purpose	Operator	Example	Comments
Remainder	`%`	`mod = num1 % num2;` If `num1` is 31 and `num2` is 6, `mod` is 1.	Remainder finds the remainder of the first number divided by the second number. 5 R(1) 6 | 31 30 --- 1 Remainder always gives an answer with the same sign as the first operand.

Programs do a lot of mathematical calculating, from the simple to the complex. Arithmetic operators let you specify how the numerical values within variables should be evaluated or combined. The standard mathematical operators, often called binary operators, used in the Java programming language are shown in the tables.

Note: The `%` is known as a modulus.

Increment and Decrement Operators (++ and --)

The long way:

```
age = age + 1;
```

or

```
count = count - 1;
```


A common requirement in programs is to add or subtract 1 from the value of a variable. You can do this by using the + operator as follows:

```
age = age + 1;
```

Increment and Decrement Operators (++ and --)

The short way:

Operator	Purpose	Example	Notes

`++`	Pre-increment (*`++variable`*)	`int i = 6;`
		`int j = ++i;`
		`i is 7, j is 7`	

	Post-increment (*`variable++`*)	`int i = 6;`
		`int j = i++;`
		`i is 7, j is 6`
			The value of `i` is assigned to `j` before `i` is incremented. Therefore, `j` is assigned `6`.

ORACLE

However, incrementing or decrementing by 1 is such a common action that there are specific unary operators for it: the increment (`++`) and decrement (`--`) operators. These operators can come before (pre-increment and pre-decrement) or after (post-increment and post-decrement) a variable.

The line of code in the previous slide, in which the age was incremented by 1, can also be written as follows:

```
age++; or ++age;
```

Increment and Decrement Operators (++ and --)

Operator	Purpose	Example	Notes

`--`	Pre-decrement (*`--variable`*)	`int i = 6;`
		`int j = --i;`
		`i is 5, j is 5`	

	Post-decrement (*`variable--`*)	`int i = 6;`
		`int j = i--;`
		`i is 5, j is 6`
			The value `i` is assigned to `j` before `i` is decremented. Therefore, `j` is assigned `6`.

ORACLE

Use these operators within an expression cautiously. With the prefix form, the operation (increment or decrement) is applied before any subsequent calculations or assignments. With the postfix form, the operation is applied after the subsequent calculations or operations, so that the original value—and not the updated value—is used in subsequent calculations or assignments. The table shows increment and decrement operators.

Increment and Decrement Operators (++ and --)

Examples:

```
int count=15;
int a, b, c, d;
a = count++;
b = count;
c = ++count;
d = count;
System.out.println(a + ", " + b + ", " + c + ", " + d);
```


The example in the slide shows basic use of the increment and decrement operators:

```
int count=15;
int a, b, c, d;
a = count++;
b = count;
c = ++count;
d = count;
System.out.println(a + ", " + b + ", " + c + ", " + d);
```

The result of this code fragment is:

```
15, 16, 17, 17
```

Discussion: What is the result of the following code?

```
int i = 16;
System.out.println(++i + " " + i++ + " " + i);
```

Operator Precedence

Here is an example of the need for rules of precedence.

Is the answer to the following problem 34 or 9?

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

Operator Precedence

Rules of precedence:

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

ORACLE

In a complex mathematical statement with multiple operators on the same line, how does the computer pick which operator it should use first? To make mathematical operations consistent, the Java programming language follows the standard mathematical rules for operator precedence. Operators are processed in the following order:

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

If standard mathematical operators of the same precedence appear 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;
```

In this example, it is not clear what the author intended. The result can be evaluated in two ways:

- The expression result when evaluated strictly from left to right: `34`

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

- The actual expression result when evaluated according to rules of precedence, indicated by the parentheses: `9`

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

Using Parentheses

Examples:

```
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;
```


Your expression will be automatically evaluated with the rules of precedence. However, you should 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;
```

Topics

- Identify the uses of variables and define the syntax for a variable
- List the eight Java programming language primitive data types
- Declare, initialize, and use variables and constants
- Modify variable values by using operators
- Use promotion and type casting

Using Promotion and Type Casting

- Example of potential issue:

```
int num1 = 53; // 32 bits of memory to hold the value
int num2 = 47; // 32 bits of memory to hold the value
byte num3; // 8 bits of memory reserved
num3 = (num1 + num2); // causes compiler error
```

- Example of potential solution:

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

ORACLE

Assigning a variable or an expression to another variable can lead to a mismatch between the data types of the calculation and the storage location that you are using to save the result. Specifically, the compiler will either recognize that precision will be lost and not allow you to compile the program, or the result will be incorrect. To fix this problem, variable types have to be either promoted to a larger size type, or type cast to a smaller size type.

For example, consider the following assignment:

```
int num1 = 53; // 32 bits of memory to hold the value
int num2 = 47; // 32 bits of memory to hold the value
byte num3; // 8 bits of memory reserved
num3 = (num1 + num2); // causes compiler error
```

This code should work, because a `byte`, though smaller than an `int`, is large enough to store a value of `100`. However, the compiler will not make this assignment and, instead, issues a "possible loss of precision" error because a `byte` value is smaller than an `int` value. To fix this problem, you can either type cast the right-side data type down to match the left-side data type, or declare the variable on the left side (`num3`) to be a larger data type, such as an `int`.

This problem is fixed by changing `num3` to an `int`:

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

Promotion

- Automatic promotions:
  - If you assign a smaller type to a larger type
  - If you assign an integral type to a floating point type
- Examples of automatic promotions:

```
long big = 6;
```


In some circumstances, the compiler changes the type of a variable to a type that supports a larger size value. This action is referred to as a promotion. Some promotions are done automatically by the compiler if data would not be lost by doing so. These promotions include:

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

The following example contains a literal (an `int`) that will automatically be promoted to another type (a `long`) before the value (`6`) is assigned to the variable (`big` of type `long`). The following examples show what will be automatically promoted by the compiler and what will not be promoted:

```
long big = 6;
```

Because `6` is an `int` type, promotion works because the `int` value is converted to a `long` value.

Caution: Prior to being assigned to a variable, the result of an equation is placed in a temporary location in memory. The location's size is always equal to the size of an `int` type or the size of the largest data type used in the expression or statement. For example, if your equation multiplies two `int` types, the container size will be an `int` type in size, or 32 bits.

If the two values that you multiply yield a value that is beyond the scope of an `int` type, (such as 55555*66666=3,703,629,630, which is too big to fit in an `int` type), the `int` value must be truncated to fit the result into the temporary location in memory. This calculation ultimately yields an incorrect answer because the variable for your answer receives a truncated value (regardless of the type used for your answer).

To solve this problem, set at least one of the variables in your equation to the `long` type to ensure the largest possible temporary container size.

Type Casting

- Syntax:

```
identifier = (target_type) value
```

- Example of potential issue:

```
int num1 = 53; // 32 bits of memory to hold the value
int num2 = 47; // 32 bits of memory to hold the value
byte num3; // 8 bits of memory reserved
num3 = (num1 + num2); // causes compiler error
```

- Example of potential solution:

```
int num1 = 53; // 32 bits of memory to hold the value
int num2 = 47; // 32 bits of memory to hold the value
byte num3; // 8 bits of memory reserved
num3 = (byte)(num1 + num2); // no data loss
```

ORACLE

Type casting lowers the range of a value, quite literally chopping it down to a smaller size, by changing the type of the value (for example, by converting a `long` value to an `int` value). You do this so that you can use methods that accept only certain types as arguments, so that you can assign values to a variable of a smaller data type, or to save memory. Put the `target_type` (the type that the value is being type cast to) in parentheses in front of the item that you are type casting. The syntax for type casting a value is:

```
identifier = (target_type) value
```

where:

- The `identifier` is the name you assign to the variable
- The `value` is the value you want to assign to the identifier
- The `(target_type)` is the type to which you want to type cast the value. Notice that the `target_type` must be in parentheses.

For example, consider the following assignment:

```
int num1 = 53; // 32 bits of memory to hold the value
int num2 = 47; // 32 bits of memory to hold the value
byte num3; // 8 bits of memory reserved
num3 = (num1 + num2); // causes compiler error
```

The compiler error is fixed by type casting the result to a `byte`.

```
int num1 = 53; // 32 bits of memory to hold the value
int num2 = 47; // 32 bits of memory to hold the value
byte num3; // 8 bits of memory reserved
num3 = (byte)(num1 + num2); // no data loss
```

Caution: Use type casting with caution. For example, if larger numbers for `num1` and `num2` were used, the type cast to a `byte` would truncate part of the data, resulting in an incorrect answer.

Type Casting

Examples:

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


Other potential problems are as follows:

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

If you type cast a `float` or `double` value with a fractional part to an integral type such as an `int`, all decimal values are lost. However, this method of type casting is sometimes useful if you want to truncate the number down to the whole number, for example 51.9 becomes 51.

Compiler Assumptions for Integral and Floating Point Data Types

- Example of potential problem:

```
short a, b, c;
a = 1 ;
b = 2 ;
c = a + b ; //compiler error
```

- Example of potential solutions:
  - Declare `c` as an `int` type in the original declaration:

```
int c;
```

- Type cast the (`a+b`) result in the assignment line:

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


The Java technology compiler makes certain assumptions when it evaluates expressions. You must understand these assumptions to make the appropriate type casts or other accommodations.

Integral Data Types and Operations

Most operations result in `int` or `long`:

- `byte`, `char`, and `short` values are promoted to `int` before the operation.
- If either argument is of the `long` type, the other is also promoted to `long`, and the result is of the `long` type.

```
byte b1 = 1, b2 = 2, b3;
b3 = b1 + b2;   // Error:  result is an int but b3 is a byte
```

Promoting Floats

- If an expression contains a `float`, the entire expression is promoted to `float`. All literal floating-point values are viewed as `double`.

In the following example, an error occurs because two of the three operands (a and b) are automatically promoted from a short type to an int type before they are added:

```
short a, b, c;
a = 1 ;
b = 2 ;
c = a + b ; //compiler error
```

In the last line, the values of a and b are converted to int types and the converted values are added to give an int result. Then the assignment operator (=) attempts to assign the int result to the short variable (c). However, this assignment is illegal and causes a compiler error.

The code works if you do either of the following:

- Declare c as an int in the original declaration:

```
int c;
```

- Type cast the (a+b) result in the assignment line:

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

Floating Point Data Types and Assignment

- Example of potential problem:

```
float float1 = 27.9; //compiler error
```

- Example of potential solutions:
  - The `F` notifies the compiler that 27.9 is a `float` value:

```
float float1 = 27.9F;
```

- `27.9` is cast to a `float` type:

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


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

For example, the following line causes a compiler error. Because `27.9` is assumed to be a `double` type, a compiler error occurs because a `double` type value cannot fit into a `float` variable.

```
float float1 = 27.9;//compiler error
```

Both of the following work correctly:

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

```
float float1 = 27.9F;
```

- `27.9` is cast to a `float` type:

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

Example

```
public class Person {

  public int ageYears = 32;

  public void calculateAge() {

    int ageDays = ageYears * 365;
    long ageSeconds = ageYears * 365 * 24L * 60 * 60;

    System.out.println("You are " + ageDays + " days old.");
    System.out.println("You are " + ageSeconds + " seconds
old.");

 } // end of calculateAge method
} // end of class
```

ORACLE

The code example uses principles from this section to calculate a person's age in days and seconds.

Quiz

Which statement is true?

a. There are eight primitive types built in to the Java programming language.
b. `byte`, `short`, `char`, and `long` are the four integral primitive data types in the Java programming language.
c. A `boolean` type variable holds `true`, `false`, and `nil`.
d. `long=10;` is a valid variable name that adheres to the variable declaration and initialization syntax.

Answer: a

- a is correct.
- b is incorrect. It should be `byte`, `short`, `int`, and `long`.
- c is incorrect because a `boolean` type variable holds only `true` and `false`.
- d is incorrect because `long` is a reserved keyword.

Summary

In this lesson, you should have learned how to:

- Identify the uses of variables and define the syntax for a variable
- List the eight Java programming language primitive data types
- Declare, initialize, and use variables and constants according to Java programming language guidelines and coding standards
- Modify variable values by using operators
- Use promotion and type casting

Practice 5-1: Declaring Field Variables in a Class

This practice covers the following topics:

- Creating a class containing several fields
- Declaring the field variables and initializing them
- Testing the class by running a provided test program

ORACLE

Practice 5-2: Using Operators and Performing Type Casting to Prevent Data Loss

This practice covers the following topics:

- Using operators
- Using type casting

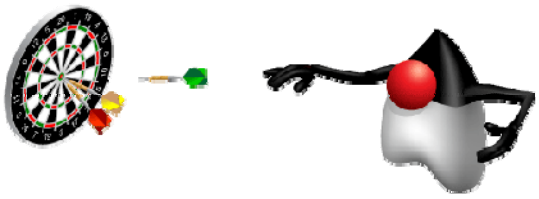
6

Working with Objects

Objectives

After completing this lesson, you should be able to:

- Declare, instantiate, and initialize object reference variables
- Compare how object reference variables are stored in relation to primitive variables
- Access fields on objects
- Call methods on objects
- Create a String object
- Manipulate data by using the String class and its methods
- Manipulate data by using the StringBuilder class and its methods
- Use the Java API documentation to explore the methods of a foundation class



ORACLE

Topics

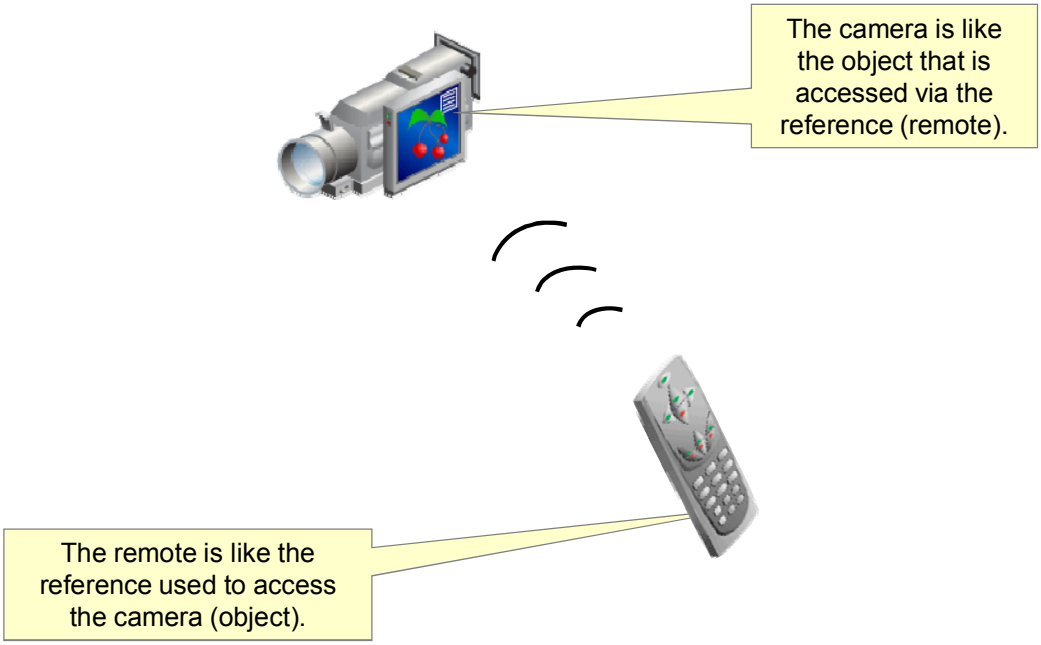
- Declaring, instantiating, and initializing objects
- Working with object references
- Using the String class
- Using the Java API documentation
- Using the StringBuilder class

Working with Objects: Introduction

Objects are accessed via references.

- Objects are instantiated versions of their class.
- Objects consist of attributes and operations:
  - In Java, these are fields and methods.

Accessing Objects by Using a Reference



To work with an object, you need to access it via a reference. A good analogy is using a remote control to operate an electronic device. The buttons on the remote control can be used to modify the operation of the device (in this case, a camera). For example, you can use the remote to have the camera stop, play, or record by interacting with the remote.

The Shirt Class

```
public class Shirt {
  public int shirtID = 0; // Default ID for the shirt
  public String description =
        "-description required-"; // default
  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  public char colorCode = 'U';
  public double price = 0.0; // Default price all items
  // This method displays the details for an item
  public void display() {
    System.out.println("Item ID: " + shirtID);
    System.out.println("Item description:" +
   description);
    System.out.println("Color Code: " + colorCode);
    System.out.println("Item price: " + price);
  } // end of display method
} // end of class
```

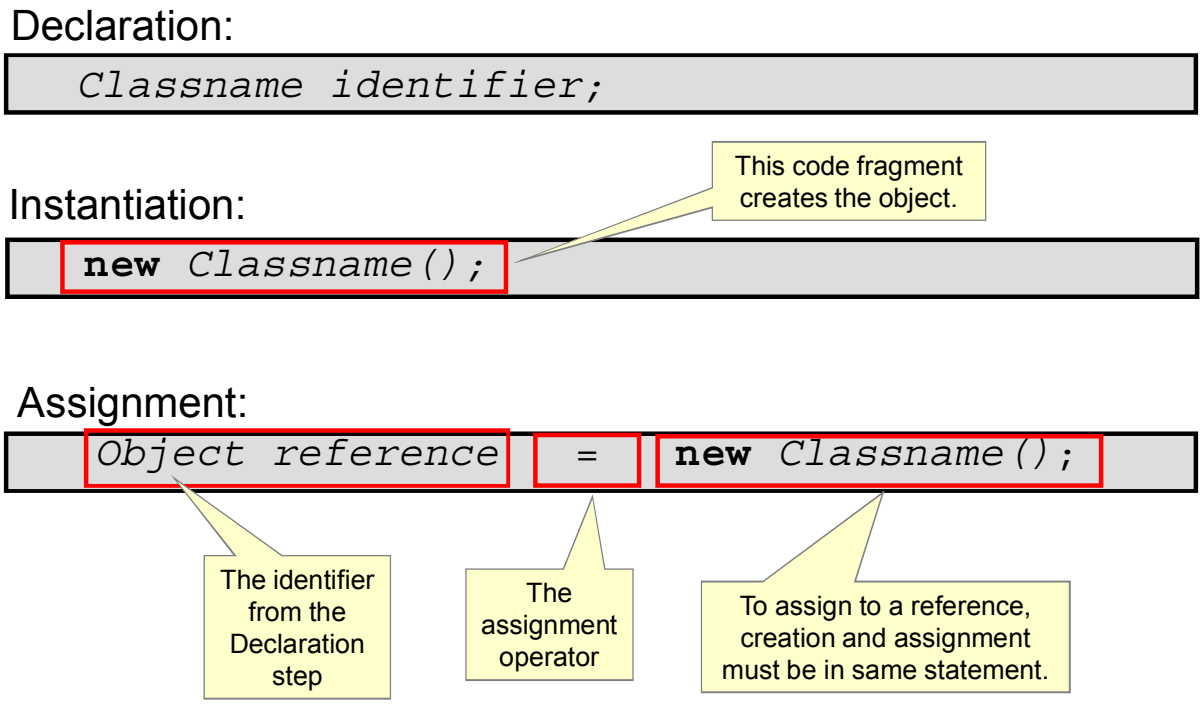
ORACLE

This topic covers accessing a simple object based on the `Shirt` class shown in the slide. It has four fields, `shirtID`, `description`, `colorCode`, and `price`, and one method, `display()`. Note that methods are usually written in this fashion, with the method name followed by a pair of parentheses to indicate that it is a method.

Topics

- Declaring, instantiating, and initializing objects
- Working with object references
- Using the String class
- Using the Java API documentation
- Using the StringBuilder class

Working with Object Reference Variables



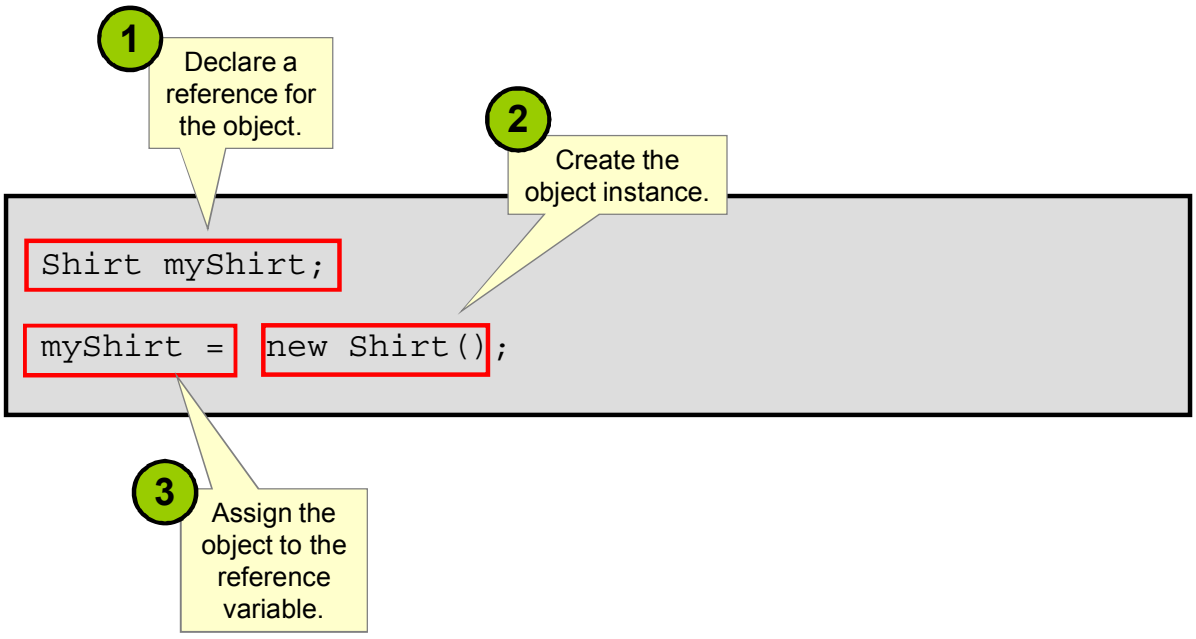
There are three steps to getting an object reference:

1. Declare the reference.
2. Instantiate the object.
3. Assign the object to the reference.

Note that, as indicated in the slide, the way that the assignment operator (an = symbol) works requires that the reference and the newly created object must be in the same statement. (Statements are ended with the semicolon symbol and are not the same as lines. The end of a line means nothing to the Java compiler; it only helps make the code more readable.)

The assignment operator for assigning objects to references is exactly the same as the assignment operator for assigning primitive values. Don't confuse it with the == (equality) symbol. You'll learn later what the equality symbol is used for in Java.

Declare and Initialize: Example



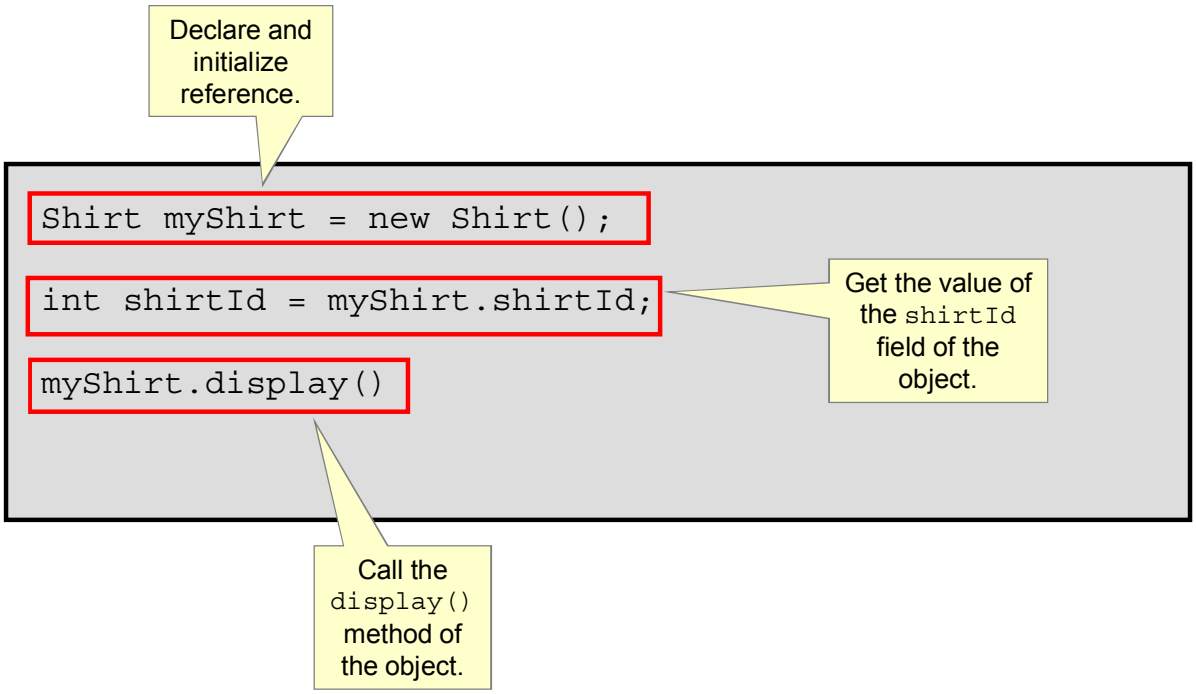
Declaring and initializing a reference variable is very similar to declaring and initializing a primitive type variable.

The main difference is that you must create an object instance (from a class) for the reference variable to point to before you can initialize the object instance.

To declare, instantiate, and initialize an object reference variable:

1. Declare a reference to the object by specifying its identifier and the type of object that the reference points to (the class of the object).
2. Create the object instance by using the `new` keyword.
3. Initialize the object reference variable by assigning the object to the object reference variable.

Work with Object References

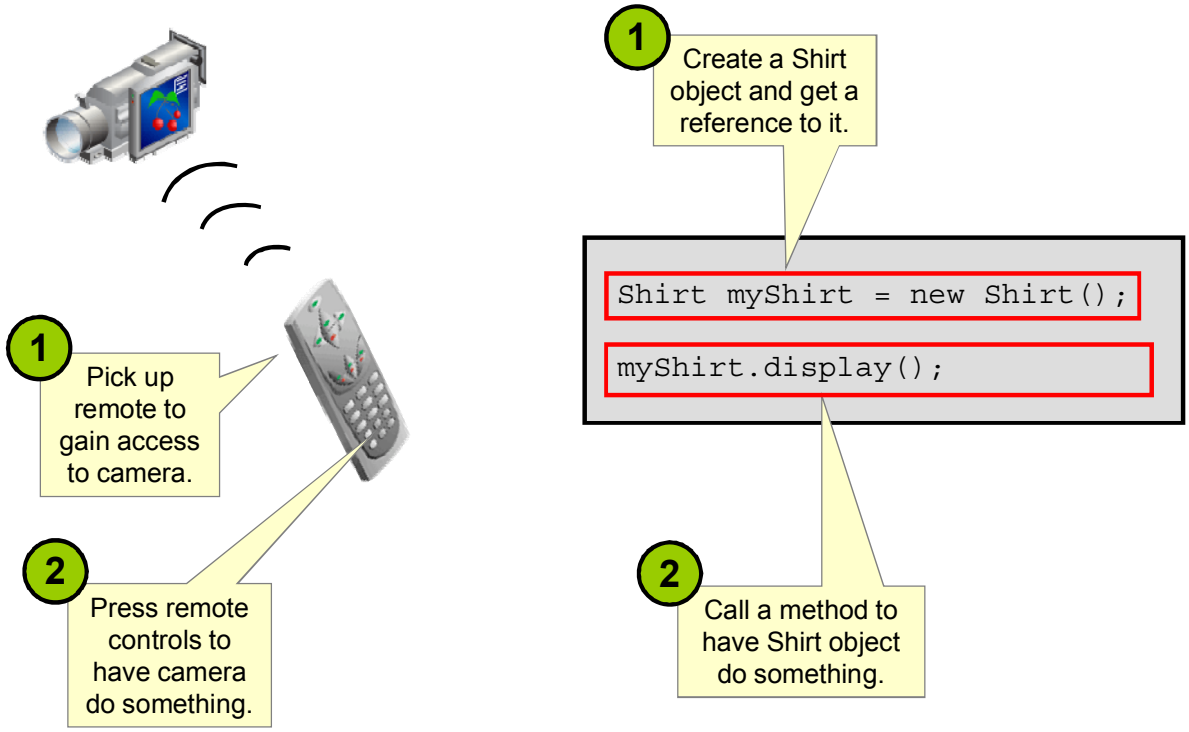


The diagram illustrates some important points. Notice how the first line declares and initializes the object reference in a single line (as opposed to two lines in the previous slide).

Also note the use of the dot (.) operator with an object reference to manipulate the values or to invoke the methods of a specific object. The example in the slide uses the dot notation to access a field of the object, in this case assigning it to a variable named `shirtId`.

The final line of code in the example shows the use of the dot notation to call a method on the object.

Working with Object References



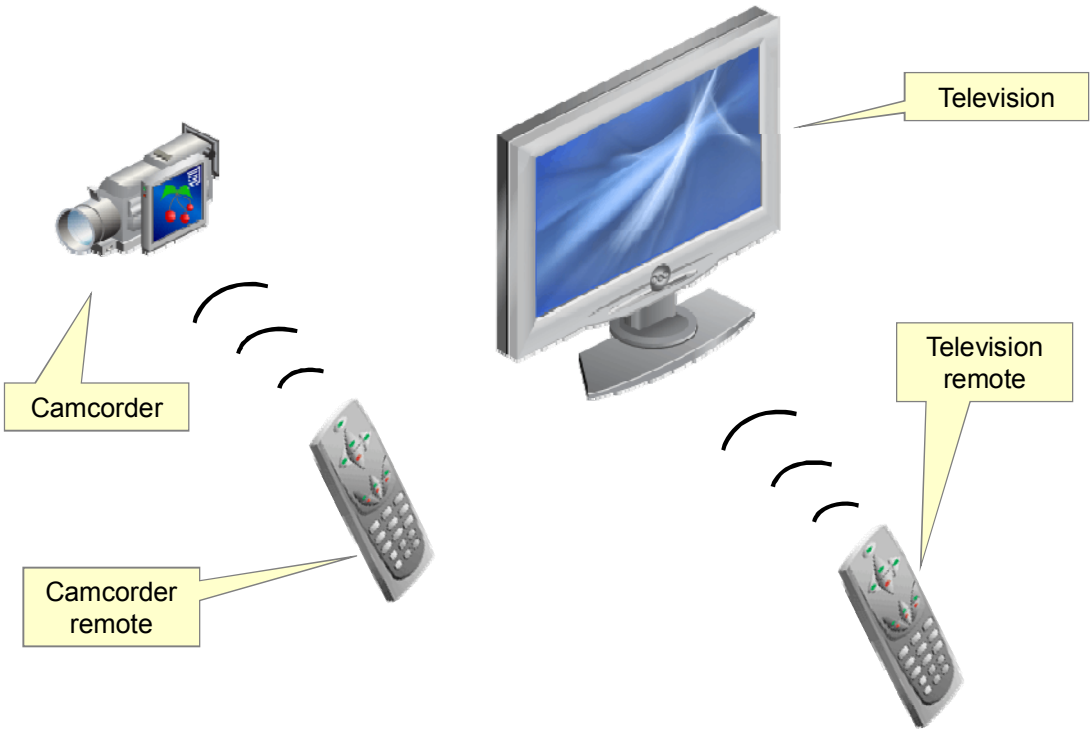
To return to the analogy of using a remote control to operate an electronic device, to operate an electronic device with a remote you need to:

1. Pick up the remote (and possibly turn it on).
2. Press a button on the remote to do something on the camera.

Similarly, to do something with a Java object you need to:

1. Get its “remote” (called a reference).
2. Press its “buttons” (called methods).

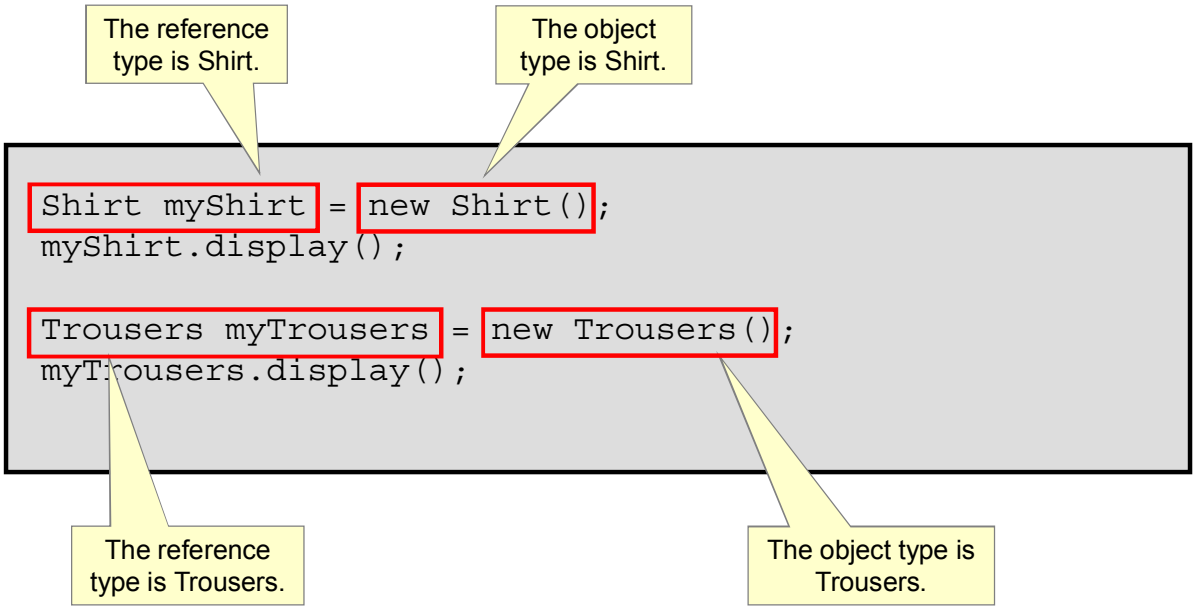
References to Different Objects



To extend the analogy just a little further, to work with a different object, say a flat-screen television, you need a remote for that object. In the Java world, you need a reference of the correct type for the object you are referencing.

You can ignore the fact that there is such a thing as a universal remote controller, although later in the course you'll discover that Java also has the concept of references that are not limited to a single object type! For the moment, let's just say that a reference of the same type as an object is one of the reference types that can be used, and is a good place to start exploring the world of Java objects.

Different Objects and Their References

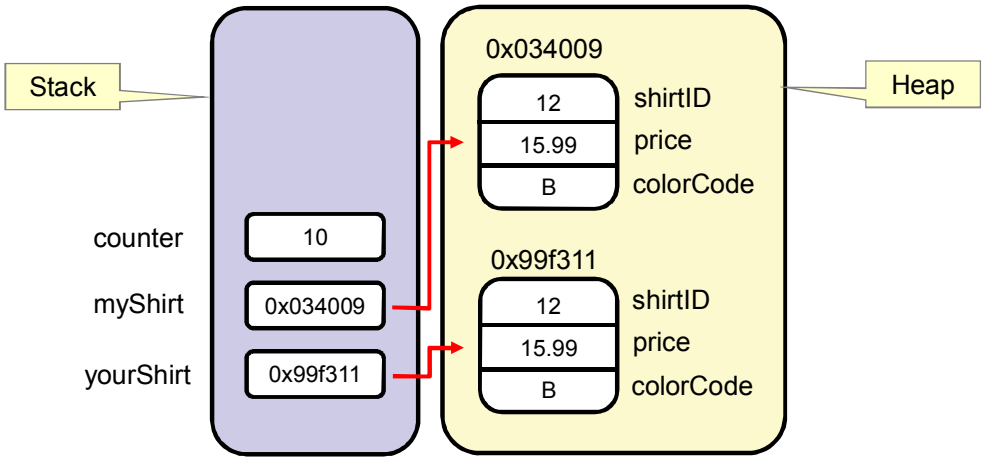


The example code in the slide shows objects being accessed with matching reference types. In the example, the Shirt reference type is used to refer to a Shirt object, and a Trousers reference type is used to refer to a Trousers object.

Later, you will see that the type of the reference doesn't have to be identical to the type of the object, but it must be compatible with it. This flexibility is a great strength of Java and you'll learn more about it in the lesson titled "Describing Advanced Object-Oriented Concepts."

References and Objects In Memory

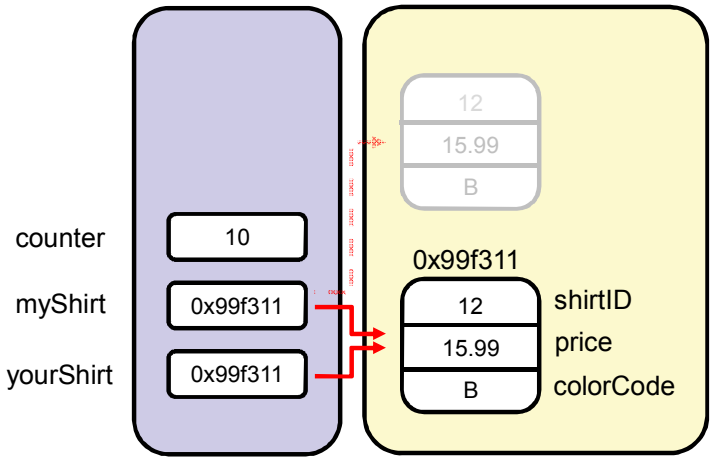
```
int counter = 10;
Shirt myShirt = new Shirt();
Shirt yourShirt = new Shirt();
```



This diagram shows how references point to a particular object in memory. Note that there are two objects in memory, although they are both of type Shirt. Also note that there are two Shirt references pointing to these two Shirt objects.

Assigning a Reference to Another Reference

```
myShirt = yourShirt;
```



The diagram shows what happens if the myShirt reference, after having its own object (in the previous slide), is now assigned the reference yourShirt. When this happens, the myShirt reference will drop its current object and be reassigned to the same object that yourShirt has. So, now two references, myShirt and yourShirt, point to the same object. Any changes to the object made by using one reference can be accessed using the other reference and vice versa.

Another effect of assigning the reference yourShirt to the reference myShirt is that if the previous object referred to by myShirt has no other references, it will now be inaccessible. In due course, it will be garbage collected, meaning that its memory will become available to store other objects.

Two References, One Object

Code fragment:

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

myShirt = yourShirt;

myShirt.colorCode = 'R';
yourShirt.colorCode = 'G';

System.out.println("Shirt color: " + myShirt.colorCode);
```

Output from code fragment:

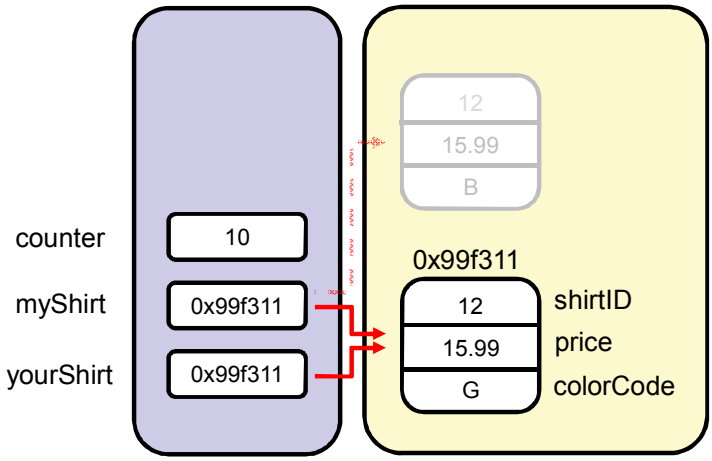
```
 Shirt color: G
```

ORACLE

This example now shows what happens if you use either reference to make a change or get a value from the object. References yourShirt and myShirt refer to the same object, so making a change or getting a field value by using one reference is exactly the same as doing it with the other reference.

Assigning a Reference to Another Reference

```
myShirt.colorCode = 'R';
yourShirt.colorCode = 'G';
```



Since myShirt and yourShirt now refer to the same object after the code in the slide completes, the `colorCode` field of the object will be `G`. And, of course, you will get the same result whether you use either of the following:

```
System.out.println(myShirt.colorCode);
System.out.println(yourShirt.colorCode);
```

Going back to the example of the television remote, it's the same as if you and a friend both have working remotes to the same television!

Quiz

Which of the following lines of code instantiates a Boat object and assigns it to a sailBoat object reference?

a. `Boat sailBoat = new Boat();`

b. `Boat sailBoat;`

c. `Boat = new Boat()`

d. `Boat sailBoat = Boat();`

Answer: a

Topics

- Declaring, instantiating, and initializing objects
- Working with object references
- Using the String class
- Using the Java API documentation
- Using the StringBuilder class

The `String` Class

The String class supports some non-standard syntax

- A String object can be instantiated without using the `new` keyword; this is preferred:
  ```
  String hisName = "Fred Smith";
  ```
  - The `new` keyword can be used, but it is *not* best practice:
    ```
    String herName = new String("Anne Smith");
    ```
- A String object is immutable; its value cannot be changed.
- A String object can be used with the string concatenation operator symbol (+) for concatenation.

ORACLE

- The String class is one of the many classes included in the Java class libraries. The String class provides you with the ability to store a sequence of characters. You will use the String class frequently throughout your programs. Therefore, it is important to understand some of the special characteristics of strings in the Java programming language.
- Creating a String object using the `new` keyword creates two String objects in memory, while creating a String object by using a string literal creates only one object; therefore, the latter practice is more memory-efficient. To avoid the unnecessary duplication of String objects in memory, create String objects without the `new` keyword.

Concatenating Strings

When you use a string literal in Java code, it is instantiated and becomes a String reference

- Concatenate strings:

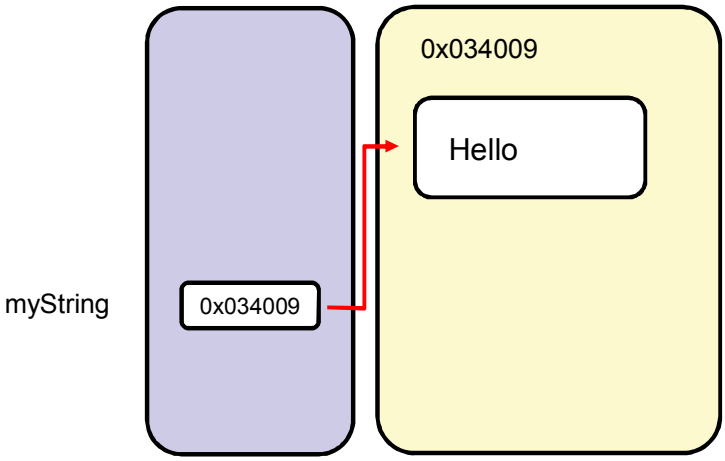
```
String name1 = "Fred"
theirNames = name1 + " and " +
             "Anne Smith";
```

- The concatenation creates a new string and the String reference theirNames now points to this new string.

String objects support the use of a special concatenation operator (+) for concatenating two or more strings. Because a literal string returns a String reference, string literals and String references can be intermixed in an expression that concatenates a number of strings, as shown in the slide.

String Concatenation

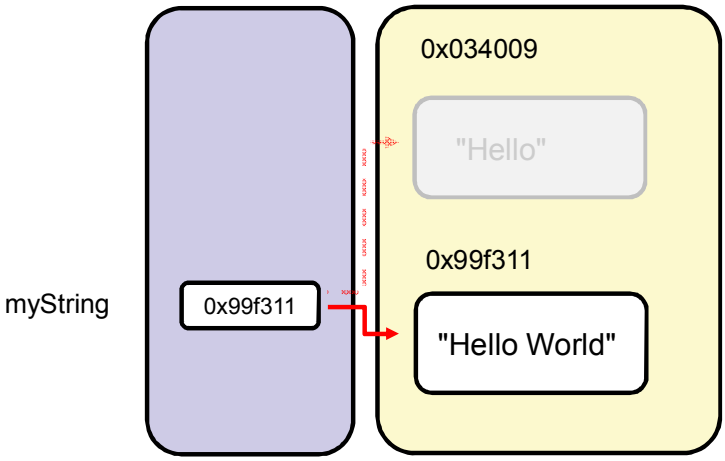
```
String myString = "Hello";
```



Because String is immutable, concatenating two strings requires creating a new string. The diagram shows a String object containing the string "`Hello`".

String Concatenation

```
String myString = "Hello";
myString = myString.concat(" World");
```

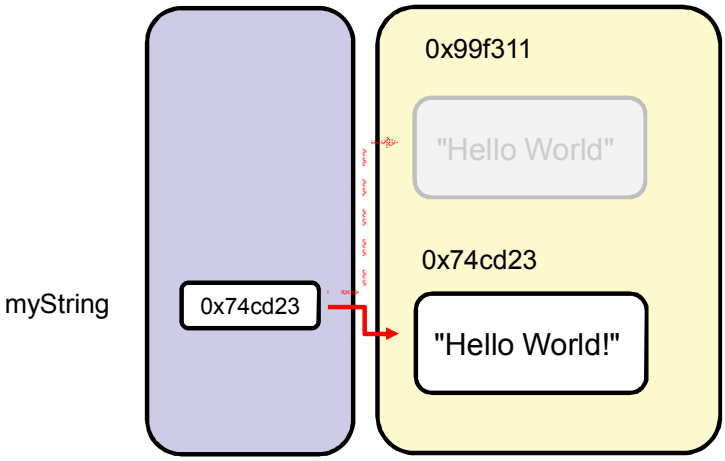


Here is the string "`World`" being concatenated to the original string. The `concat()` method is being used here, but whether you use that or the concatenation operator (`+`), a new String object will be created and a new String reference returned that points to this new object.

In the diagram, this is shown by the fact that the String reference myString is no longer 0x034009, and because that object is no longer referred to, it is now inaccessible and will be garbage collected.

String Concatenation

```
String myString = "Hello";
myString = myString.concat(" World");
myString = myString + "!"
```



Finally, on concatenating another string, this time using the concatenation operator, the same thing happens again. A new object is created and the reference for this object is assigned to myString.

String Method Calls with Primitive Return Values

A method call can return a single value of any type.

- An example of a method of primitive type `int`:

```
String hello = "Hello World";
int stringLength = hello.length();
```


Like most classes, the String class has a number of useful methods. Almost all of these methods do their useful work by returning a single value (Java allows only a single return from a method). The return type (essentially the type of the method) can be a primitive or a reference to an object.

To be able to use the return value in your code, you will typically use the assignment operator to assign the value (or reference) to a type that you have declared for this purpose.

The example in the slide shows the use of reference `hello` to call the method `length()`. Because the object this reference refers to is the string `Hello World`, this method call will return the value `11` and place it in the variable `stringLength`. `int` is the type of the method call `length()`.

String Method Calls with Object Return Values

Method calls returning objects:

```
String greet = " HOW ".trim();
String lc = greet + "DY".toLowerCase();
```

Or

```
String lc = (greet + "DY").toLowerCase();
```


This example shows several method calls that return object references.

First, the string object `"  HOW  "` is instantiated and has `trim` called on it. As a string literal returns an object reference, this is exactly the same as calling the method `trim()` on the reference. Notice that the string `"  HOW  "` has two spaces on either side of the word. The string returned will be just three characters long because these spaces will be removed. This new string will be referenced by "`greet`."

The next example shows a method call not being assigned to a type, but simply used in an expression. `toLowerCase()` is called on the String `"DY"`, returning `"dy"`. `lc` now references an object containing `"HOWdy"`.

Finally, note how an alternative version with parentheses ensures that the two strings are concatenated (creating a new string) before `toLowerCase()` is called. `lc` now references an object containing `"howdy"`.

Method Calls Requiring Arguments

Method calls may require passing one or more arguments:

- Pass a primitive

```
String theString = "Hello World";
String partString = theString.substring(6);
```

- Pass an object

```
boolean endWorld =
            "Hello World".endsWith("World");
```


Some method calls require an argument to be passed into the method.

For example the `substring()` method shown in the example needs an index (an `int`) to indicate where to split the string. It returns a new string that consists of the remaining part of the string starting at "W", so in this case it returns `"World"`. (Substring indexes from 0 and begins with the character at the specified index and extends to the end of this string. "W" is at index 6.)

The method `endsWith()` requires a String reference to be passed as an argument. It returns a `boolean` because it simply determines if the string ends with the sequence of characters passed in. In this case it does, so `true` will be returned.

Topics

- Declaring, instantiating, and initializing objects
- Working with object references
- Using the String class
- Using the Java API documentation
- Using the StringBuilder class

Java API Documentation

Consists of a set of webpages;

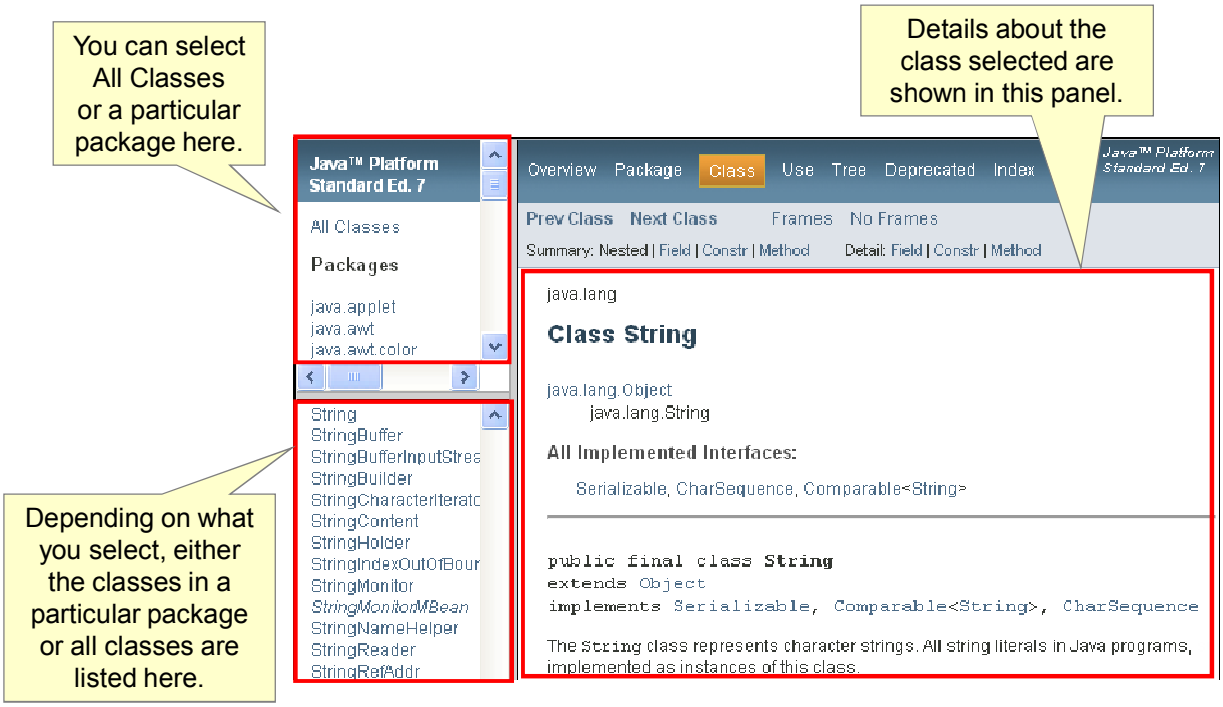
- Lists all the classes in the API
  - Descriptions of what the class does
  - List of constructors, methods, and fields for the class
- Highly hyperlinked to show the interconnections between classes and to facilitate lookup
- Available on the Oracle website at:
  http://download.oracle.com/javase/7/docs/api/index.html

ORACLE

All of the Java technology JDKs contain a series of prewritten classes for you to use in your programs. These Java technology class libraries are documented in the Java API documentation for the version of the JDK that you are using. The class library specification is a series of Hypertext Markup Language (HTML) webpages that you can load in your web browser.

A Java class library specification is a very detailed document outlining the classes in the API. Every API includes documentation describing the use of classes and their fields and methods. When you are looking for a way to perform a certain set of tasks, this documentation is the best source for information about the predeveloped classes in the Java class libraries.

Java Platform SE 7 Documentation

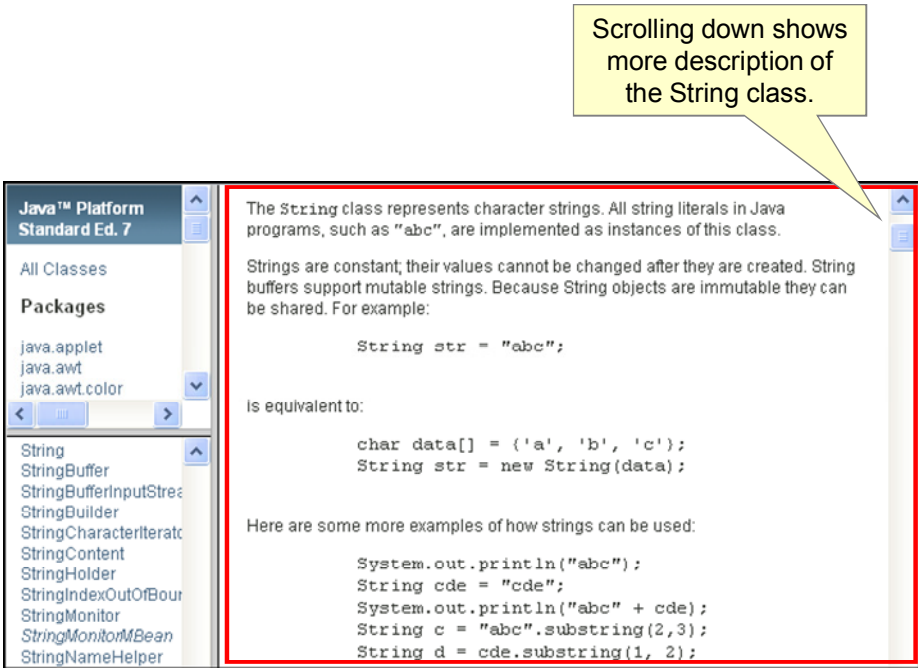


In the screenshot in the slide, you can see the three main panels of the webpage.

The top-right panel allows you to select a package. Java classes are organized into packages, but if you don't know the package of a particular class, you can select All Classes.

The bottom-left panel gives the list of classes in a package, or all classes if that has been selected. In this panel, the class String has been selected, populating the main panel on the right with the details of the class String.

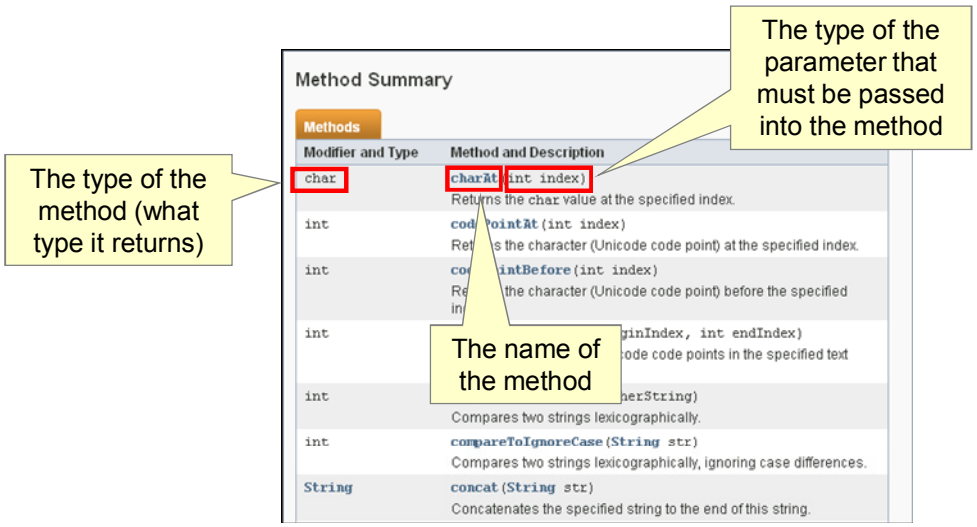
Java Platform SE 7 Documentation



ORACLE

The main panel on the right contains a lot of information about the class, so you need to scroll down to access the information you need.

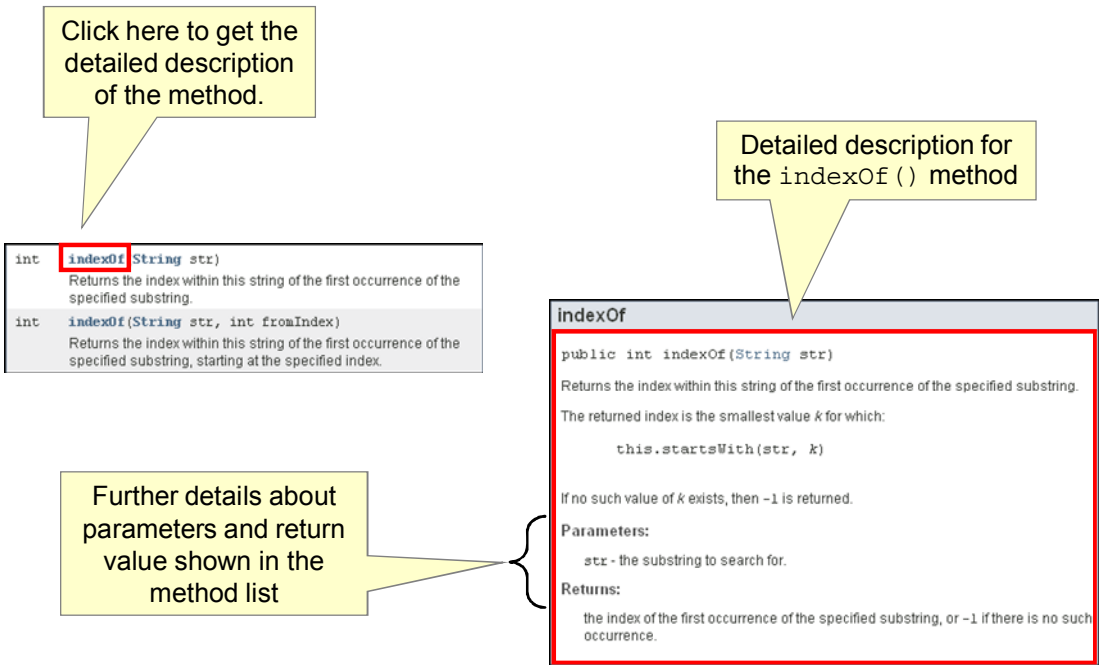
Java Platform SE 7: Method Summary



If you keep scrolling on the details for the String class, you will come to the list of methods (only a small subset of this list is shown here).

This master list of methods gives the basic details for the method. In this case, you can see that the type of the method is called `charAt()`, its type is `char`, and it requires an index (of type `int`) to be passed in. There's also a brief description that this method returns the `char` value at a particular index in the string.

Java Platform SE 7: Method Detail



For any of the methods, the method name and the parameter types are hyperlinked, so that you can get more details. The example here shows the detailed description for one of the `indexOf()` methods of String.

`System.out` Methods

To find details for `System.out.println()`, consider the following:

- System is a class (in `java.lang`).
- `out` is a field of System.
- `out` is a reference type that allows calling `println()` on the object type it references.

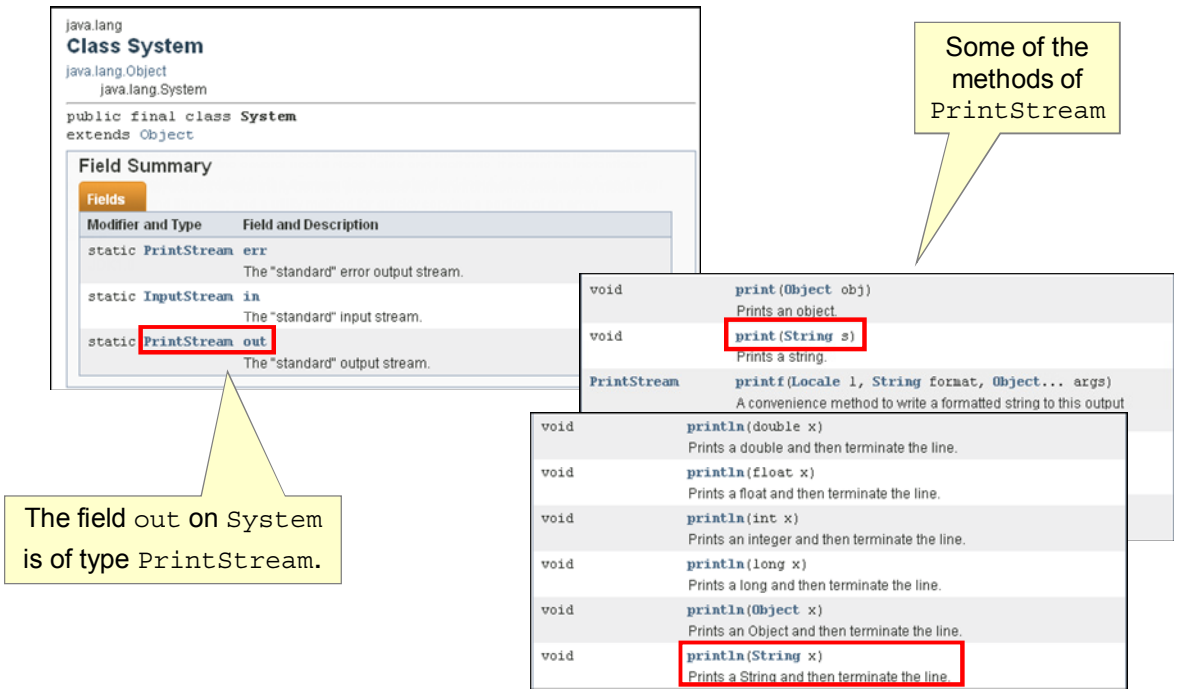
To find the documentation:

1. Go to System class and find the type of the `out` field.
2. Go to the documentation for that field.
3. Review the methods available.

ORACLE

The next few slides show how you might use the Java API documentation to find out more about `System.out.println()`. As you will see, this is a little unusual, because the class whose methods you need to investigate is not System. Rather, it is the class that is the type of the `out` field of the System object.

Documentation on `System.out.println()`



The diagram shows the Field Summary for the class `System`. Here, you can see that there is indeed a field called `out`, and it is of type `PrintStream`. By clicking `PrintStream`, you can now see the details for that class and, if you scroll down to the Method Summary, you will find (among many other methods) the `print()` method and the `println()` method.

Using `print()` and `println()` Methods

- The `println` method:
  `System.out.println(`*`data_to_print`*`);`
- Example:
  ```
  System.out.print("Carpe diem ");
  System.out.println("Seize the day");
  ```
- This method prints the following:
  ```
  Carpe diem Seize the day
  ```

ORACLE

Note that `System.out` is of type `PrintStream` but a `PrintStream` is not only for printing to the console. It is just that the default for this field is a reference to a `PrintStream` object that outputs to the console, but you could conceivably change the `out` reference if you wanted the output to go elsewhere.

The difference between the `print()` method and the `println()` method is that `print()` does not create a new line after printing the String, whereas `println()` does. Consequently, in the example in the slide, "`Seize the day`" occurs on the same line as "`Carpe diem`".

Topics

- Declaring, instantiating, and initializing objects
- Working with object references
- Using the String class
- Using the Java API documentation
- Using the StringBuilder class

StringBuilder Class

StringBuilder provides a mutable alternative to String.

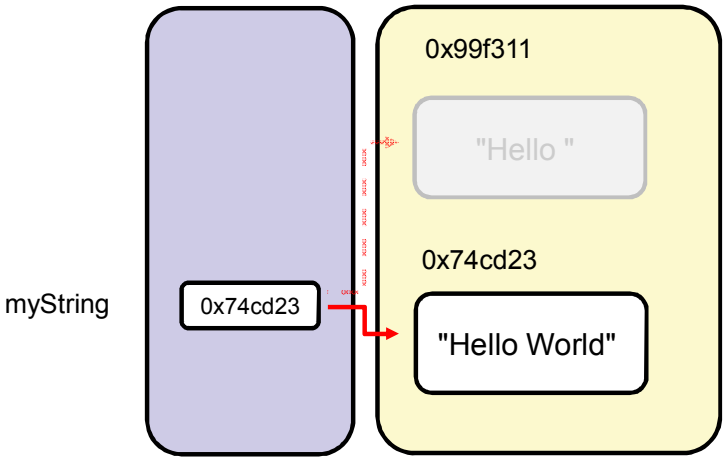
- Is a normal class; use `new` to instantiate.
- Has extensive set of methods for append, insert, delete
- Many methods return reference to current object; there is no instantiation cost.
- Can be created with an initial capacity best suited to need
- String is still needed because:
  - It may be safer to use an immutable object
  - A class in the API may require a string
  - It has many more methods not available on StringBuilder

ORACLE

Unlike String, there is no shortcut to instantiate a StringBuilder. It is simply instantiated like any other object by using the `new` keyword. StringBuilder is not a complete replacement for String, but it is more suitable if many modifications are likely to be made to the string represented by the data type.

StringBuilder Advantages Over String for Concatenation (or Appending)

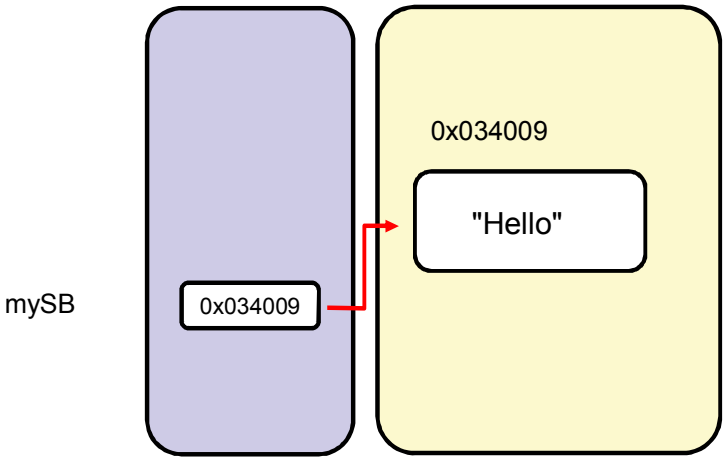
```
String myString = "Hello";
myString = myString.concat(" World);
```



This slide offers a reminder of what happens when the strings `"Hello"` and `" World"` are concatenated. A new String object is created, and the reference for that object is assigned to myString.

StringBuilder: Declare and Instantiate

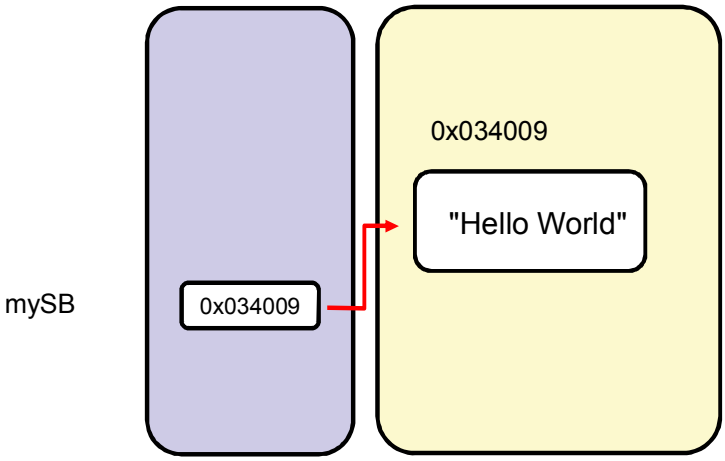
```
StringBuilder mySB = new StringBuilder("Hello");
```



This diagram shows the start of a sequence involving a StringBuilder. A new StringBuilder is instantiated, populated with the string `"Hello"`, and the reference for this new object is assigned to `mySB`.

StringBuilder Append

```
StringBuilder mySB = new StringBuilder("Hello");
mySB.append(" World");
```



To append the string `"World"`, all you need to do is call the `append()` method and pass in `"World"`. Note that no assignment is necessary because there already is a reference to the StringBuilder object, and this StringBuilder object now contains a representation of the combined strings `"Hello World"`.

Even if you did assign the return type of the `append()` method (which is StringBuilder), there would still be no object creation cost; the `append()` method modifies the current object and returns the reference to that object, the one already contained in `mySB`. (This may be useful to know if the entire method call is used as a type.)

Quiz

Which of the following statements are true? (Choose all that apply.)

a. The dot (.) operator creates a new object instance.

b. The String class provides you with the ability to store a sequence of characters.

c. The Java API specification contains documentation for all of the classes in a Java technology product.

d. String types are unique because they are the only class that allows you to build objects without using the `new` keyword.

Answer: b, c, d

Summary

Objects are accessed via references:

- Objects are instantiated versions of their class.
- Objects consist of attributes and operations:
  - In Java, these are fields and methods.
- To access the fields and methods of an object, get a reference variable to the object:
  - The same object may have more than one reference.
- An existing object's reference can be reassigned to a new reference variable.
- The `new` keyword instantiates a new object and returns a reference.

ORACLE

Practice 6-1 Overview: Creating and Manipulating Java Objects

This practice covers the following topics:

- Creating and initializing object instances
- Manipulating object references

ORACLE

In this practice, you create instances of a class and manipulate these instances in several ways.

Practice 6-2 Overview: Using the String and StringBuilder Classes

This practice covers the following topics:

- Working with String objects
- Working with StringBuilder objects

In this practice, you create and initialize String objects and print their contents. You also create and manipulate StringBuilder objects.

Practice 6-3 Overview: Using the Java API Documentation

This practice covers becoming familiar with the Java API documentation.

In this practice, you examine the Java API specification to become familiar with the documentation and with looking up classes and methods. You are not expected to understand everything you see. As you progress through this course, the Java API documentation should make more sense.

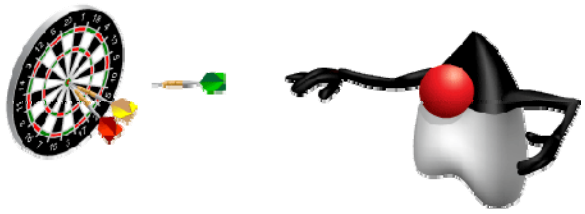
7

Using Operators and Decision Constructs

Objectives

After completing this lesson, you should be able to:

- Use a relational operator
- Test equality between strings
- Use a conditional operator
- Create `if` and `if/else` constructs
- Nest an `if` statement
- Chain an `if/else` statement
- Use a `switch` statement



ORACLE

Relevance

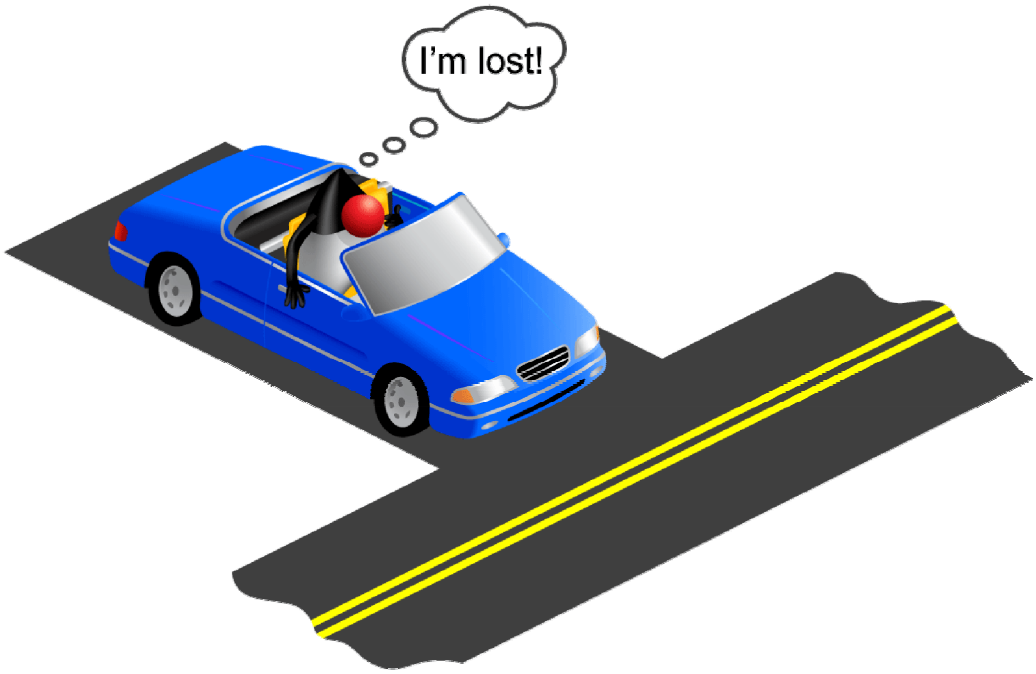
- When you have to make a decision that has several different paths, how do you ultimately choose one path over all the other paths?
- For example, what are all of the things that go through your mind when you are going to purchase an item?

We are trying to get you to recognize that you have to make a lot of decisions and that you often use the word “if” with some condition when making decisions. For example, “If the house is blue, I will take a tour of it.” Or, “If the car is a sports car and it is safe, I will buy it.” These types of decisions go through our minds subconsciously every day.

Topics

- Use relational and conditional operators
- Create `if` and `if/else` constructs
- Chain an `if/else` statement
- Use a `switch` statement

Using Relational and Conditional Operators



One of the tasks that programs often perform is to evaluate a condition and, depending on the result, execute different blocks or branches of code. For example, your program might check to see if the value of one variable is equal to the value of another variable and, if so, do something. The image illustrates the type of decision people make every day. In addition to arithmetic operators, such as plus (`+`) and increment (`++`), the Java programming language provides several relational operators, including `<` and `>` for less than and greater than, respectively, and `&&` for `AND`. These operators are used when you want your program to execute different blocks or branches of code depending on different conditions, such as checking if the value of two variables is the same.

Note: Each of these operators is used within the context of a decision construct, such as an `if` or `if/else` construct, which will be presented later.

Elevator Example

```
public class Elevator {

  public boolean doorOpen=false; // Doors are closed by default
  public int currentFloor = 1; // All elevators start on first floor
  public final int TOP_FLOOR = 10;
  public final int MIN_FLOORS = 1;

  public void openDoor() {
    System.out.println("Opening door.");
    doorOpen = true;
    System.out.println("Door is open.");
  }

  public void closeDoor() {
    System.out.println("Closing door.");
    doorOpen = false;
    System.out.println("Door is closed.");
  }
...
```

Open door

Close door

An elevator has many functions. We will begin with an elevator that has only the following functionality. (This functionality will be enhanced as you look at further examples in subsequent lessons.) The functions of the elevator in this lesson are:

- The doors of the elevator can open.
- The doors of the elevator can close.
- The elevator can go up one floor.
- The elevator can go down one floor.

You will see several variations of the `Elevator` class in this and subsequent lessons, including several variations illustrating the use of decision constructs. The complete Elevator Example code for this lesson is shown as follows.

```
public class Elevator {

        public boolean doorOpen=false; // Default setting
        public int currentFloor = 1; // Default starting point
        public final int TOP_FLOOR = 10;
        public final int MIN_FLOORS = 1;

        public void openDoor() {
          System.out.println("Opening door.");
          doorOpen = true;
          System.out.println("Door is open.");
        }

        public void closeDoor() {
          System.out.println("Closing door.");
          doorOpen = false;
          System.out.println("Door is closed.");
        }
        public void goUp() {
          System.out.println("Going up one floor.");
          currentFloor++;
          System.out.println("Floor: " + currentFloor);
        }

        public void goDown() {
          System.out.println("Going down one floor.");
          currentFloor--;
          System.out.println("Floor: " + currentFloor);
        }
}
```

The `ElevatorTest.java` File

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

    Elevator myElevator = new Elevator();

     myElevator.openDoor();
     myElevator.closeDoor();
     myElevator.goDown();
     myElevator.goUp();
     myElevator.goUp();
     myElevator.goUp();
     myElevator.openDoor();
     myElevator.closeDoor();
     myElevator.goDown();
     myElevator.openDoor();
     myElevator.goDown();
     myElevator.openDoor();
   }
}
```

ORACLE

A test class, similar to the example test class, runs the elevator through some tests.

Relational Operators

Condition	Operator	Example
Is equal to	==	int i=1; (i == 1)
Is not equal to	!=	int i=2; (i != 1)
Is less than	<	int i=0; (i < 1)
Is less than or equal to	<=	int i=1; (i <= 1)
Is greater than	>	int i=2; (i > 1)
Is greater than or equal to	>=	int i=1; (i >= 1)

ORACLE

Relational operators compare two values to determine their relationship. The table lists the different conditions you can test by using relational operators. The result of all relational operators is a Boolean value. Boolean values can be either true or false. For example, all of the examples in the table yield a Boolean result of true.

Note: The equal sign (=) is used to make an assignment.

Testing Equality Between Strings

Example:

```
public class Employees {

  public String name1 = "Fred Smith";
  public String name2 = "Joseph Smith";

  public void areNamesEqual() {

      if (name1.equals(name2)) {
        System.out.println("Same name.");
      }
      else {
        System.out.println("Different name.");
      }
    }
  }
```

If you use the == operator to compare object references to String objects, the operator tests whether the address in the String object references in memory are equal, not their contents.

Discussion: Are all of the following String object references equal?

```
String helloString1 = ("hello");
String helloString2 = "hello";
String helloString3 = new String("hello");
```

If you want to test equality between the strings of characters, such as whether the name “Fred Smith” is equal to “Joseph Smith,” use the `equals` method of the String class. The class in the example contains two employee names and a method that compares names.

Common Conditional Operators

Operation	Operator	Example

If one condition AND another condition	`&&`	`int i = 2;`
		`int j = 8;`
		`((i < 1) && (j > 6))`

If either one condition OR another condition	`||`	`int i = 2;`
		`int j = 8;`
		`((i < 1) || (j > 10))`

NOT	`!`	`int i = 2;` `(!(i < 3))`

ORACLE

You will also need to be able to make a single decision based on more than one condition. Under such circumstances, you can use conditional operators to evaluate complex conditions as a whole. The table in the slide lists the common conditional operators in the Java programming language. For example, all of the examples in the table yield a Boolean result of false.

Discussion: What relational and conditional operators are expressed in the following paragraph? If the toy is red, I will purchase it. However, if the toy is yellow and costs less than a red item, I will also purchase it. If the toy is yellow and costs the same as or more than another red item, I will not purchase it. Finally, if the toy is green, I will not purchase it.

Topics

- Use relational and conditional operators
- Create `if` and `if/else` constructs
- Chain an `if/else` statement
- Use a `switch` statement

Creating `if` and `if/else` Constructs

An `if` statement, or an `if` construct, executes a block of code if an expression is *true.*

The `if` Construct

- Syntax:

```
if (boolean_expression) {
 code_block;
} // end of if construct
// program continues here
```

- Example of potential output:

```
Opening door.
Door is open.
Closing door.
Door is closed.
Going down one floor.
Floor: 0 <-- this is an error in logic
Going up one floor.
Floor: 1
Going up one floor.
Floor: 2
...
```

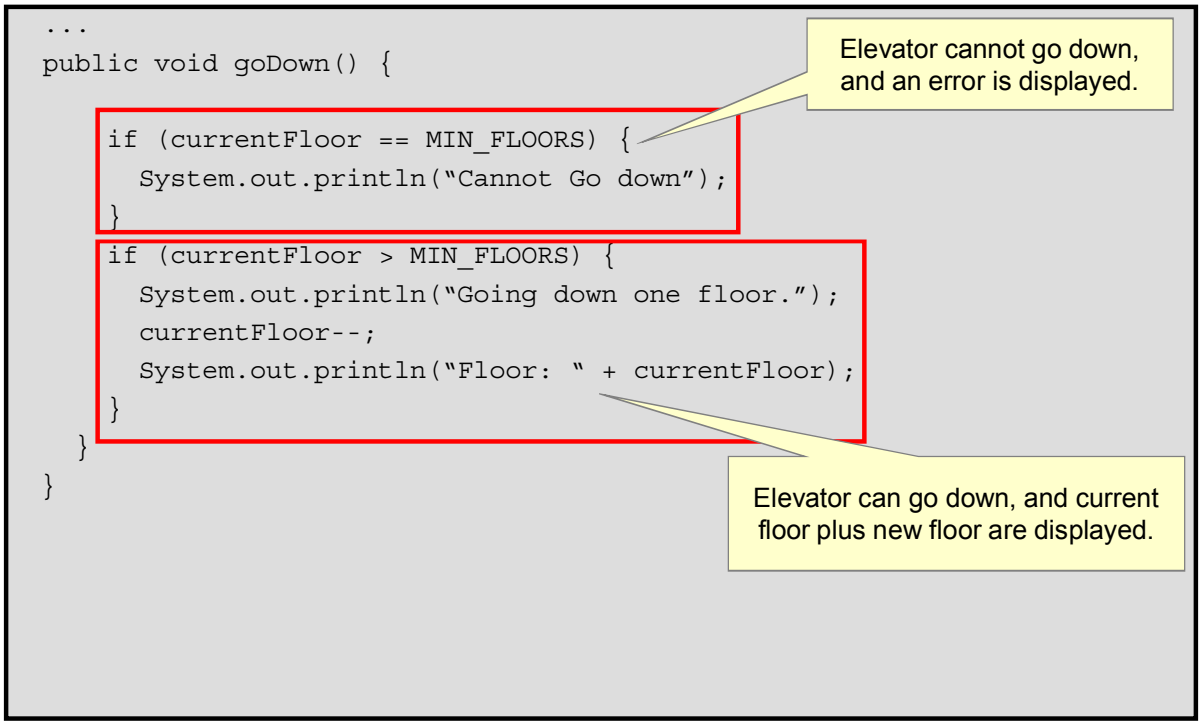

An `if` statement, or an `if` construct, executes a block of code if an expression is true. There are a few variations on the basic `if` construct. However, the simplest is:

```
if (boolean_expression) {
      <code_block>
} // end of if construct
// program continues here
```

where:

- The `boolean_expression` is a combination of relational operators, conditional operators, and values resulting in a value of true or false
- The `code_block` represents the lines of code that are executed if the expression is true
- First, the `boolean_expression` is tested. If the expression is true, then the code block is executed. If the `boolean_expression` is not true, the program skips to the brace marking the end of the `if` construct code block.

The `if` Construct: Example



The `ElevatorTest` class tests an Elevator object by invoking its methods. One of the first methods that the `ElevatorTest` class invokes is the `goDown` method. Two `if` statements can fix this problem. The following `goDown` method contains two `if` constructs that determine whether the elevator should go down or display an error. The entire `ElevatorTest` class is as follows:

```
public class IfElevator {

        public boolean doorOpen=false; // Default setting
        public int currentFloor = 1; // Default starting point
        public final int TOP_FLOOR = 10;
        public final int MIN_FLOORS = 1;

        public void openDoor() {
            System.out.println("Opening door.");
            doorOpen = true;
            System.out.println("Door is open.");
          }
        public void closeDoor() {
            System.out.println("Closing door.");
            doorOpen = false;
            System.out.println("Door is closed.");
            }
        public void goUp() {
          System.out.println("Going up one floor.");
          currentFloor++;
          System.out.println("Floor: " + currentFloor);
        }
        public void goDown() {

          if (currentFloor == MIN_FLOORS) {
            System.out.println("Cannot Go down");
          }
          if (currentFloor > MIN_FLOORS) {
            System.out.println("Going down one floor.");
            currentFloor--;
            System.out.println("Floor: " + currentFloor);
          }
        }
  }
```

The `if` Construct: Output

Example potential output:

```
Opening door.
Door is open.
Closing door.
Door is closed.
Cannot Go down ← elevator logic prevents problem
Going up one floor.
Floor: 2
Going up one floor.
Floor: 3
...
```

Nested `if` Statements

```
...
public void goDown() {

    if (currentFloor == MIN_FLOORS) {
      System.out.println("Cannot Go down");
    }

    if (currentFloor > MIN_FLOORS) {

       if (!doorOpen) {
            System.out.println("Going down one floor.");
            currentFloor--;
            System.out.println("Floor: " + currentFloor);
       }
    }
  }

}
```

Nested `if` statement

Sometimes you might need to execute an `if` statement as part of another `if` statement. The code example illustrates how to use nested `if` statements to check the values of two variables. If the value of the `currentFloor` variable is equal to the `MIN_FLOORS` constant, an error message is printed and the elevator does not go down. If the value of the `currentFloor` variable is greater than the `MIN_FLOORS` constant and the doors are closed, the elevator goes down. The entire `NestedIfElevator` example code is listed as follows.

Note: Use nested `if/else` constructs sparingly because they can be confusing to debug.

```
public class NestedIfElevator {

        public boolean doorOpen=false; // Doors are closed by default
        public int currentFloor = 1; // All elevators start on first floor
        public final int TOP_FLOOR = 10;
        public final int MIN_FLOORS = 1;

        public void openDoor() {
            System.out.println("Opening door.");
            doorOpen = true;
          System.out.println("Door is open.");
        }

      public void closeDoor() {
          System.out.println("Closing door.");
          doorOpen = false;
          System.out.println("Door is closed.");
    }

        public void goUp() {
          System.out.println("Going up one floor.");
          currentFloor++;
          System.out.println("Floor: " + currentFloor);
        }

    public void goDown() {
          if (currentFloor == MIN_FLOORS) {
             System.out.println("Cannot Go down");
          }
          if (currentFloor > MIN_FLOORS) {

             if (!doorOpen) {
               System.out.println("Going down one floor.");
               currentFloor--;
              System.out.println("Floor: " + currentFloor);
            }
          }
   }

}
```

The `if/else` Construct

Syntax:

```
if (boolean_expression) {

    <code_block1>

} // end of if construct

else {

    <code_block2>

} // end of else construct

// program continues here
```


Often, you want one block of code to be executed if the expression is true, and another block of code to be executed if the expression is false. You can use an `if` construct to execute a code block if the expression is true with an `else` construct that executes only if the expression is false. The syntax for an `if/else` construct is shown in the example in the slide, where:

- The `boolean_expression` is a combination of relational operators, conditional operators, and values resulting in a value of true or false
- The `code_block1` represents the lines of code that are executed if the expression is true, and `code_block2` represents the lines of code that are executed if the expression is false.

The `if/else` Construct: Example

```
public void goUp() {
   System.out.println("Going up one floor.");
   currentFloor++;
   System.out.println("Floor: " + currentFloor);
 }

 public void goDown() {

   if (currentFloor == MIN_FLOORS) {
     System.out.println("Cannot Go down");
   }
   else {
     System.out.println("Going down one floor.");
     currentFloor--;
     System.out.println("Floor: " + currentFloor);}
   }
 }
}
```

Executed if expression is true

Executed if expression is false

ORACLE

You can use one `if/else` statement to fix the problem of the elevator going to an invalid floor. The `goDown` method shown in the slide example contains one `if/else` construct that determines whether the elevator should go down or display an error. If the value of the `currentFloor` variable is equal to the `MIN_FLOORS` constant, an error message is printed and the elevator does not go down. Otherwise (`else`), the value of the `currentFloor` variable is assumed to be greater than the `MIN_FLOORS` constant and the elevator goes down. The complete code example is as follows:

```
public class IfElseElevator {
      public boolean doorOpen=false; // Default setting
      public int currentFloor = 1; // Default setting
      public final int TOP_FLOOR = 10;
      public final int MIN_FLOORS = 1;

      public void openDoor() {
        System.out.println("Opening door.");
        doorOpen = true;
        System.out.println("Door is open.");
      }
      public void closeDoor() {
        System.out.println("Closing door.");
        doorOpen = false;
              System.out.println("Door is closed.");
      }

      public void goUp() {
        System.out.println("Going up one floor.");
        currentFloor++;
        System.out.println("Floor: " + currentFloor);
      }

      public void goDown() {

        if (currentFloor == MIN_FLOORS) {
          System.out.println("Cannot Go down");
        }
        else {
          System.out.println("Going down one floor.");
          currentFloor--;
          System.out.println("Floor: " + currentFloor);}
         }
        }
    }
```

The `if/else` Construct

Example potential output:

```
Opening door.
Door is open.
Closing door.
Door is closed.
Cannot Go down ← elevator logic prevents problem
Going up one floor.
Floor: 2
Going up one floor.
Floor: 3
...
```

Topics

- Use relational and conditional operators
- Create `if` and `if/else` constructs
- Chain an `if/else` statement
- Use a `switch` statement

Chaining `if/else` Constructs

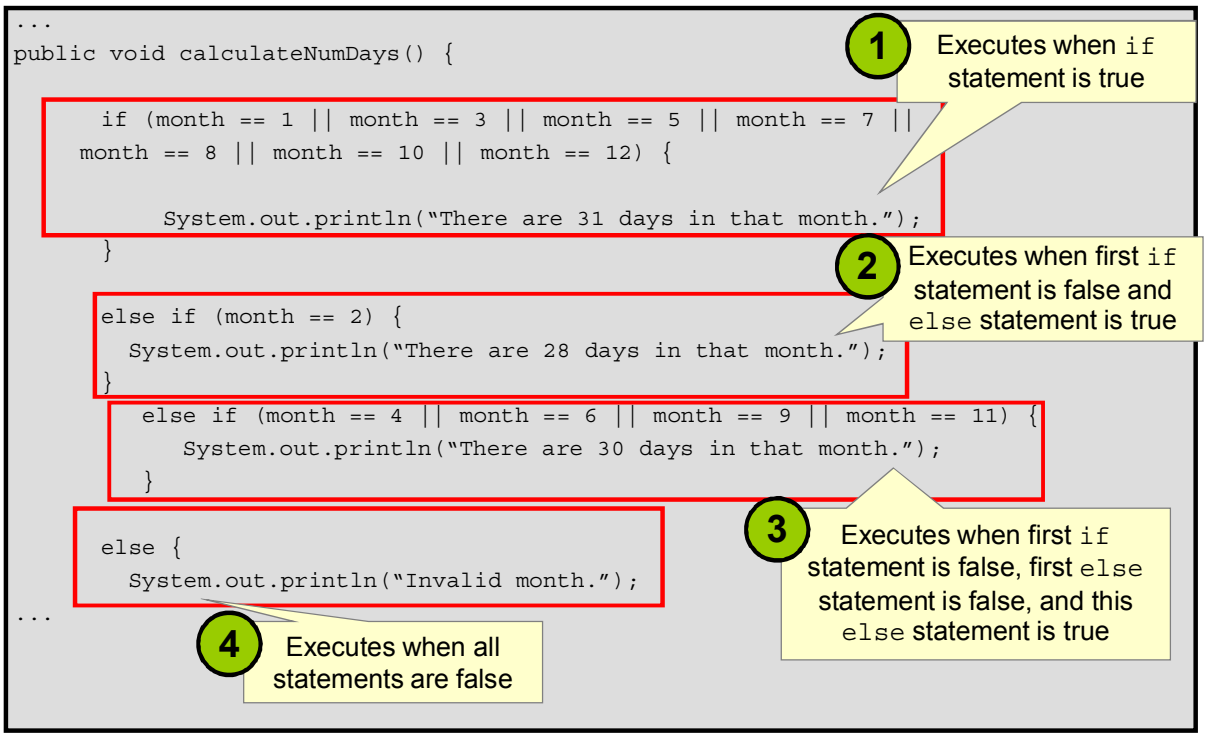
Syntax:

```
if (boolean_expression) {

    <code_block1>

} // end of if construct

else if (boolean_expression){

    <code_block2>

} // end of else if construct

else {

    <code_block3>
}
// program continues here
```


You can chain `if` and `else` constructs together to state multiple outcomes for several different expressions. The syntax for a chained `if/else` construct is shown in the slide example, where:

- The `boolean_expression` is a combination of relational operators, conditional operators, and values resulting in a value of true or false
- The `code_block1` represents the lines of code that are executed if the expression is true. The `code_block2` represents the lines of code that are executed if the expression is false and the condition in the second `if` is true. The `code_block3` represents the lines of code that are executed if the expression in the second `if` also evaluates to false.

Chaining `if/else` Constructs



The example is an `IfElseDate` class containing several chained `if/else` constructs that determine how many days there are in a month. The `calculateNumDays` method chains three `if/else` statements together to determine the number of days in a month. Although this code is syntactically correct, chaining `if/else` statements can result in confusing code and should be avoided.

Topics

- Use relational and conditional operators
- Create `if` and `if/else` constructs
- Chain an `if/else` statement
- Use a `switch` statement

Using the `switch` Construct

Syntax:

```
switch (variable) {
    case literal_value:
       <code_block>
       [break;]
    case another_literal_value:
        <code_block>
       [break;]
    [default:]
       <code_block>
}
```


Another keyword used in decision making is the `switch` keyword. The `switch` construct helps you avoid confusing code because it simplifies the organization of the various branches of code that can be executed.

The `IfElseDate` class example could be rewritten by using a `switch` construct. The syntax for the `switch` construct is shown in the slide, where:

- The `switch` keyword indicates a `switch` statement
- The `variable` is the variable whose value you want to test. The `variable` can be only of type `char`, `byte`, `short`, or `int`.
- The `case` keyword indicates a value that you are testing. The combination of the `case` keyword and a `literal_value` is referred to as a case label.

- The `literal_value` is any valid value that a variable might contain. You can have a case label for each value that you want to test. Literal values cannot be variables, expressions, String, or method calls. Literal values can be constants (final variables like `MAX_NUMBER` defined somewhere else), literals (like '`A`' or `10`), or both.
- The `[break;]` statement is an optional keyword that causes the flow of code to immediately exit the `switch` statement. Without a `break` statement, all `code_block` statements following the accepted case statement are executed (until a `break` statement or the end of the `switch` construct is reached).

Using the `switch` Construct: Example

```
public class SwitchDate {

  public int month = 10;

  public void calculateNumDays() {

    switch(month) {
    case 1:
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12:
        System.out.println("There are 31 days in that month.");
        break;
...
```

The example contains a `SwitchDate` class that uses a `switch` construct to determine how many days there are in a month.

The `calculateNumDays` method in the `SwitchDate` class uses a `switch` statement to branch on the value of the month variable. If the month variable is equal to `1`, `3`, `5`, `7`, `8`, `10`, or `12`, the code jumps to the appropriate case label and then drops down to execute `System.out.println("There are 31 days in that month.")`.

```
public class SwitchDate {

        public int month = 10;

        public void calculateNumDays() {

          switch(month) {
          case 1:
          case 3:
          case 5:
          case 7:
          case 8:
          case 10:
          case 12:
            System.out.println("There are 31 days in that month.");
            break;
          case 2:
            System.out.println("There are 28 days in that month.");
            break;
          case 4:
          case 6:
          case 9:
          case 11:
            System.out.println("There are 30 days in that month.");
            break;
          default:
            System.out.println("Invalid month.");
            break;
          }
        }
  }
```

When To Use `switch` Constructs

- Equality tests
- Tests against a *single* value, such as `customerStatus`
- Tests against the value of an `int`, `short`, `byte`, or `char` type and `Strings`
- Tests against a fixed value known at compile time

Quiz

What is the purpose of the `else` block in an `if/else` statement?

a. To contain the remainder of the code for a method
b. To contain code that is executed when the expression in an `if` statement is false
c. To test if an expression is false

Answer: b

Quiz

Which of the following sentences is suitable for testing a value in a `switch` construct?

a. The `switch` construct tests whether values are greater than or less than a single value.
b. The `switch` construct tests against a single variable.
c. The `switch` construct tests the value of a `float`, `double`, or `boolean` data type and String.

Answer: b

- Answer a is incorrect because a relational operator should be used to test whether values are greater than or less than a single value.
- Answer b is correct.
- Answer c is incorrect. The `switch` construct tests the value of types `char`, `byte`, `short`, or `int`.

Summary

In this lesson, you should have learned how to:

- Use a relational operator
- Test equality between strings
- Use a conditional operator
- Create `if` and `if/else` constructs
- Nest an `if` statement
- Chain an `if/else` statement
- Use a `switch` statement

Practice 7.1 Overview: Writing a Class That Uses the `if/else` Statement

In this practice, you create classes that use `if` and `if/else` constructs. There are two sections in this practice:

- In the first section, you create the `DateTwo` class that uses `if/else` statements to display the day of the week based on the value of a variable.
- In the second section, you create the `Clock` class that uses `if/else` statements to display the part of the day, depending on the time of day.

ORACLE

Practice 7.2 Overview: Writing a Class That Uses the `switch` Statement

In this practice, you create a class called `Month` that uses `switch` statements to display the name of the month based upon the numeric value of a field.