Java SE 7 Programming

Student Guide - Volume I

D67238GC20
Edition 2.0
June 2012
D74996

ORACLE

Authors

Michael Williams

Tom McGinn

Matt Heimer

Technical Contributors and Reviewers

Peter Hall

Marnie Knue

Lee Klement

Steve Watts

Brian Earl

Vasily Strelnikov

Andy Smith

Nancy K.A.N

Chris Lamb

Todd Lowry

Ionut Radu

Joe Darcy

Brian Goetz

Alan Bateman

David Holmes

Editors

Richard Wallis

Daniel Milne

Vijayalakshmi Narasimhan

Graphic Designer

James Hans

Publishers

Syed Imtiaz Ali

Sumesh Koshy

Contents

1

Introduction

Course Goals

- This course covers the core APIs that you use to design object-oriented applications with Java. This course also covers writing database programs with JDBC.
- Use this course to further develop your skills with the Java language and prepare for the Oracle Certified Professional, Java SE 7 Programmer Exam.

Course Objectives

After completing this course, you should be able to do the following:

- Create Java technology applications that leverage the object-oriented features of the Java language, such as encapsulation, inheritance, and polymorphism
- Execute a Java application from the command line
- Create applications that use the Collections framework
- Implement error-handling techniques using exception handling
- Implement input/output (I/O) functionality to read from and write to data and text files and understand advanced I/O streams

Course Objectives

(continued)

- Manipulate files, directories, and file systems using the JDK7 NIO.2 specification
- Perform multiple operations on database tables, including creating, reading, updating, and deleting, using the JDBC API
- Process strings using a variety of regular expressions
- Create high-performing multi-threaded applications that avoid deadlock
- Localize Java applications

Audience

The target audience includes those who have:

- Completed the *Java SE 7 Fundamentals* course or have experience with the Java language, and can create, compile, and execute programs
- Experience with at least one programming language
- An understanding of object-oriented principles
- Experience with basic database concepts and a basic knowledge of SQL

Prerequisites

To successfully complete this course, you must know how to:

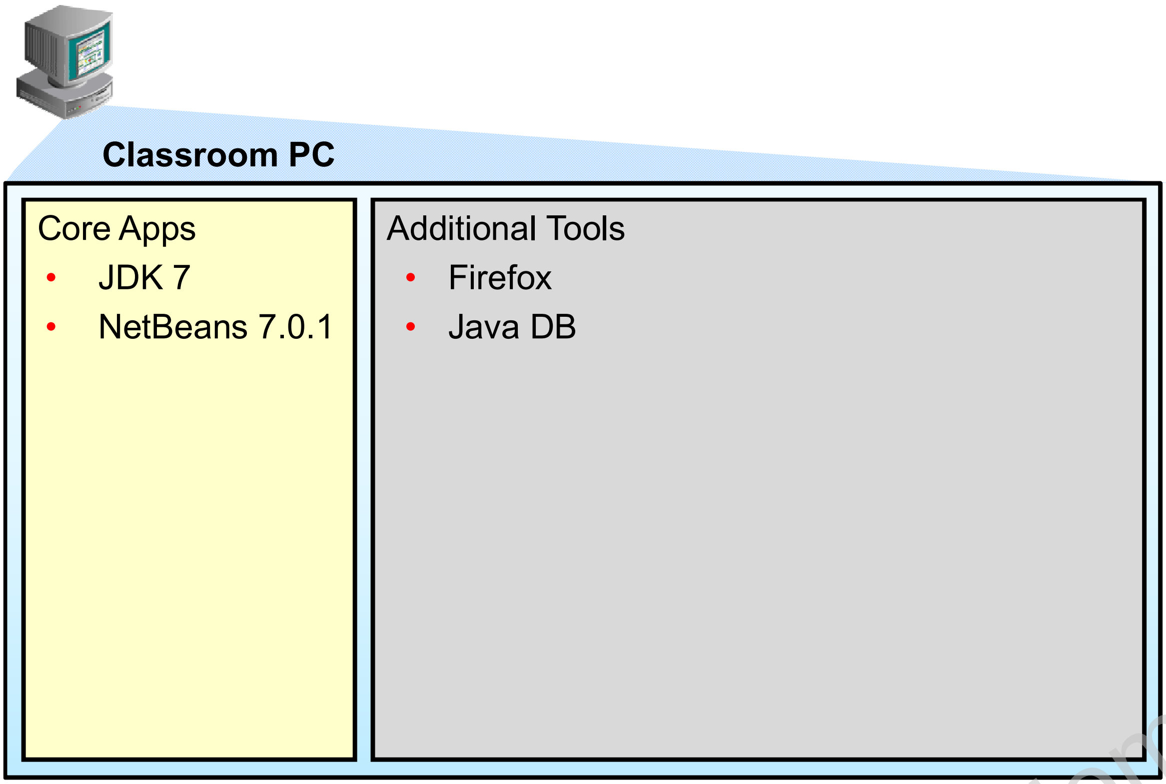
- Compile and run Java applications
- Create Java classes
- Create object instances using the `new` keyword
- Declare Java primitive and reference variables
- Declare Java methods using return values and parameters
- Use conditional constructs such as `if` and `switch` statements
- Use looping constructs such as `for`, `while`, and `do` loops
- Declare and instantiate Java arrays
- Use the Java Platform, Standard Edition API Specification (Javadocs)

Class Introductions

Briefly introduce yourself:

- Name
- Title or position
- Company
- Experience with Java programming and Java applications
- Reasons for attending

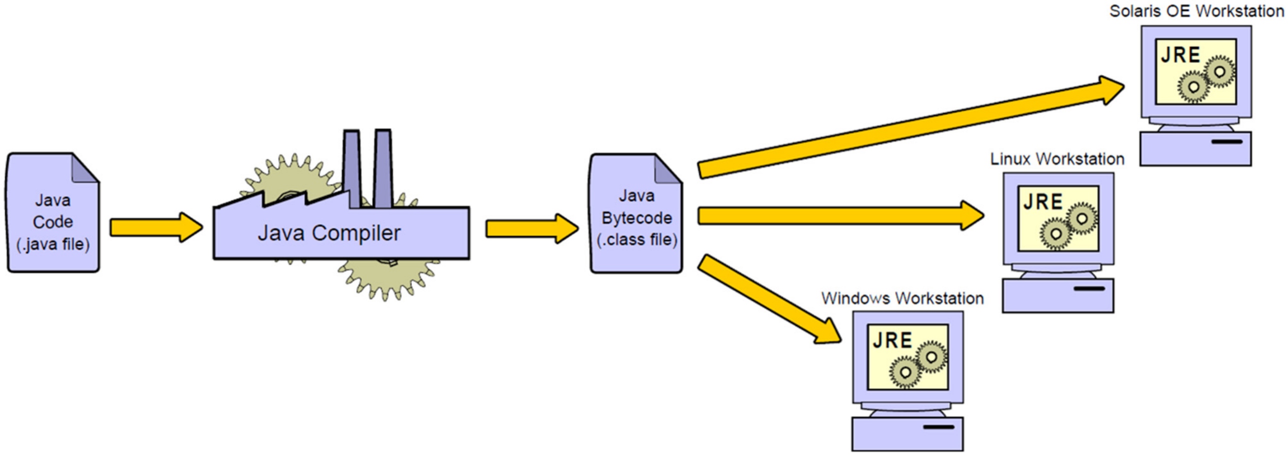
Course Environment



In this course, the following products are preinstalled for the lesson practices:

- **JDK 7:** The Java SE Development Kit includes the command-line Java compiler (`javac`) and the Java Runtime Environment (JRE), which supplies the `java` command needed to execute Java applications.
- **Firefox:** A web browser is used to view the HTML documentation (Javadoc) for the Java SE Platform libraries.
- **NetBeans 7.0.1:** The NetBeans IDE is a free and open-source software development tool for professionals who create enterprise, web, desktop, and mobile applications. NetBeans 7.0.1 fully supports the Java SE 7 Platform. Support is provided by Oracle's Development Tools Support offering.
- **Java DB:** Java DB is Oracle's supported distribution of the open-source Apache Derby 100% Java technology database. It is fully transactional, secure, easy-to-use, standards-based SQL, JDBC API, and Java EE yet small, only 2.5 MB.

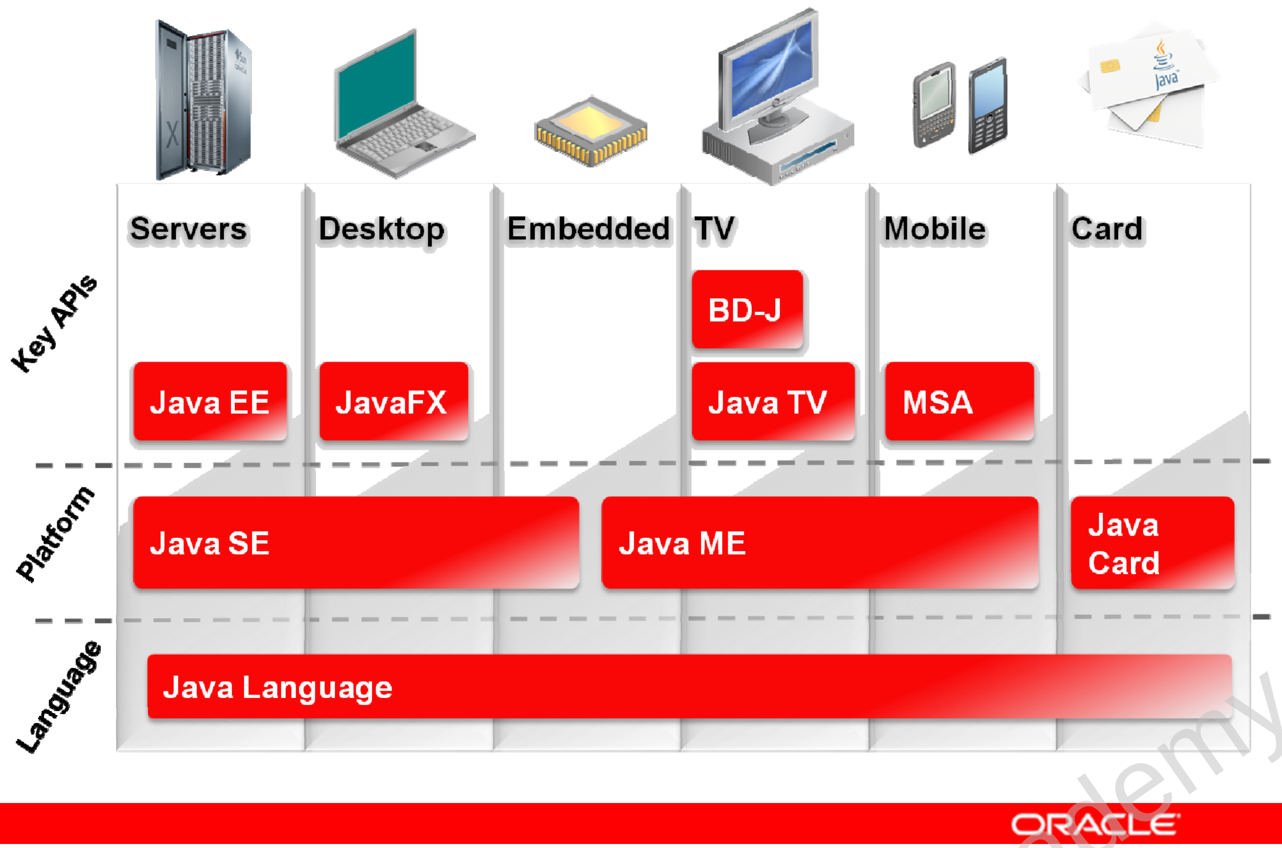
Java Programs Are Platform-Independent



Platform-Independent Programs

Java technology applications are written in the Java programming language and compiled to Java bytecode. Bytecode is executed on the Java platform. The software that provides you with a runnable Java platform is called a Java Runtime Environment (JRE). A compiler, included in the Java SE Development Kit (JDK), is used to convert Java source code to Java bytecode.

Java Technology Product Groups



Identifying Java Technology 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. Among other Java technologies, each edition includes a Software Development kit (SDK) that allows programmers to create, compile, and execute Java technology programs on a particular platform:

- **Java Platform, Standard Edition (Java SE):** Develops applets and applications that run within Web browsers and on desktop computers, respectively. For example, you can use the Java SE Software Development Kit (SDK) to create a word processing program for a personal computer. You can also use the Java SE to create an application that runs in a browser.

Note: Applets and applications differ in several ways. Primarily, applets are launched inside a web browser, whereas applications are launched within an operating system.

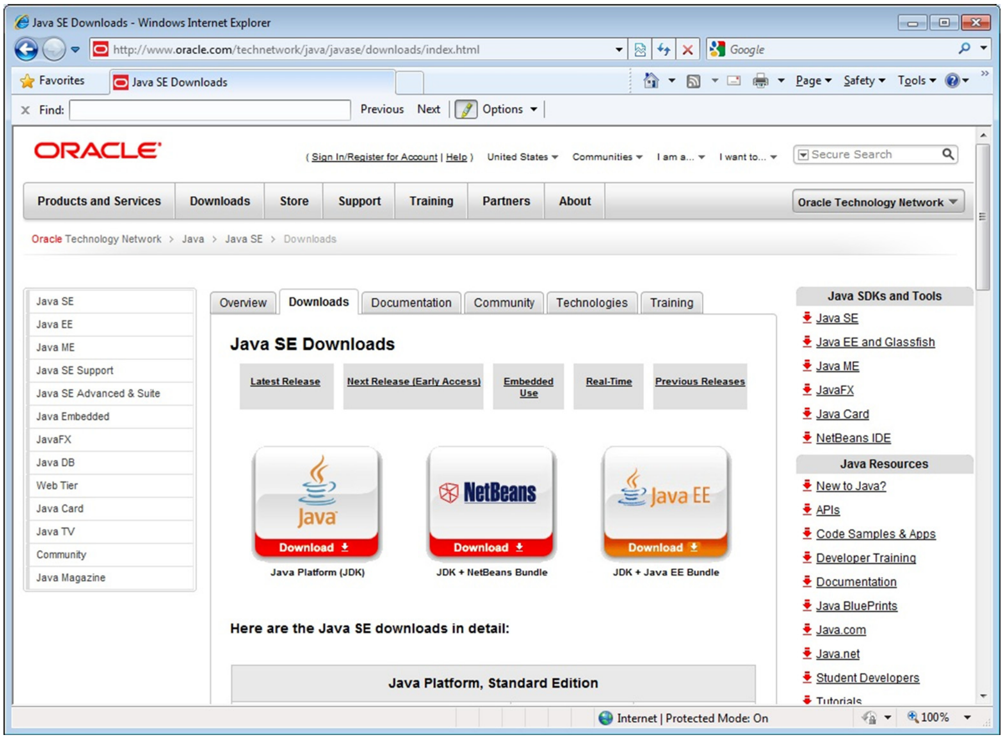
Java SE Platform Versions

Year	Developer Version (JDK)	Platform
1996	1.0	1
1997	1.1	1
1998	1.2	2
2000	1.3	2
2002	1.4	2
2004	1.5	5
2006	1.6	6
2011	1.7	7

How to Detect Your Version

If Java SE is installed on your system, you can detect the version number by running `java -version`. Note that the `java` command is included with the Java Runtime Environment (JRE). As a developer, you also need a Java compiler, typically `javac`. The `javac` command is included in the Java SE Development Kit (JDK). Your operation system's `PATH` may need to be updated to include the location of `javac`.

Downloading and Installing the JDK



1. Go to http://www.oracle.com/technetwork/java/javase/downloads/index.html.
2. Choose the Java Platform, Standard Edition (Java SE) link.
3. Download the version that is appropriate for your operation system.
4. Follow the installation instructions.
5. Set your `PATH`.

Java in Server Environments

Java is common in enterprise environments:

- Oracle Fusion Middleware
  - Java application servers
    - GlassFish
    - WebLogic
- Database servers
  - MySQL
  - Oracle Database

Enterprise Environments

In this course, you develop Java SE applications. There are standard patterns you need to follow when implementing Java SE applications, such as always creating a `main` method that may be different when implementing enterprise applications. Java SE is only the starting point in your path to becoming a Java developer. Depending on the needs of your organization, you may be required to develop applications that run inside Java EE application servers or other types of Java middleware.

Often, you will also need to manipulate information stored inside relational databases such as MySQL or Oracle Database. This course introduces you to the fundamentals of database programming.

The Java Community

What Is the Java Community?

At a very high level, *Java Community* is the term used to refer to the many individuals and organizations that develop, innovate, and use Java technology. This community includes developers as individuals, organizations, businesses, and open-source projects.

It is very common for you to download and use Java libraries from non-Oracle sources within the Java community. For instance, in this course, you use an Apache-developed JDBC library to access a relational database.

The Java Community Process (JCP)

The JCP is used to develop new Java standards:

- http://jcp.org
- Free download of all Java Specification Requests (JSRs)
- Early access to specifications
- Public review and feedback opportunities
- Open membership

JCP.next

The JCP produces the JSRs that outline the standards of the Java platform. The behavior of the JCP itself is also defined and improved through the JSR process. The JCP is evolving and its improvements are defined in JSR-348. JSR-348 introduces changes in the areas of transparency, participation, agility, and governance.

- **Transparency**: In the past, some aspects of the development of a JSR may have occurred behind closed doors. Transparent development is now the recommended practice.
- **Participation:** Individuals and Java User Groups are encouraged to become active in the JCP.
- **Agility**: Slow-moving JSRs are now actively discouraged.
- **Governance**: The SE and ME expert groups are merging into a single body.

OpenJDK

OpenJDK is the open-source implementation of Java:

- http://openjdk.java.net/
- GPL licensed open-source project
- JDK reference implementation
- Where new features are developed
- Open to community contributions
- Basis for Oracle JDK

Why OpenJDK Is Important

Because it is open source, OpenJDK enables users to port Java to operating systems and hardware platforms of their choosing. Ports are underway for many platforms (besides those already supported) including FreeBSD, OpenBSD, NetBSD, and MacOS X.

Oracle Java SE Support

Java is available free of charge. However, Oracle does provide pay-for Java solutions:

- The Java SE Support Program provides updates for end-of-life Java versions.
- Oracle Java SE Advanced and Oracle Java SE Suite:
  - JRockit Mission Control
  - Memory Leak Detection
  - Low Latency GC (Suite)
  - JRockit Virtual Edition (Suite)

Still Free

Java (Oracle JDK) is freely available at no cost. Oracle offers advanced commercial solutions at cost. The previously offered "Java for Business" program has been replaced by Oracle Java SE Support, which provides access to Oracle Premier Support and the Oracle Java SE Advanced and Oracle Java SE Suite binaries. For more information, visit http://www.oracle.com/us/technologies/java/java-se-suite-394230.html.

Additional Resources

Topic	Website
Education and Training	http://education.oracle.com
Product Documentation	http://www.oracle.com/technology/documentation
Product Downloads	http://www.oracle.com/technology/software
Product Articles	http://www.oracle.com/technology/pub/articles
Product Support	http://www.oracle.com/support
Product Forums	http://forums.oracle.com
Product Tutorials	http://www.oracle.com/technetwork/tutorials/index.html
Sample Code	https://www.samplecode.oracle.com
Oracle Technology Network for Java Developers	http://www.oracle.com/technetwork/java/index.html
Oracle Learning Library	http://www.oracle.com/goto/oll

The table in the slide lists various web resources that are available for you to learn more about Java SE programming.

Summary

In this lesson, you should have learned about:

- The course objectives
- Software used in this course
- Java platforms (ME, SE, and EE)
- Java SE version numbers
- Obtaining a JDK
- The open nature of Java and its community
- Commercial support options for Java SE

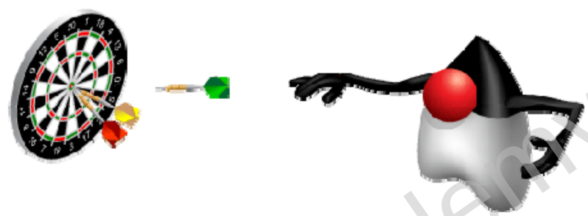
2

Java Syntax and Class Review

Objectives

After completing this lesson, you should be able to do the following:

- Create simple Java classes
  - Create primitive variables
  - Manipulate Strings
  - Use `if-else` and `switch` branching statements
  - Iterate with loops
  - Create arrays
- Use Java fields, constructors, and methods
- Use `package` and `import` statements



Java Language Review

This lesson is a review of fundamental Java and programming concepts. It is assumed that students are familiar with the following concepts:

- The basic structure of a Java class
- Program block and comments
- Variables
- Basic `if-else` and `switch` branching constructs
- Iteration with `for` and `while` loops

Class Structure

```
package <package_name>;

import <other_packages>;

public class ClassName {
   <variables(also known as fields)>;

   <constructor method(s)>;

   <other methods>;
}
```


A Java class is described in a text file with a `.java` extension. In the example shown, the Java keywords are highlighted in bold.

- The `package` keyword defines where this class lives relative to other classes, and provides a level of access control. You use access modifiers (such as `public` and `private`) later in this lesson.
- The `import` keyword defines other classes or groups of classes that you are using in your class. The `import` statement helps to narrow what the compiler needs to look for when resolving class names used in this class.
- The `class` keyword precedes the name of this class. The name of the class and the file name must match when the class is declared `public` (which is a good practice). However, the keyword `public` in front of the `class` keyword is a modifier and is not required.
- Variables, or the data associated with programs (such as integers, strings, arrays, and references to other objects), are called *instance fields* (often shortened to *fields*).
- Constructors are functions called during the creation (instantiation) of an object (a representation in memory of a Java class.)
- Methods are the functions that can be performed on an object. They are also referred to as *instance methods*.

A Simple Class

A simple Java class with a main method:

```
public class Simple {

    public static void main(String args[]){

    }
}
```


To run a Java program, you must define a `main` method as shown in the slide. The `main` method is automatically called when the class is called from the command line. Command-line arguments are passed to the program through the `args[]` array.

Note: A method that is modified with the keyword `static` is invoked without a reference to a particular object. The class name is used instead. These methods are referred to as *class methods*. The `main` method is a special method that is invoked when this class is run using the Java runtime.

Code Blocks

- Every class declaration is enclosed in a code block.
- Method declarations are enclosed in code blocks.
- Java fields and methods have block (or class) scope.
- Code blocks are defined in braces:

```
{ }
```

- Example:

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

ORACLE

Java fields (variables) and methods have a class scope defined by the opening left curly brace and ending at the closing right curly brace.

Class scope allows any method in the class to call or invoke any other method in the class. Class scope also allows any method to access any field in the class.

Code blocks are always defined using braces `{}`. A block is executed by executing each of the statements defined within the block in order from first to last (left to right).

The Java compiler ignores white space that precedes or follows the elements that make up a line of code. Line indentation is not required but makes code much more readable. In this course, the line indentation is four spaces, which is the default line indentation used by the NetBeans IDE.

Primitive Data Types

Integer	Floating Point	Character	True False
byte short int long	float double	char	boolean
1, 2, 3, 42 07 0xff	3.0F .3337F 4.022E23	'a' '\u0061' '\n'	true false
0	0.0	'\u0000'	false

Append uppercase or lowercase "`L`" or "`F`" to the number to specify a long or a float number.

ORACLE

Integer

Java provides four different integer types to accommodate different size numbers. All the numeric types are signed, which means that they can hold positive or negative numbers.

The integer types have the following ranges:

- `byte` range is –128 to +127. Number of bits = 8.
- `short` range is –32,768 to +32,767. Number of bits = 16.
- `int` range is –2,147,483,648 to +2,147,483,647. The most common integer type is `int`. Number of bits = 32.
- `long` range is –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807. Number of bits = 64.

Floating Point

The floating-point types hold numbers with a fractional part and conform to the IEEE 754 standard. There are two types of floating points: `float` and `double`.

`double` is so called because it provides double the precision of `float`. A `float` uses 32 bits to store data, whereas a `double` uses 64 bits.

Character

The `char` type is used for individual characters, as opposed to a string of characters (which is implemented as a `String` object). Java supports Unicode, an international standard for representing a character in any written language in the world in a single 16-bit value. The first 256 characters coincide with the ISO Latin 1 character set, part of which is ASCII.

Boolean

The `boolean` type can hold either `true` or `false`.

Note: `true` and `false` may appear to be keywords, but they are technically boolean literals.

Default Values

If a value is not specified, a default value is used. The values in red in the slide are the defaults used. The default `char` value is `null` (represented as `'\u0000'`), and the default value for `boolean` is `false`.

Note: Local variables (that is, variables declared within methods) do not have a default value. An attempt to use a local variable that has not been assigned a value will cause a compiler error. It is a good practice always to supply a default value to any variable.

Java SE 7 Numeric Literals

In Java SE 7 (and later versions), any number of underscore characters (_) can appear between digits in a numeric field. This can improve the readability of your code.

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```


Rules for Literals

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an `F` or `L` suffix
- In positions where a string of digits is expected

Note: The Java language is case-sensitive. In Java, the variable `creditCardNumber` is different from `CREDITCARDNUMBER`. Convention indicates that Java variables and method names use "lower camel case"—lowercase for the first letter of the first element of a variable name and uppercase for the first letter of subsequent elements.

Java SE 7 Binary Literals

In Java SE 7 (and later versions), binary literals can also be expressed using the binary system by adding the prefixes `0b` or `0B` to the number:

```
// An 8-bit 'byte' value:
byte aByte = 0b0010_0001;

// A 16-bit 'short' value:
short aShort = (short)0b1010_0001_0100_0101;

// Some 32-bit 'int' values:
int anInt1 = 0b1010_0001_0100_0101_1010_0001_0100_0101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.
```


Binary literals are Java `int` values. A cast is required when the integer value of the literal exceeds the greatest non-negative value that the type can hold. For example:

```
byte aByte = 0b0111_1111;  // aByte is 127
byte aByte = 0b1000_0000;  // compiler error - a cast is required
                           // (value is -128)
```

Operators

- Simple assignment operator
  - = Simple assignment operator
- Arithmetic operators
  - + Additive operator (also used for String concatenation)
  - – Subtraction operator
  - * Multiplication operator
  - / Division operator
  - % Remainder operator
- Unary operators
  - + Unary plus operator; indicates positive
  - – Unary minus operator; negates an expression
  - ++ Increment operator; increments a value by 1
  - -- Decrement operator; decrements a value by 1
  - ! Logical complement operator; inverts the value of a boolean

Because numbers have been introduced, the slide shows a list of common operators. Most are common to any programming language, and a description of each is provided in the slide.

The binary and bitwise operators have been omitted for brevity. For details about those operators, refer to the Java Tutorial:

http://download.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Note: Operators have definitive precedence. For the complete list, see the Java Tutorial link mentioned above. Precedence can be overridden using parentheses.

Strings

```
1 public class Strings {
2
3     public static void main(String args[]){
4
5         char letter = 'a';
6
7         String string1 = "Hello";
8         String string2 = "World";
9         String string3 = "";
10        String dontDoThis = new String ("Bad Practice");
11
12        string3 = string1 + string2; // Concatenate strings
13
14        System.out.println("Output: " + string3 + " " + letter);
15
16    }
17 }
```

String literals are also String objects.

ORACLE

The code in the slide demonstrates how text characters are represented in Java. Single characters can be represented with the `char` type. However, Java also includes a `String` type for representing multiple characters. Strings can be defined as shown in the slide and combined using the "`+`" sign as a concatenation operator.

The output from the code in the slide is:

```
Output: HelloWorld a
```

Caution: Strings should always be initialized using the assignment operator "`=`" and text in quotation marks, as shown in the examples. The use of `new` to initialize a `String` is strongly discouraged. The reason is that "`Bad Practice`" in line 10 is a `String` literal of type `String`, Using the `new` keyword simply creates another instance functionally identical to the literal. If this statement appeared inside of a loop that was frequently invoked, there could be a lot of needless `String` instances created.

String Operations

```
public class StringOperations {
    public static void main(String arg[]){
        String string2 = "World";
        String string3 = "";

        string3 = "Hello".concat(string2);
        System.out.println("string3: " + string3);

        // Get length
        System.out.println("Length: " + string1.length());

        // Get SubString
        System.out.println("Sub: " + string3.substring(0, 5));

        // Uppercase
        System.out.println("Upper: " + string3.toUpperCase());
    }
}
```

String literals are automatically created as String objects if necessary.

This slide demonstrates some common string methods, including:

- `concat()`
- `length()`
- `substring()`
- `toUpperCase()`

To see what other methods can be used on a `String`, see the API documentation.

The output from the program is:

```
string3: HelloWorld
Length: 5
Sub: Hello
Upper: HELLOWORLD
```

Note: `String` is a class, not a primitive type. Instances of the class `String` represent sequences of Unicode characters. `String` literals are stored as `String` objects and "interned", meaning that for strings with matching characters, they all point to the same `String` object.

`if else`

```
public class IfElse {

    public static void main(String args[]){
        long a = 1;
        long b = 2;

        if (a == b){
            System.out.println("True");
        } else {
            System.out.println("False");
        }

    }
}
```


The example in the slide demonstrates the syntax for an `if-else` statement in Java.

The output from the code in the slide is as follows:

```
False
```

Logical Operators

- Equality and relational operators
  - == Equal to
  - != Not equal to
  - > Greater than
  - >= Greater than or equal to
  - < Less than
  - <= Less than or equal to
- Conditional operators
  - && Conditional-AND
  - || Conditional-OR
  - ?: Ternary (shorthand for `if-then-else` statement)
- Type comparison operator
  - `instanceof` Compares an object to a specified type

The slide shows a summary of the logic and conditional operators in Java.

Arrays and `for-each` Loop

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

        String[] names = new String[3];

        names[0] = "Blue Shirt";
        names[1] = "Red Shirt";
        names[2] = "Black Shirt";

         int[] numbers = {100, 200, 300};

         for (String name:names){
             System.out.println("Name: " + name);
         }

         for (int number:numbers){
             System.out.println("Number: " + number);
         }
    }
}
```

Arrays are objects.
Array objects have a final field length.

ORACLE

This class demonstrates how to define arrays in Java. The first example creates a `String` array and initializes each element separately. The second `int` array is defined in a single statement.

Each array is iterated through using the Java `for-each` construct. The loop defines an element which will represent each element of the array and the array to loop through. The output of the class is shown here:

```
Name: Blue Shirt
Name: Red Shirt
Name: Black Shirt
Number: 100
Number: 200
Number: 300
```

Note: Arrays are also objects by default. All arrays support the methods of the class Object. You can always obtain the size of an array using its `length` field.

`for` Loop

```
public class ForLoop {

    public static void main(String args[]){

        for (int i = 0; i < 9;  i++ ){
            System.out.println("i: " + i);
        }

    }
}
```


The classic `for` loop is shown in the slide. A counter is initialized and incremented for each step of the loop. When the condition statement evaluates to false (when `i` is no longer less than 9), the loop exits. Here is the sample output for this program.

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
```

`while` Loop

```
public class WhileLoop {

   public static void main(String args[]){

       int i = 0;
       int[] numbers = {100, 200, 300};

       while (i < numbers.length ){
           System.out.println("Number: " + numbers[i]);
            i++;
        }
    }
}
```


The `while` loop performs a test and continues if the expression evaluates to `true`. The `while` loop, shown here, iterates through an array using a counter. Here is the output from the code in the slide:

```
Number: 100
Number: 200
Number: 300
```

Note: There is also a `do-while` loop, where the test after the expression has run at least once:

```
class DoWhileDemo {
     public static void main(String[] args){
          int count = 1;
          do {
               System.out.println("Count is: " + count);
               count++;
          } while (count <= 11);
     }
}
```

String `switch` Statement

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

       String color = "Blue";
       String shirt = " Shirt";

       switch (color){
           case "Blue":
               shirt = "Blue" + shirt;
                break;
            case "Red":
                shirt = "Red" + shirt;
                break;
            default:
                shirt = "White" + shirt;
        }

        System.out.println("Shirt type: " + shirt);
    }
}
```


This example shows a `switch` statement in Java using a `String`. Prior to version 7 of Java, only enums and `byte`, `short`, `char`, and `int` primitive data types could be used in a `switch` statement. You will see enums in the lesson titled "Advanced Class Design."

Java Naming Conventions

```java
public class CreditCard {
    public final int VISA = 5001;
    public String accountName;
    public String cardNumber;
    public Date expDate;

    public double getCharges(){
        // ...
    }

    public void disputeCharge(String chargeId, float amount){
        // ...
    }
}
```

Class names are nouns in upper camel case.

Constants should be declared in all uppercase. letters

Variable names are short but meaningful in lower camel case.

Methods should be verbs, in lower camel case.

- Class names should be nouns in mixed case, with the first letter uppercase and the first letter of each internal word capitalized. This is approach is termed "upper camel case."
- Methods should be verbs in mixed case, with the first letter lowercase and the first letter of each internal word capitalized. This is termed "lower camel case."
- Variable names should be short but meaningful. The choice of a variable name should be mnemonic: designed to indicate to the casual observer the intent of its use.
- One-character variable names should be avoided except as temporary "throwaway" variables.
- Constants should be declared using all uppercase letters. **Note:** The keyword `final` is used to declare a variable whose value may only be assigned once. Once a `final` variable has been assigned, it always contains the same value. You will learn more about the keyword `final` in the lesson "Advanced Class Design."

For the complete *Code Conventions for the Java Programming Language* document, go to http://www.oracle.com/technetwork/java/codeconv-138413.html.

A Simple Java Class: `Employee`

A Java class is often used to represent a concept.

```
package com.example.domain;
public class Employee {   class declaration
    public int empId;
    public String name;
    public String ssn;          fields
    public double salary;

    public Employee () {    a constructor
    }

     public int getEmpId () {    a method
       return empId;
    }
}
```


A Java class is often used to store or represent data for the construct that the class represents. For example, you could create a model (a programmatic representation) of an Employee. An `Employee` object defined using this model will contain values for `empId`, `name`, Social Security Number (`ssn`), and `salary`.

The constructor in this class creates an instance of an object called `Employee`.

A constructor is unique in Java. A constructor is used to create an instance of a class. Unlike methods, constructors do not declare a return type, and are declared with the same name as their class. Constructors can take arguments and you can declare more than one constructor, as you will see in the lesson titled “Java Class Design.”

Methods

When a class has data fields, a common practice is to provide methods for storing data (setter methods) and retrieving data (getter methods) from the fields.

```
package com.example.domain;
public class Employee {
    public int empId;
    // other fields...
    public void setEmpId(int empId) {
        this.empId = empId;
    }
    public int getEmpId() {
        return empId;
    }
    // getter/setter methods for other fields...
}
```

Often a pair of methods to set and get the current field value.

Adding Instance Methods to the `Employee` Class

A common practice is to create a set of methods that manipulate field data: methods that set the value of each field, and methods that get the value of each field. These methods are called *accessors* (getters) and *mutators* (setters).

The convention is to use `set` and `get` plus the name of the field with the first letter of the field name capitalized (lower camel case). Most modern integrated development environments (IDEs) provide an easy way to automatically generate the accessor (getter) and mutator (setter) methods for you.

Notice that the set methods use the keyword `this`. The `this` keyword allows the compiler to distinguish between the field name of the class (`this`) and the parameter name being passed in as an argument. Without the keyword `this`, the net effect is you are assigning a value to itself. (In fact, NetBeans provides a warning: "Assignment to self.")

In this simple example, you could use the `setName` method to change the employee name and the `setSalary` method to change the employee `salary`.

Note: The methods declared on this slide are called *instance* methods. They are invoked using an instance of this class (described on the next slide.)

Creating an Instance of an Object

To construct or create an instance (object) of the `Employee` class, use the `new` keyword.

```
/* In some other class, or a main method */
Employee emp = new Employee();
emp.empId = 101;    // legal if the field is public,
                    // but not good OO practice
emp.setEmpId(101); // use a method instead
emp.setName("John Smith");
emp.setSsn("011-22-3467");
emp.setSalary(120345.27);
```

Invoking an instance method.

- In this fragment of Java code, you construct an instance of the `Employee` class and assign the reference to the new object to a variable called `emp`.
- Then you assign values to the `Employee` object.

Creating an instance of the Employee Class

In order to use the `Employee` class to hold the information of an employee, you need to allocate memory for the `Employee` object and call a constructor method in the class. An instance of an object is created when you use the `new` keyword with a constructor. All of the fields declared in the class are provided memory space and initialized to their default values. If the memory allocation and constructor are successful, a reference to the object is returned as a result. In the example in the slide, the reference is assigned to a variable called `emp`.

To store values (data) into the `Employee` object instance, you could just assign values to each field by accessing the fields directly. However, this is not a good practice and negates the principle of encapsulation. Instead, you should invoke instance methods and pass a value to the method to set the value of each data field. Later in this lesson you will look at restricting access to the fields to promote encapsulation.

Once all the data fields are set with values, you have an instance of an `Employee` with an `empId` with a value of 101, `name` with the string John Smith, Social Security number string (`ssn`) set to 011-22-3467, and `salary` with the value of 120,345.27.

Constructors

```
public class Employee {
    public Employee() {
    }
}
```

A simple no-argument (no-arg) constructor.

```
Employee emp = new Employee();
```

- A constructor is used to create an instance of a class.
- Constructors can take parameters.
- A constructor that takes no arguments is called a *no-arg* constructor.

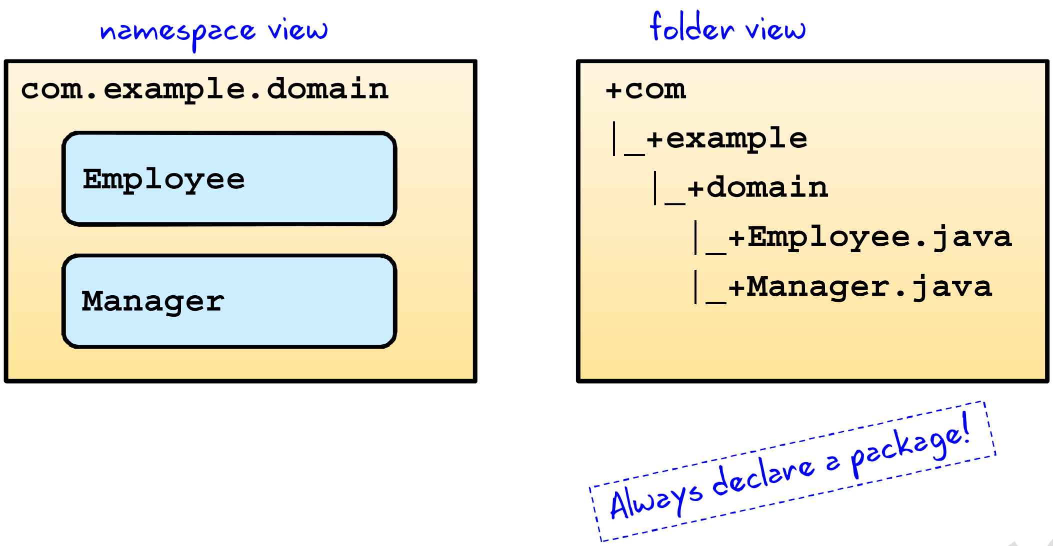
ORACLE

A constructor is used to create an object. In the Java programming language, constructors are declared with the same name as their class used to create an instance of an object. Constructors are invoked using the `new` keyword.

Constructors are covered in more detail in the lesson titled "Encapsulation and Subclassing."

`package` Statement

The `package` keyword is used in Java to group classes together. A package is implemented as a folder and, like a folder, provides a *namespace* to a class.



Packages

In Java, a package is a group of (class) types. There can be only one `package` declaration for a file.

Packages are more than just a convenience. Packages create a namespace, a logical collection of things, like a directory hierarchy.

It is a good practice to always use a `package` declaration. The `package` declaration is always at the top of the file.

`import` Statements

The `import` keyword is used to identify classes you want to reference in your class.

- The `import` statement provides a convenient way to identify classes that you want to reference in your class.

```
import java.util.Date;
```

- You can import a single class or an entire package:

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

- You can include multiple `import` statements:

```
import java.util.Date;
import java.util.Calendar;
```

- It is good practice to use the full package and class name rather than the wildcard `*` to avoid class name conflicts.

ORACLE

Imports

You could refer to a class using its fully qualified namespace in your applications, as in the following example:

```
java.util.Date date = new java.util.Date();
```

But that would quickly lead to a lot of typing! Instead, Java provides the import statement to allow you to declare that you want to reference a class in another package.

Note: It is a good practice to use the specific, fully qualified package and class name to avoid confusion when there are two classes with the same name, as in the following example: `java.sql.Date` and `java.util.Date`. The first is a `Date` class used to store a `Date` type in a database, and `java.util.Date` is a general purpose `Date` class. As it turns out, `java.sql.Date` is a subclass of `java.util.Date`. This is covered in more detail later in the course.

Note: Modern IDEs, like NetBeans and Eclipse, automatically search for and add import statements for you. In NetBeans, for example, use the Ctrl + Shift + I key sequence to fix imports in your code.

More on `import`

- Import statements follow the package declaration and precede the class declaration.
- An import statement is not required.
- By default, your class always imports `java.lang.*`
- You do not need to import classes that are in the same package:

```
package com.example.domain;
import com.example.domain.Manager; // unused import
```


Details about the `java.lang` package and its classes are covered later in the course.

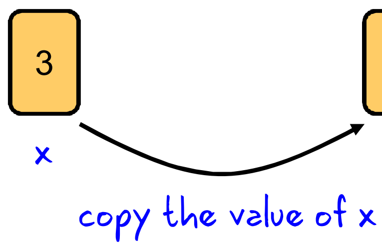
Java Is Pass-By-Value

The Java language (unlike C++) uses pass-by-value for all assignment operations.

- To visualize this with primitives, consider the following:

```
int x = 3;
int y = x;
```

- The value of `x` is copied and passed to `y`:



- If `x` is later modified (for example, `x = 5;`), the value of `y` remains unchanged.

The Java language uses pass-by-value for all assignment operations. This means that the argument on the right side of the equal sign is evaluated, and the value of the argument is assigned to the left side of the equal sign.

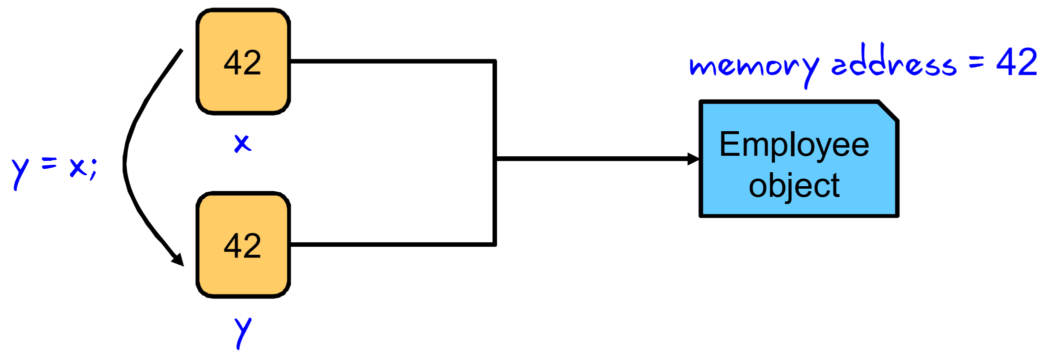
For Java primitives, this is straightforward. Java does not pass a reference to a primitive (such as an integer), but rather a copy of the value.

Pass-By-Value for Object References

For Java objects, the *value* of the right side of an assignment is a reference to memory that stores a Java object.

```
Employee x = new Employee();
Employee y = x;
```

- The reference is some address in memory.



- After the assignment, the value of `y` is the same as the value of `x`: a reference to the same `Employee` object.

For Java objects, the value of an object reference is the memory pointer to the instance of the `Employee` object created.

When you assign the value of `x` to `y`, you are not creating a new `Employee` object, but rather a copy of the value of the reference.

Note: An object is a class instance or an array. The reference values (references) are pointers to these objects, and a special `null` reference, which refers to no object.

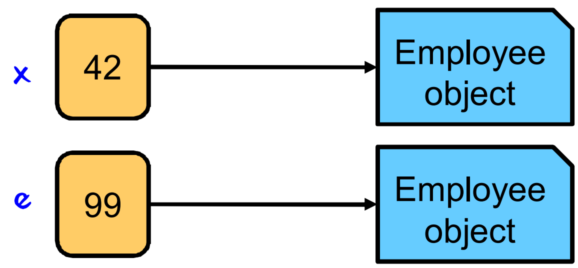
Objects Passed as Parameters

- Whenever a new object is created, a new reference is created. Consider the following code fragments:

```
Employee x = new Employee();
foo(x);
```

```
public void foo(Employee e) {
    e = new Employee();
    e.setSalary (1_000_000.00);  // What happens to x here?
}
```

- The value of `x` is unchanged as a result of the method call `foo`:



In the first line of code, a new object (Employee) is created and the reference to that object is assigned to the variable `x`.

In the second line of code, the value of that reference is passed to a method called `foo`.

When the `foo` method is called, (`Employee e`) holds a reference to the Employee object, `x`. In the next line, the value of `e` is now a new Employee object, by virtue of the call to the constructor.

The reference to the `x` object is replaced by a reference to a new object. The `x` object remains unchanged.

Note: The object `e`, created inside of the method `foo`, can no longer be referenced when the method finishes. As a result, it will be eligible for garbage collection at some future point.

If the code in the `foo` method was written differently, like this:

```
public void foo(Employee e) {
    e.setSalary(1_000_000.00):
}
```

Then referenced object that the `setSalary` method is being called on is the object referenced by `x`, and after the `foo` method returns, the object `x` is modified..

Note: The memory locations 42 and 99 are simply for illustrative purposes!

How to Compile and Run

Java class files must be compiled before running them.
To compile a Java source file, use the Java compiler (`javac`).

```
javac -cp <path to other classes> -d <complier output
path> <path to source>.java
```

- You can use the `CLASSPATH` environment variable to the directory above the location of the package hierarchy.
- After compiling the source `.java` file, a `.class` file is generated.
- To run the Java application, run it using the Java interpreter (`java`):

```
java -cp <path to other classes> <package
name>.<classname>
```


CLASSPATH

The default value of the `classpath` is the current working directory (.), however, specifying the `CLASSPATH` variable or the `-cp` command line switch overrides this value.

The `CLASSPATH` variable is used by both the Java compiler and the Java interpreter (runtime).

The `classpath` can include:

- A list of directory names (separated by semicolons in Windows and colons in UNIX)
  - The classes are in a package tree relative to any of the directories on the list.
- A `.zip` or `.jar` file name that is fully qualified with its path name
  - The classes in these files must be zipped with the path names that are derived from the directories formed by their package names.

Note: The directory containing the root name of the package tree must be added to the `classpath`. Consider putting `classpath` information in the command window or even in the Java command, rather than hard-coding it in the environment.

Compiling and Running: Example

- Assume that the class shown in the notes is in the directory `D:\test\com\example`:

```
javac -d D:\test D:\test\com\example\HelloWorld.java
```

- To run the application, you use the interpreter and the fully qualified class name:

```
java -cp D:\test com.example.HelloWorld
Hello World!

java -cp D:\test com.example.HelloWorld Tom
Hello Tom!
```

- The advantage of an IDE like NetBeans is that management of the class path, compilation, and running the Java application are handled through the tool.

Example

Consider the following simple class in a file named `HelloWorld.java` in the `D:\test\com\example` directory:

```
package com.example;
public class HelloWorld {
    public static void main (String [] args) {
        if (args.length < 1) {
            System.out.println("Hello World!");
        } else {
            System.out.println("Hello " + args[0] + "!");
        }
    }
}
```

Java Class Loader

During execution of a Java program, the Java Virtual Machine loads the compiled Java class files using a Java class of its own called the "class loader" (`java.lang.ClassLoader`).

- The class loader is called when a class member is used for the first time:

```
public class Test {
    public void someOperation() {
        Employee e = new Employee();
        //...
    }
}
```

The class loader is called to "load" this class into memory.

```
Test.class.getClassLoader().loadClass("Employee");
```


Typically, the use of the class loader is completely invisible to you. You can see the results of the class loader by using the `-verbose` flag when you run your application. For example:

```
java -verbose -classpath D:\test com.example.HelloWorld
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects
file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
[Loaded java.lang.ClassLoader from shared objects file]
... and many more
```

Garbage Collection

When an object is instantiated using the `new` keyword, memory is allocated for the object. The scope of an object reference depends on where the object is instantiated:

```
public void someMethod() {
    Employee e = new Employee();
    // operations on e
}
```

Object `e` scope ends here.

- When `someMethod` completes, the memory referenced by `e` is no longer accessible.
- Java's garbage collector recognizes when an instance is no longer accessible and eligible for collection.

Note: When an object's memory is freed depends upon a number of factors.

Java's garbage collection scheme can be tuned depending on the type of application you are creating. For more information, consider taking the Oracle University course *Java Performance Tuning and Optimization* (D69518GC10).

Summary

In this lesson, you should have learned how to:

- Create simple Java classes
  - Create primitive variables
  - Manipulate Strings
  - Use `if-else` and `switch` branching statements
  - Iterate with loops
  - Create arrays
- Use Java fields, constructors, and methods
- Use `package` and `import` statements

Quiz

In the following fragment, what three issues can you identify?

```
package com.oracle.test;
public class BrokenClass {
    public boolean valid = "false";
    public String s = new String ("A new string");
    public int i = 40_000.00;
    public BrokenClass() { }
}
```

a. An `import` statement is missing.
b. The `boolean valid` is assigned a String.
c. `String s` is created using `new`.
d. `BrokenClass` method is missing a `return` statement.
e. Need to create a new `BrokenClass` object.
f. The integer value `i` is assigned a double.

Answer: b and f will cause compilation errors. c will compile but is not a good practice.

a. An `import` statement is not required, unless the class uses classes outside of `java.lang`.
d. `BrokenClass()` is a constructor.
e. Construction of a `BrokenClass` instance would typically happen in another class.

Quiz

Using the `Employee` class defined in this lesson, determine what will be printed in the following fragment:

```
public Employee changeName (Employee e, String name) {
    e.name = name;
    return (e);
}
//... in another method in the same class
Employee e = new Employee();
e.name = "Fred";
e = changeName(e, "Bob");
System.out.println (e.getName());
```

a. Fred
b. Bob
c. `null`
d. an empty String

Answer: b

Quiz

In the following fragment, what is the printed result?

```
public float average (int[] values) {
    float result = 0;
    for (int i = 1; i < values.length; i++)
        result += values[i];
    return (result/values.length);
}
// ... in another method in the same class
int[] nums = {100, 200, 300};
System.out.println (average(nums));
```

a. 100.00
b. 150.00
c. 166.66667
d. 200.00

Answer: c

Arrays begin with an index of 0. This average method is only averaging the 2nd through Nth values. Therefore, the result is the average of 200+300/3 = 166.66667. Change the for loop to int = 0; to properly calculate the average.

Practice 2-1 Overview: Creating Java Classes

This practice covers the following topics:

- Creating a Java class using the NetBeans IDE
- Creating a Java class with a `main` method
- Writing code in the body of the `main` method to create an instance of the `Employee` object and print values from the class to the console
- Compiling and testing the application using the NetBeans IDE

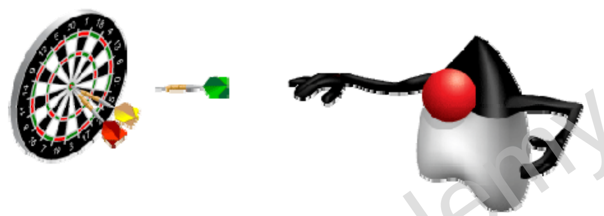
3

Encapsulation and Subclassing

Objectives

After completing this lesson, you should be able to do the following:

- Use encapsulation in Java class design
- Model business problems using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods
- Use variable argument methods



Encapsulation

The term *encapsulation* means to enclose in a capsule, or to wrap something around an object to cover it. In object-oriented programming, encapsulation covers, or wraps, the internal workings of a Java object.

- Data variables, or fields, are hidden from the user of the object.
- Methods, the functions in Java, provide an explicit service to the user of the object but hide the implementation.
- As long as the services do not change, the implementation can be modified without impacting the user.

The term *encapsulation* is defined by the Java Technology Reference Glossary as follows:

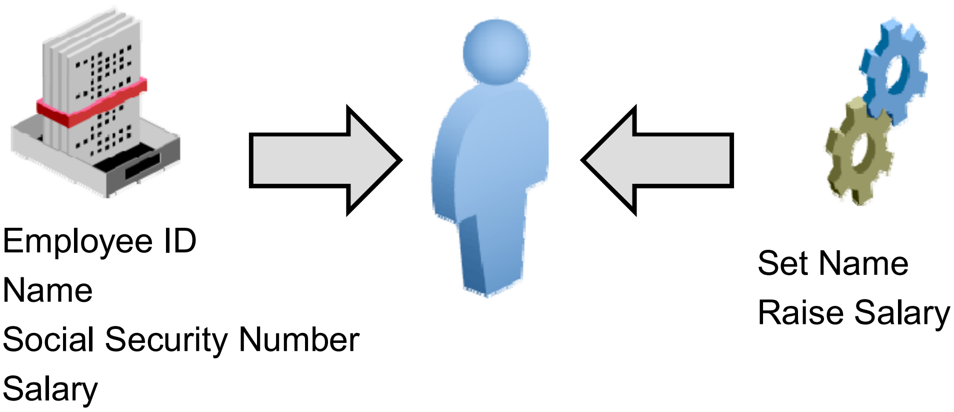
"The localization of knowledge within a module. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but if the services provided by the object remain the same, the code that uses the object can continue to use it without being rewritten."

An analogy for encapsulation is the steering wheel of a car. When you drive a car, whether it is your car, a friend's car, or a rental car, you probably never worry about how the steering wheel implements a right-turn or left-turn function. The steering wheel could be connected to the front wheels in a number of ways: ball and socket, rack and pinion, or some exotic set of servo mechanisms.

As long as the car steers properly when you turn the wheel, the steering wheel encapsulates the functions you need—you do not have to think about the implementation.

Encapsulation: Example

What data and operations would you encapsulate in an object that represents an employee?



A Simple Model

Suppose that you are asked to create a model of a typical employee. What data might you want to represent in an object that describes an employee?

- **Employee ID:** You can use this as a unique identifier for the employee.
- **Name**: Humanizing an employee is always a good idea!
- **Social Security Number:** For United States employees only. You may want some other identification for non-U.S. employees.
- **Salary**: How much the employee makes is always good to record.

What operations might you allow on the employee object?

- **Change Name:** If the employee gets married or divorced, there could be a name change.
- **Raise Salary:** Increases based on merit

After an employee object is created, you probably do not want to allow changes to the Employee ID or Social Security fields. Therefore, you need a way to create an employee without alterations except through the allowed methods.

Encapsulation: Private Data, Public Methods

One way to hide implementation details is to declare all of the fields `private`.

```
public class CheckingAccount {
    private int custID;
    private String name;
    private double amount;
    public CheckingAccount {
    }
    public void setAmount (double amount) {
        this.amount = amount;
    }
     public double getAmount () {
         return amount;
     }
     //... other public accessor and mutator methods
}
```

Declaring fields `private` prevents direct access to this data from a class instance.

In this example, the fields `custID`, `name`, and `amount` are now marked `private`, making them invisible outside of the methods in the class itself.

For example:

```
CheckingAccount ca = new CheckingAccount():
ca.amount = 1_000_000_000.00;    // illegal!
```

Public and Private Access Modifiers

- The `public` keyword, applied to fields and methods, allows any class in any package to access the field or method.
- The `private` keyword, applied to fields and methods, allows access only to other methods within the class itself.

```
CheckingAccount chk = new CheckingAccount ();
chk.amount = 200; // Compiler error – amount is a private field
chk.setAmount (200); // OK
```

- The `private` keyword can also be applied to a method to hide an implementation detail.

```
// Called when a withdrawal exceeds the available funds
private void applyOverdraftFee () {
    amount += fee;
}
```

Revisiting Employee

The `Employee` class currently uses `public` access for all of its fields. To encapsulate the data, make the fields `private`.

```
package come.example.model;
public class Employee {
    private int empId;
    private String name;
    private String ssn;
    private double salary;
    //... constructor and methods
}
```

Encapsulation step 1:
Hide the data (fields).

Method Naming: Best Practices

Although the fields are now hidden using `private` access, there are some issues with the current `Employee` class.

- The setter methods (currently `public` access) allow any other class to change the ID, SSN, and salary (up or down).
- The current class does not really represent the operations defined in the original `Employee` class design.
- Two best practices for methods:
  - Hide as many of the implementation details as possible.
  - Name the method in a way that clearly identifies its use or functionality.
- The original model for the `Employee` class had a Change Name and Increase Salary operation.

Choosing Well-Intentioned Methods

Just as fields should clearly define the type of data that they store, methods should clearly identify the operations that they perform. One of the easiest ways to improve the readability of your code (Java code or any other) is to write method names that clearly identify what they do.

`Employee` Class Refined

```
package com.example.domain;
public class Employee {
    // private fields ...
    public Employee () {
    }
    // Remove all of the other setters
    public void setName(String newName) {
        if (newName != null) {
            this.name = newName;
         }
     }

     public void raiseSalary(double increase) {
        this.salary += increase;
     }
}
```

Encapsulation step 2: These method names make sense in the context of an Employee.

The current setter methods in the class allow any class that uses an instance of `Employee` to alter the object's ID, salary, and SSN fields. From a business standpoint, these are not operations you would want on an employee. Once the employee is created, these fields should be immutable (no changes allowed).

The `Employee` model as defined in the slide titled "Encapsulation: Example" had only two operations: one for changing an employee name (as a result of a marriage or divorce) and one for increasing an employee's salary.

To refine the `Employee` class, the first step is to remove the setter methods and create methods that clearly identify their purpose. Here there are two methods, one to change an employee name (`setName`) and the other to increase an employee salary (`raiseSalary`).

Note that the implementation of the `setName` method tests the string parameter passed in to make sure that the string is not a null. The method can do further checking as necessary.

Make Classes as Immutable as Possible

```
package com.example.domain;
public class Employee {
    // private fields ...
    // Create an employee object
    public Employee (int empId, String name,
                     String ssn, double salary) {
        this.empId = empId;
        this.name = name;
        this.ssn = ssn;
        this.salary = salary;
     }

     public void setName(String newName) { ... }

     public void raiseSalary(double increase) { ... }
}
```

Encapsulation step 3: Replace the no-arg constructor with a constructor to set the value of all fields.

Good Practice: Immutability

Finally, because the class no longer has setter methods, you need a way to set the initial value of the fields. The answer is to pass each field value in the construction of the object. By creating a constructor that takes all of the fields as arguments, you can guarantee that an `Employee` instance is fully populated with data *before* it is a valid employee object. This constructor *replaces* the no-arg constructor.

Granted, the user of your class could pass null values, and you need to determine if you want to check for those in your constructor. Strategies for handling those types of situations are discussed in later lessons.

Removing the setter methods and replacing the no-arg constructor also guarantees that an instance of `Employee` has immutable Employee ID and Social Security Number (SSN) fields.

Creating Subclasses

You created a Java class to model the data and operations of an `Employee`. Now suppose you wanted to specialize the data and operations to describe a `Manager`.

```
package com.example.domain;
public class Manager {
    private int empId;
    private String name;
    private String ssn;
    private double salary;
    private String deptName;
    public Manager () {  }
    // access and mutator methods...
}
```

wait a minute...
this code looks very familiar....

Specialization Using Java Subclassing

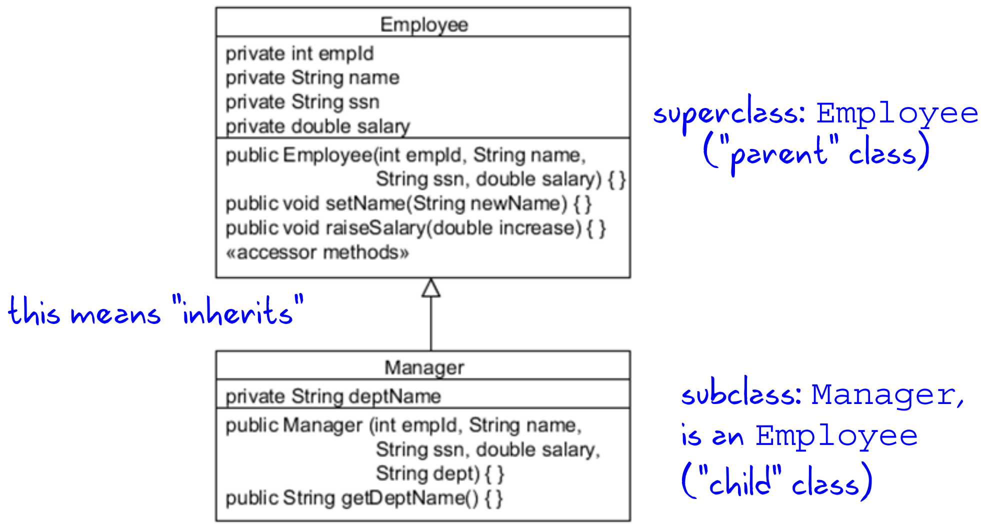
The `Manager` class shown here closely resembles the `Employee` class, but with some specialization. A `Manager` also has a department, with a department name. As a result, there are likely to be additional operations as well.

What this demonstrates is that a `Manager` is an `Employee`—but an `Employee` with additional features.

However, if we were to define Java classes this way, there would be a lot of redundant coding!

Subclassing

In an object-oriented language like Java, subclassing is used to define a new class in terms of an existing one.



A Simple Java Program

When an existing class is subclassed, the new class created is said to inherit the characteristics of the other class. This new class is called the *subclass* and is a specialization of the superclass. All of the non-private fields and methods from the superclass are part of the subclass.

So in this diagram, a `Manager` class gets `empId`, `name`, `SSN`, `salary`, and all of the public methods from `Employee`.

It is important to grasp that although `Manager` specializes `Employee`, a `Manager` is still an `Employee`.

Note: The term *subclass* is a bit of a misnomer. Most people think of the prefix "*sub*-" as meaning "less." However, a Java subclass is the sum of itself and its parent. When you create an instance of a subclass, the resulting in-memory structure contains all codes from the parent class, grandparent class, and so on all the way up the class hierarchy until you reach the class `Object`.

`Manager` Subclass

```
1  package com.example.domain;
2  public class Manager extends Employee {
3      private String deptName;
4      public Manager (int empId, String name,
5                      String ssn, double salary, String dept) {
6          super (empId, name, ssn, salary);
7          this.deptName = dept;
8      }
9
10     public String getDeptName () {
11         return deptName;
12     }
13     // Manager also gets all of Employee's public methods!
14 }
```

The `super` keyword is used to call the constructor of the parent class. It must be the first statement in the constructor.

Java Syntax for Subclassing

The keyword `extends` is used to create a subclass.

The `Manager` class, by extending the `Employee` class, inherits all of the non-private data fields and methods from `Employee`. After all, if a manager is also an employee, then it follows that `Manager` has all of the same attributes and operations of `Employee`.

Note that the `Manager` class declares its own constructor. Constructors are *not* inherited from the parent class. There are additional details about this in the next slide.

The constructor that `Manager` declares in line 4 calls the constructor of its parent class, `Employee`, using the `super` keyword. This sets the value of all of the `Employee` fields: `id`, `name`, `ssn`, and `salary`. `Manager` is a specialization of `Employee`, so constructing a `Manager` requires a department name, which is assigned to the `deptName` field in line 7.

What other methods might you want in a model of Manager? Perhaps you want a method that adds an `Employee` to this `Manager`. You can use an array or a special class called a *collection* to keep track of the employees for whom this manager is responsible. For details about collections, see the lesson titled "Generics and Collections."

Constructors Are Not Inherited

Although a subclass inherits all of the methods and fields from a parent class, it does not inherit constructors. There are two ways to gain a constructor:

- Write your own constructor.
- Use the default constructor.
  - If you do not declare a constructor, a default no-argument constructor is provided for you.
  - If you declare your own constructor, the default constructor is no longer provided.

Constructors in Subclasses

Every subclass inherits the non-private fields and methods from its parent (superclass). However, the subclass does not inherit the constructor from its parent. It must provide a constructor.

The *Java Language Specification* includes the following description:

"Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding."

Using `super` in Constructors

To construct an instance of a subclass, it is often easiest to call the constructor of the parent class.

- In its constructor, `Manager` calls the constructor of `Employee`.

```
super (empId, name, ssn, salary);
```

- The `super` keyword is used to call a parent's constructor.
- It must be the first statement of the constructor.
- If it is not provided, a default call to `super()` is inserted for you.
- The `super` keyword may also be used to invoke a parent's method or to access a parent's (non-private) field.

The `Manager` class declares its own constructor and calls the constructor of the parent class using the `super` keyword.

Note: The `super` call of the parent's constructor must appear first in the constructor.

The `super` keyword can also be used to explicitly call the methods of the parent class or access fields.

Constructing a `Manager` Object

Creating a `Manager` object is the same as creating an `Employee` object:

```
Manager mgr = new Manager (102, "Barbara Jones",
                  "107-99-9078", 109345.67, "Marketing");
```

- All of the `Employee` methods are available to `Manager`:

```
mgr.raiseSalary (10000.00);
```

- The `Manager` class defines a new method to get the Department Name:

```
String dept = mgr.getDeptName();
```


Even though the `Manager.java` file does not contain all of the methods from the `Employee.java` class (explicitly), they are included in the definition of the object. Thus, after you create an instance of a `Manager` object, you can use the methods declared in `Employee`.

You can also call methods that are specific to the `Manager` class as well.

What Is Polymorphism?

The word *polymorphism*, strictly defined, means “many forms.”

```
Employee emp = new Manager();
```

- This assignment is perfectly legal. An employee can be a manager.
- However, the following does not compile:

```
emp.setDeptName ("Marketing"); // compiler error!
```

- The Java compiler recognizes the `emp` variable only as an `Employee` object. Because the `Employee` class does not have a `setDeptName` method, it shows an error.

In object-oriented programming languages such as Java, *polymorphism* is the ability to refer to an object using either its actual form or a parent form.

This is particularly useful when creating a general-purpose business method. For example, you can raise the salary of any `Employee` object (parent or child) by simply passing the object reference to a general-purpose business method that accepts an `Employee` object as an argument.

Overloading Methods

Your design may call for several methods in the same class with the same name but with different arguments.

```
public void print (int i)
public void print (float f)
public void print (String s)
```

- Java permits you to reuse a method name for more than one method.
- Two rules apply to overloaded methods:
  - Argument lists *must* differ.
  - Return types *can* be different.
- Therefore, the following is not legal:

```
public void print (int i)
public String print (int i)
```

ORACLE

You might want to design methods with the same intent (method name), like `print`, to print out several different types. You could design a method for each type:

```
printInt(int i)
printFloat(float f)
printString(String s)
```

But this would be tedious and not very object-oriented. Instead, you can create a reusable method name and just change the argument list. This process is called *overloading*.

With overloading methods, the argument lists must be different—in order, number, or type. And the return types can be different. However, two methods with the same argument list that differ only in return type are not allowed.

Methods Using Variable Arguments

A variation of method overloading is when you need a method that takes any number of arguments of the same type:

```
public class Statistics {
    public float average (int x1, int x2) {}
    public float average (int x1, int x2, int x3) {}
    public float average (int x1, int x2, int x3, int x4) {}
}
```

- These three overloaded methods share the same functionality. It would be nice to collapse these methods into one method.

```
Statistics stats = new Statistics ();
float avg1 = stats.average(100, 200);
float avg2 = stats.average(100, 200, 300);
float avg3 = stats.average(100, 200, 300, 400);
```

ORACLE

Methods with a Variable Number of the Same Type

One case of overloading is when you need to provide a set of overloaded methods that differ in the number of the same type of arguments. For example, suppose you want to have methods to calculate an average. You may want to calculate averages for 2, 3, or 4 (or more) integers.

Each of these methods performs a similar type of computation—the average of the arguments passed in, as in this example:

```
public class Statistics {
    public float average(int x1, int x2) { return (x1 + x2) / 2; }
    public float average(int x1, int x2, int x3) {
        return (x1 + x2 + x3) / 3;
    }
    public float average(int x1, int x2, int x3, int x4) {
        return (x1 + x2 + x3 + x4) / 4;
    }
}
```

Java provides a convenient syntax for collapsing these three methods into just one and providing for any number of arguments.

Methods Using Variable Arguments

- Java provides a feature called *varargs* or *variable arguments*.

The varargs notation treats the `nums` parameter as an array.

```
public class Statistics {
    public float average(int... nums) {
        int sum = 0;
        for (int x : nums) {  // iterate int array nums
            sum += x;
        }
        return ((float) sum / nums.length);
    }
}
```

- Note that the `nums` argument is actually an array object of type `int[]`. This permits the method to iterate over and allow any number of elements.

Using Variable Arguments

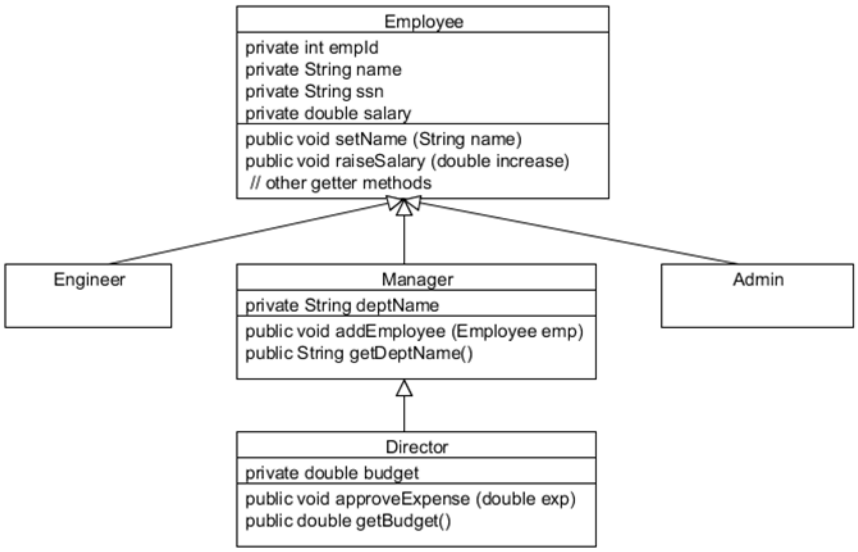
The `average` method shown in the slide takes any number of integer arguments. The notation `(int... nums)` converts the list of arguments passed to the `average` method into an array object of type `int`.

Note: Methods that use varargs can also take no parameters—an invocation of `average()` is legal. You will see varargs as optional parameters in use in the NIO.2 API in the lesson titled "Java File I/O." To account for this, you could rewrite the `average` method in the slide as follows:

```
public float average(int... nums) {
    int sum = 0; float result = 0;
        if (nums.length > 0) {
            for (int x : nums)   // iterate int array nums
                sum += x;
            result = (float) sum / nums.length;
        }
        return (result);
    }
}
```

Single Inheritance

The Java programming language permits a class to extend only one other class. This is called *single inheritance*.



Although Java does not permit more than one class to a subclass, the language does provide features that enable multiple classes to implement the features of other classes. You will see this in the lesson on inheritance.

Single inheritance does not prevent continued refinement and specialization of classes as shown above.

In the diagram shown in the slide, a manager can have employees, and a director has a budget and can approve expenses.

Summary

In this lesson, you should have learned how to:

- Create simple Java classes
- Use encapsulation in Java class design
- Model business problems using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods
- Use variable argument methods

Quiz

Given the diagram in the slide titled "Single Inheritance" and the following Java statements, which statements do *not* compile?

```
Employee e = new Director();
Manager m = new Director();
Admin a = new Admin();
```

a. `e.addEmployee (a);`

b. `m.addEmployee(a);`

c. `m.approveExpense(100000.00);`

d. All of them fail to compile.

Answer: a, c

a. A compiler error because the `Employee` class does not have an `addEmployee` method. This is a part of the `Manager` class.

b. Compiles properly because, although the constructor is creating a `Director`, it is the `Manager` class that the compiler is looking at to determine if there is an `addEmployee` method

c. A compiler error because the `Manager` class does not contain an `approveExpense` method

Quiz

Consider the following classes that do not compile:

```
public class Account {
    private double balance;
    public Account(double balance) { this.balance = balance; }
    //... getter and setter for balance
}
public class Savings extends Account {
    private double interestRate;
    public Savings(double rate) { interestRate = rate; }
}
```

What fix allows these classes to compile?

a. Add a no-arg constructor to `Savings`.
b. Call the `setBalance` method of `Account` from `Savings`.
c. Change the access of `interestRate` to `public`.
d. Replace the constructor in `Savings` with one that calls the constructor of `Account` using `super`.

Answer: d

`Savings` must call the constructor of its parent class (`Account`). To do that, you must replace the current `Savings` constructor with one that includes an initial balance, and calls the Account constructor using super, as in this example:

```
public Savings (double balance, double rate) {
    super(balance);
    interestRate = rate;
}
```

Quiz

Which of the following declarations demonstrates the application of good Java naming conventions?

a. `public class repeat { }`

b. `public void Screencoord (int x, int y){}`

c. `private int XCOORD;`

d.
```
public int calcOffset (int x1, int y1,
                       int x2, int y2) { }
```


Answer: d

a. Uses a lowercase first letter and a verb for a class name. Class names should be nouns with an initial uppercase letter.
b. Is a method name with its first letter uppercase, rather than lower camel case (with the first letter lowercase and the first letter of each name element in uppercase). In addition, `Screencoord` sounds like a noun rather than a verb.
c. Is questionable because it appears to be a constant. It is in uppercase, however, it is not declared final and there is no assigned value.
d. Follows the Java naming convention. It clearly identifies its intent and will calculate the offset between the two coordinate pairs passed as arguments.

Quiz

What changes would you perform to make this class immutable? (Choose all that apply.)

```
public class Stock {
    public String symbol;
    public double price;
    public int shares;
    public double getStockValue() { }
    public void setSymbol(String symbol) { }
    public void setPrice(double price) { }
    public void setShares(int number) { }
}
```

a. Make the fields `symbol`, `shares`, and `price` `private`.
b. Remove `setSymbol`, `setPrice`, and `setShares`.
c. Make the `getStockValue` method `private`.
d. Add a constructor that takes `symbol`, `shares`, and `price` as arguments.

Answer: a, b, d

"Immutable" simply means that the object cannot be changed after it is created. Making the fields `private` prevents access from outside the class. Removing the setter methods prevents changes. Adding the constructor allows the object to be built for the first time with values. The `getStockValue` method does not change any of the fields of the object, so it does not need to be removed.

Practice 3-1 Overview: Creating Subclasses

This practice covers the following topics:

- Applying encapsulation principles to the `Employee` class that you created in the previous practice
- Creating subclasses of `Employee`, including `Manager`, `Engineer`, and Administrative assistant (`Admin`)
- Creating a subclass of `Manager` called `Director`
- Creating a test class with a `main` method to test your new classes

(Optional) Practice 3-2 Overview: Adding a Staff to a Manager

This practice covers the following topics:

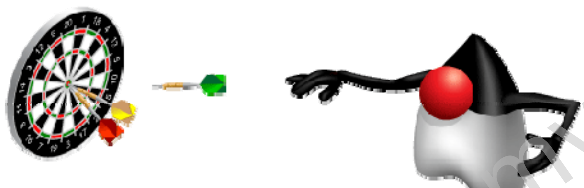
- Creating an array of `Employees` called `staff`
- Creating a method to add an employee to the manager's `staff`
- Creating a method to remove an employee from the manager's `staff`

4 Java Class Design

Objectives

After completing this lesson, you should be able to do the following:

- Use access levels: `private`, `protected`, default, and `public`.
- Override methods
- Overload constructors and other methods appropriately
- Use the `instanceof` operator to compare object types
- Use virtual method invocation
- Use upward and downward casts
- Override methods from the `Object` class to improve the functionality of your class



Using Access Control

You have seen the keywords `public` and `private`. There are four access levels that can be applied to data fields and methods. The following table illustrates access to a field or method marked with the access modifier in the left column.

Modifier (keyword)	Same Class	Same Package	Subclass in Another Package	Universe
`private`	Yes			
`default`	Yes	Yes		
`protected`	Yes	Yes	Yes *	
`public`	Yes	Yes	Yes	Yes

Classes can be default (no modifier) or `public`.

The access modifier keywords shown in this table are `private`, `protected`, and `public`. When a keyword is absent, the *default* access modifier is applied.

The `private` keyword provides the greatest control over access to fields and methods. With `private`, a data field or method can be accessed only within the same Java class.

The `public` keyword provides the greatest access to fields and methods, making them accessible anywhere: in the class, package, subclasses, and any other class.

The `protected` keyword is applied to keep access within the package and subclass. Fields and methods that use `protected` are said to be "subclass-friendly."

***Note:** `protected` access is extended to subclasses that reside in a package different from the class that owns the protected feature. As a result, protected fields or methods are actually more accessible than those marked with default access control. You should use `protected` access when it is appropriate for a class's subclass, but not unrelated classes.

Protected Access Control: Example

```
package demo;
public class Foo {
    protected int result = 20;
    int other = 25;
}
```

subclass-friendly declaration

```
package test;
import demo.Foo;
public class Bar extends Foo {
    private int sum = 10;
    public void reportSum () {
        sum += result;
        sum += other;
    }
}
```

compiler error

In this example, there are two classes in two packages. Class `Foo` is in the package `demo`, and declares a data field called `result` with a `protected` access modifier.

In the class `Bar`, which extends `Foo`, there is a method, `reportSum`, that adds the value of `result` to `sum`. The method then attempts to add the value of `other` to sum. The field `other` is declared using the default modifier, and this generates a compiler error. Why?

Answer: The field `result`, declared as a `protected` field, is available to all subclasses—even those in a different package. The field `other` is declared as using default access and is only available to classes and subclasses declared in the same package.

This example is from the `JavaAccessExample` project.

Field Shadowing: Example

```
package demo;
public class Foo2 {
    protected int result = 20;
}
```

```
package test;
import demo.Foo2;
public class Bar2 extends Foo2 {
    private int sum = 10;
    private int result = 30;
    public void reportSum() {
        sum += result;
    }
}
```

result field shadows the parent's field.

In this example, the class `Foo2` declares the field `result`. However, the class `Bar2` declares its own field `result`. The consequence is that the field `result` from class `Foo2` is shadowed by the field `result` in class `Bar2`. What is `sum` in this example? `sum` is now 40 (10 + 30). Modern IDEs (such as NetBeans) detect shadowing and produce a warning.

Methods with the same name are not shadowed but are overridden. You learn about overriding later in this lesson.

Access Control: Good Practice

A good practice when working with fields is to make fields as inaccessible as possible, and provide clear intent for the use of fields through methods.

```
package demo;
public class Foo3 {
    private int result = 20;
    protected int getResult() { return result; }
}
```

```
package test;
import demo.Foo3;
public class Bar3 extends Foo3 {
    private int sum = 10;
    public void reportSum() {
        sum += getResult();
    }
}
```

ORACLE

A slightly modified version of the example using the `protected` keyword is shown in the slide. If the idea is to limit access of the field result to classes within the package and the subclasses (package-protected), you should make the access explicit by defining a method purposefully written for package and subclass-level access.

Overriding Methods

Consider a requirement to provide a String that represents some details about the `Employee` class fields.

```
public class Employee {
    private int empId;
    private String name;
    // ... other fields and methods
    public String getDetails () {
        return "Employee id:  " + empId +
               " Employee name:" + name;
    }
}
```


Although the `Employee` class has getters to return values for a print statement, it might be nice to have a utility method to get specific details about the employee. Consider a method added to the `Employee` class to print details about the `Employee` object.

In addition to adding fields or methods to a subclass, you can also modify or change the existing behavior of a method of the parent (superclass).

You may want to specialize this method to describe a `Manager` object.

Overriding Methods

In the `Manager` class, by creating a method with the same signature as the method in the `Employee` class, you are *overriding* the `getDetails` method:

```
public class Manager extends Employee {
    private String deptName;
    // ... other fields and methods
    public String getDetails () {
        return super.getDetails () +
                   " Department: " + deptName;
    }
}
```

A subclass can invoke a parent method by using the `super` keyword.

When a method is overridden, it replaces the method in the superclass (parent) class.

This method is called for any `Manager` instance.

A call of the form `super.getDetails()` invokes the `getDetails` method of the parent class.

Note: If, for example, a class declares two public methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method.

Invoking an Overridden Method

- Using the previous examples of `Employee` and `Manager`:

```
Employee e = new Employee (101, "Jim Smith", "011-12-2345",
100_000.00);
Manager m = new Manager (102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
System.out.println (e.getDetails());
System.out.println (m.getDetails());
```

- The correct `getDetails` method of each class is called:

```
Employee id: 101 Employee name: Jim Smith
Employee id: 102 Employee name: Joan Kern Department: Marketing
```


During run time, the Java Virtual Machine invokes the `getDetails` method of the appropriate class. If you comment out the `getDetails` method in the `Manager` class shown in the previous slide, what happens when `m.getDetails()` is invoked?

Answer: Recall that methods are inherited from the parent class. So, at run time, the `getDetails` method of the parent class (`Employee`) is executed.

Virtual Method Invocation

- What happens if you have the following?

```
Employee e = new Manager (102, "Joan Kern", "012-23-4567",
110_450.54, "Marketing");
System.out.println (e.getDetails());
```

- During execution, the object's runtime type is determined to be a `Manager` object:

```
Employee id: 102 Employee name: Joan Kern Department: Marketing
```

- The compiler is satisfied because the `Employee` class has a `getDetails` method, and at run time the method that is executed is referenced from a `Manager` object.
- This is an aspect of polymorphism called *virtual method invocation*.

Compiler Versus Runtime Behavior

The important thing to remember is the difference between the compiler (which checks that each method and field is accessible based on the strict definition of the class) and the behavior associated with an object determined at run time.

This distinction is an important and powerful aspect of polymorphism: The behavior of an object is determined by its runtime reference.

Because the object you created was a `Manager` object, at run time, when the `getDetails` method was invoked, the run time reference is to the `getDetails` method of a `Manager` class, even though the variable `e` is of the type `Employee`.

This behavior is referred to as *virtual method invocation*.

Note: If you are a C++ programmer, you get this behavior in C++ only if you mark the method using the C++ keyword `virtual`.

Accessibility of Overridden Methods

An overriding method must provide at least as much access as the overridden method in the parent class.

```
public class Employee {
    //... other fields and methods
    public String getDetails() { ... }
}
```

```
public class Manager extends Employee {
    //... other fields and methods
    private String getDetails() { //... }  // Compile time error
}
```


To override a method, the name and the order of arguments must be identical.

When a method is a subclass overrides a method in the parent class, it must provide the same or greater access than the parent class. For example, if the parent method getDetails() on the slide was protected, then the overriding method getDetails() in the subclass must be protected or public.

Applying Polymorphism

Suppose that you are asked to create a new class that calculates a stock grant for employees based on their salary and their role (manager, engineer, or admin):

```
public class EmployeeStockPlan {
   public int grantStock (Manager m) {
      // perform a calculation for a Manager
   }
   public int grantStock (Engineer e) {
      // perform a calculation for an Engineer
   }
   public int grantStock (Admin a) {
      // perform a calculation for an Admin
   }
   //... one method per employee type
}
```

not very object-oriented!

ORACLE

Design Problem

What is the problem in the example in the slide? Each method performs the calculation based on the type of employee passed in, and returns the number of shares.

Consider what happens if you add two or three more employee types. You would need to add three additional methods, and possibly replicate code depending upon the business logic required to compute shares.

Clearly, this is not a good way to treat this problem. Although the code will work, this is not easy to read and is likely to create much duplicate code.

Applying Polymorphism

A good practice is to pass parameters and write methods that use the most generic form of your object as possible.

```
public class EmployeeStockPlan {
   public int grantStock (Employee e) {
      // perform a calculation based on Employee data
   }
}
```

```
// In the application class
EmployeeStockPlan esp = new EmployeeStockPlan ():
Manager m = new Manager ();
int stocksGranted = grantStock (m);
...
```


Use the Most Generic Form

A good practice is to design and write methods that take the most generic form of your object as possible.

In this case, `Employee` is a good base class to start from.

Applying Polymorphism

Adding a method to `Employee` allows `EmployeeStockPlan` to use polymorphism to calculate stock.

```
public class Employee {
    protected int calculateStock() { return 10; }
}
```

```
public class Manager extends Employee {
    public int calculateStock() { return 20; }
}
```

```
public class EmployeeStockPlan {
    private float stockMultiplier;   // Calculated elsewhere
    public int grantStock (Employee e) {
        return (int)(stockMultipier * e.calculateStock());
    }
}
```


In this modified `EmployeeStockPlan`, the `grantStock` method calls a method defined by the base class, `Employee`, that returns a base stock count. The `EmployeeStockPlan` class uses a multiplier to determine how many stocks to grant. A class may override the `calculateStock` method (as the `Manager` class does here) based on business policy.

In this approach, regardless of the number of different Employee types (all extending the `Employee` object) the `EmployeeStockPlan` class will always function.

Using the `instanceof` Keyword

The Java language provides the `instanceof` keyword to determine an object's class type at run time.

```
public class EmployeeRequisition {
   public boolean canHireEmployee(Employee e) {
       if (e instanceof Manager) {
           return true;
       } else {
           return false;
       }
   }
}
```


In this example, a class, `EmployeeRequisition` has a method that uses the `instanceof` keyword to determine if the object can open an requisition for an employee. Per the business policy, only Managers and above can open a requisition for a new employee.

Casting Object References

In order to access a method in a subclass passed as an generic argument, you must cast the reference to the class that will satisfy the compiler:

```
public void modifyDeptForManager (Employee e, String dept) {
   if (e instanceof Manager) {
      Manager m = (Manager) e;
      m.setDeptName (dept);
   }
}
```

Without the cast to `Manager`, the `setDeptName` method would not compile.

Although a generic superclass reference is useful for passing objects around, you may need to use a method from the subclass.

In the slide, for example, you need the `setDeptName` method of the `Manager` class. To satisfy the compiler, you can cast a reference from the generic superclass to the specific class.

However, there are rules for casting references. You see these in the next slide.

Note: The `instanceof` operator shown on the slide is not required by the compiler before the cast, however, without checking the object's type, if a non-`Manager` object type is passed to the `modifyDeptForManager` method, an exception (`ClassCastException`) will be thrown at runtime.

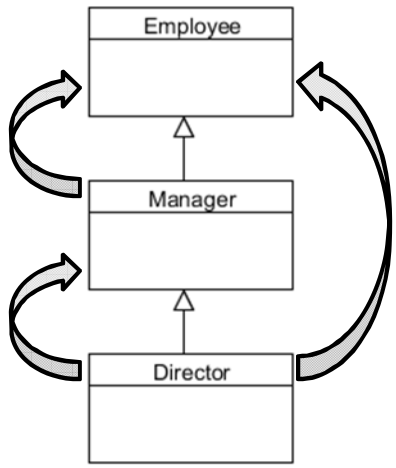
Casting Rules

Upward casts are always permitted and do not require a cast operator.

```
Director d = new Director();
Manager m = new Manager();
```

```
Employee e = m; // OK
```

```
Manager m = d; // OK
```



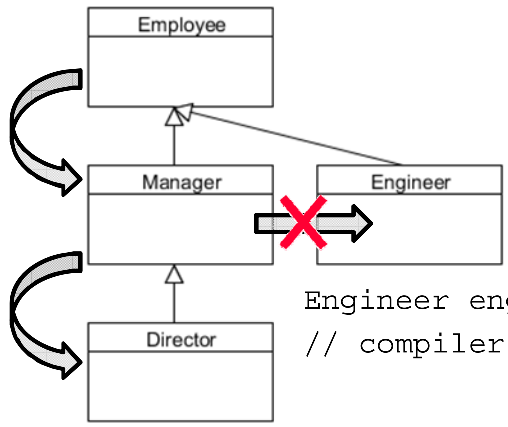

```
Employee e = d; // OK
```

Casting Rules

For downward casts, the compiler must be satisfied that the cast is at least possible.

```
Employee e = new Manager();
Manager m = new Manager();
```

```
Manager m = (Manager)e;
// Would also work if
// e was a Director obj
```




```
Director d = (Director)m;
// fails at run time
```

```
Engineer eng = (Engineer)m;
// compiler error
```


Downward Casts

With a downward cast, the compiler simply determines if the cast is possible; if the cast down is to a subclass, then it is possible that the cast will succeed.

Note that at run time the cast results in a `java.lang.ClassCastException` if the object reference is of a superclass and not of the class type or a subclass.

The cast of the variable `e` to a Manager reference `m` satisfies the compiler, because `Manager` and `Employee` are in the same class hierarchy, so the cast will possibly succeed. This cast also works at run time, because it turns out that the variable `e` is actually a `Manager` object. This cast would also work at run time if `e` pointed to an instance of a `Director` object.

The cast of the variable `m` to a `Director` instance satisfies the compiler, but because `m` is actually a `Manager` instance, this cast fails at run time with a `ClassCastException`.

Finally, any cast will fail that is outside the class hierarchy, such as the cast from a `Manager` instance to an `Engineer`. A `Manager` and an `Engineer` are both employees, but a `Manager` is not an `Engineer`.

Overriding Object methods

One of the advantages of single inheritance is that every class has a parent object by default. The root class of every Java class is `java.lang.Object`.

- You do not have to declare that your class extends `Object`. The compiler does that for you.

```
public class Employee { //... }
```

is equivalent to:

```
public class Employee extends Object { //... }
```

- The root class contains several non-final methods, but there are three that are important to consider overriding:
  - `toString`, `equals`, and `hashCode`

Best Practice: Overload Object Methods

The `java.lang.Object` class is the root class of all classes in the Java programming language. All classes will subclass `Object` by default.

`Object` defines several non-final methods that are designed to be overridden by your class. These are `equals`, `hashCode`, `toString`, `clone`, and `finalize`. Of these, there are three methods that you should consider overriding.

`Object` `toString` Method

The `toString` method is called to return the string value of an object. For example, using the method `println`:

```
Employee e = new Employee (101, "Jim Kern", ...)
System.out.println (e);
```

- String concatenation operations also invoke `toString`:

```
String s = "Employee: " + e;
```

- You can use `toString` to provide instance information:

```
public String toString () {
    return "Employee id:  " + empId + "\n"
           "Employee name:" + name;
}
```

- This is a better approach to getting details about your class than creating your own `getDetails` method.

The `println` method is overloaded with a number of parameter types. When you invoke `System.out.println(e);` the method that takes an `Object` parameter is matched and invoked. This method in turn invokes the `toString()` method on the object instance.

Note: Sometimes you may want to be able to print out the name of the class that is executing a method. The `getClass()` method is an `Object` method used to return the `Class` object instance, and the `getName()` method provides the fully qualified name of the runtime class.

```
getClass().getName(); // returns the name of this class instance
```

These methods are in the `Object` class.

`Object equals` Method

The `Object equals` method compares only object references.

- If there are two objects x and y in any class, x is equal to y if and only if x and y refer to the same object.
- Example:

```
Employee x = new Employee (1,"Sue","111-11-1111",10.0);
Employee y = x;
x.equals (y); // true
Employee z = new Employee (1,"Sue","111-11-1111",10.0);
x.equals (z); // false!
```

- Because what we really want is to test the contents of the Employee object, we need to override the `equals` method:

```
public boolean equals (Object o) { ... }
```


The `equals` method of `Object` determines (by default) only if the values of two object references point to the same object. Basically, the test in the `Object` class is simply as follows:

If x == y, return true.

For an object (like the `Employee` object) that contains values, this comparison is not sufficient, particularly if we want to make sure there is one and only one employee with a particular ID.

Overriding `equals` in `Employee`

An example of overriding the `equals` method in the `Employee` class compares every field for equality:

```
public boolean equals (Object o) {
    boolean result = false;
    if ((o != null) && (o instanceof Employee)) {
        Employee e = (Employee)o;
        if ((e.empId == this.empId) &&
            (e.name.equals(this.name)) &&
            (e.ssn.equals(this.ssn)) &&
            (e.salary == this.salary)) {
               result = true;
         }
     }
     return result;
}
```

ORACLE

This simple `equals` test first tests to make sure that the object passed in is not null, and then tests to make sure that it is an instance of an `Employee` class (all subclasses are also employees, so this works). Then the `Object` is cast to `Employee`, and each field in `Employee` is checked for equality.

Note: For `String` types, you should use the `equals` method to test the strings character by character for equality.

Overriding `Object` `hashCode`

The general contract for `Object` states that if two objects are considered equal (using the `equals` method), then integer hashcode returned for the two objects should also be equal.

```
// Generated by NetBeans
public int hashCode() {
     int hash = 7;
     hash = 83 * hash + this.empId;
     hash = 83 * hash + Objects.hashCode(this.name);
     hash = 83 * hash + Objects.hashCode(this.ssn);
     hash = 83 * hash +
                  (int)(Double.doubleToLongBits(this.salary) ^
                  (Double.doubleToLongBits(this.salary) >>> 32));
      return hash;
 }
```


Overriding `hashCode`

The Java documentation for the `Object` class states:

"... It is generally necessary to override the `hashCode` method whenever this method [`equals`] is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes."

The `hashCode` method is used in conjunction with the `equals` method in hash-based collections, such as `HashMap`, `HashSet`, and `Hashtable`.

This method is easy to get wrong, so you need to be careful. The good news is that IDEs such as NetBeans can generate `hashCode` for you.

To create your own hash function, the following will help approximate a reasonable hash value for equal and unequal instances:

1) Start with a non-zero integer constant. Prime numbers result in fewer hashcode collisions.
2) For each field used in the `equals` method, compute an `int` hash code for the field. Notice that for the `Strings`, you can use the `hashCode` of the `String`.
3) Add the computed hash codes together.
4) Return the result.

Summary

In this lesson, you should have learned how to:

- Use access levels: `private`, `protected`, default, and `public`
- Override methods
- Overload constructors and other methods appropriately
- Use the `instanceof` operator to compare object types
- Use virtual method invocation
- Use upward and downward casts
- Override methods from the `Object` class to improve the functionality of your class

Quiz

Suppose that you have an `Account` class with a `withdraw()` method, and a `Checking` class that extends `Account` that declares its own `withdraw()` method. What is the result of the following code fragment?

```
1  Account acct = new Checking();
2  acct.withdraw(100);
```

a. The compiler complains about line 1.
b. The compiler complains about line 2.
c. Runtime error: incompatible assignment (line 1)
d. The `Account.withdraw()` method executes.
e. The `Checking.withdraw()` method executes.

Answer: e

Because the `Checking` class extends `Account`, the `withdraw` method declared in `Checking` overrides the `withdraw` method in `Account`. At run time, the method for the object (`Checking`) is executed.

Quiz

Suppose that you have an `Account` class and a `Checking` class that extends `Account`. The body of the `if` statement in line 2 will execute.

```
1   Account acct = new Checking();
2   if (acct instanceof Checking) { // will this block run? }
```

a. True

b. False

Answer: a

Actually, `acct` is also an `instanceof` the `Account` class.

Quiz

Suppose that you have an `Account` class and a `Checking` class that extends `Account`. You also have a `Savings` class that extends `Account`. What is the result of the following code?

```
1  Account acct1 = new Checking();
2  Account acct2 = new Savings();
3  Savings acct3 = (Savings)acct1;
```

a. `acct3` contains the reference to `acct1`.
b. A runtime `ClassCastException` occurs.
c. The compiler complains about line 2.
d. The compiler complains about the cast in line 3.

Answer: b

The compiler will assume that it is possible to cast an `Account` type object to another `Account`. Because `Savings` extends from `Account`, this looks like a typical downward cast. However, at run time, the true type of the object is determined, and you cannot cast between children.

Quiz

```
1 package com.bank;
2 public class Account {
3     double balance;
4 }
```

```
10 package com.bank.type;
11 import com.bank.Account;
12 public class Savings extends Account {
13     private double interest;
14     Account acct = new Account();
15     public double getBalance (){ return (interest + balance); }
16 }
```

What change would make this code compile?

a. Make `balance private` in line 3.

b. Make `balance protected` in line 3.

c. Replace `balance` with `acct.balance` in line 15.

d. Replace `balance` with `Account.balance` in line 15.

Answer: b

Practice 4-1 Overview: Overriding Methods and Applying Polymorphism

This practice covers the following topics:

- Modifying the `Employee`, `Manager`, and `Director` classes; overriding the `toString()` method
- Creating an `EmployeeStockPlan` class with a grant stock method that uses the `instanceof` keyword

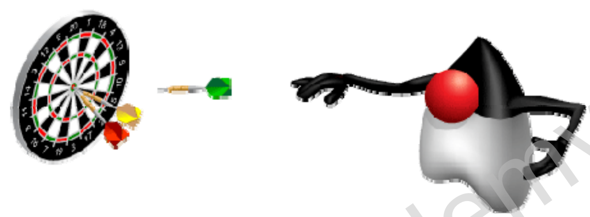
5

Advanced Class Design

Objectives

After completing this lesson, you should be able to do the following:

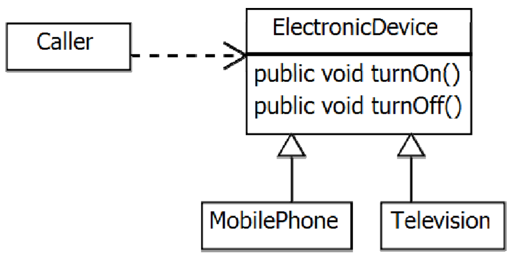
- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Model business problems by using the `static` and `final` keywords
- Implement the singleton design pattern
- Distinguish between top-level and nested classes



Modeling Business Problems with Classes

Inheritance (or subclassing) is an essential feature of the Java programming language. Inheritance provides code reuse through:

- Method inheritance: Subclasses avoid code duplication by inheriting method implementations.
- Generalization: Code that is designed to rely on the most generic type possible is easier to maintain.



Class Inheritance Diagram

Class Inheritance

When designing an object-oriented solution, you should attempt to avoid code duplication. One technique to avoid duplication is to create library methods and classes. Libraries function as a central point to contain often reused code. Another technique to avoid code duplication is to use class inheritance. When there is a shared base type identified between two classes, any shared code may be placed in a parent class.

When possible, use object references of the most generic base type possible. In Java, generalization and specialization enable reuse through method inheritance and virtual method invocation (VMI). VMI, sometimes called “late-binding,” enables a caller to dynamically call a method as long as the method has been declared in a generic base type.

Enabling Generalization

Coding to a common base type allows for the introduction of new subclasses with little or no modification of any code that depends on the more generic base type.

```
ElectronicDevice dev = new Television();
dev.turnOn(); // all ElectronicDevices can be turned on
```

Always use the most generic reference type possible.

Coding for Generalization

Always use the most generic reference type possible. Java IDEs may contain refactoring tools that assist in changing existing references to a more generic base type.

Identifying the Need for Abstract Classes

Subclasses may not need to inherit a method implementation if the method is specialized.

```
public class Television extends ElectronicDevice {

    public void turnOn() {
        changeChannel(1);
        initializeScreen();
    }
    public void turnOff() {}

    public void changeChannel(int channel) {}
    public void initializeScreen() {}

}
```


Method Implementations

When sibling classes have a common method, it is typically placed in a parent class. Under some circumstances, however, the parent class's implementation will always need to be overridden with a specialized implementation.

In these cases, inclusion of the method in a parent class has both advantages and disadvantages. It allows the use of generic reference types, but developers can easily forget to supply the specialized implementation in the subclasses.

Defining Abstract Classes

A class can be declared as abstract by using the `abstract` class-level modifier.

```
public abstract class ElectronicDevice { }
```

- An abstract class can be subclassed.

```
public class Television extends ElectronicDevice { }
```

- An abstract class cannot be instantiated.

```
ElectronicDevice dev = new ElectronicDevice(); // error
```


Declaring a class as abstract prevents any instances of that class from being created. It is a compile-time error to instantiate an abstract class. An abstract class will typically be extended by a child class and may be used as a reference type.

Defining Abstract Methods

A method can be declared as abstract by using the `abstract` method-level modifier.

```
public abstract class ElectronicDevice {

   public abstract void turnOn();
   public abstract void turnOff();
}
```

No braces

An abstract method:

- Cannot have a method body
- Must be declared in an abstract class
- Is overridden in subclasses

Inheriting Abstract Methods

When a child class inherits an abstract method, it is inheriting a method signature but no implementation. For this reason, no braces are allowed when defining an abstract method. An abstract method is a way to guarantee that any child class will contain a method with a matching signature.

Validating Abstract Classes

The following additional rules apply when you use abstract classes and methods:

- An abstract class may have any number of abstract and non-abstract methods.
- When inheriting from an abstract class, you must do either of the following:
  - Declare the child class as abstract.
  - Override all abstract methods inherited from the parent class. Failure to do so will result in a compile-time error.

```
error: Television is not abstract and does not override
abstract method turnOn() in ElectronicDevice
```


Making Use of Abstract Classes

While it is possible to avoid implementing an abstract method by declaring child classes as abstract, this only serves to delay the inevitable. Applications require non-abstract methods to create objects. Use abstract methods to outline functionality required in child classes.

Quiz

To compile successfully, an abstract method must not have:

a. A return value
b. A method implementation
c. Method parameters
d. `private` access

ORACLE

Answer: b, d

`static` Keyword

The `static` modifier is used to declare fields and methods as class-level resources. Static class members:

- Can be used without object instances
- Are used when a problem is best solved without objects
- Are used when objects of the same type need to share fields
- Should *not* be used to bypass the object-oriented features of Java unless there is a good reason

Java: Object-oriented by Design

The Java programming language was designed as an object-oriented language, unlike languages like Objective-C and C++, which inherited the procedural design of C. When developing in Java, you should always attempt to design an object-oriented solution.

Static Methods

Static methods are methods that can be called even if the class they are declared in has not been instantiated. Static methods:

- Are called class methods
- Are useful for APIs that are not object oriented.
  - `java.lang.Math` contains many static methods
- Are commonly used in place of constructors to perform tasks related to object initialization
- Cannot access non-static members within the same class
- Can be hidden in subclasses but not overridden
  - No virtual method invocation

Factory Methods

In place of directly invoking constructors, you will often use static methods to retrieve object references. Unless something unexpected happens, a new object is created whenever a constructor is called. A static factory method could maintain a cache of objects for reuse or create new instances if the cache was depleted. A factory method may also produce an object that subclasses the method's return type.

Example:

```
NumberFormat nf = NumberFormat.getInstance();
```

Implementing Static Methods

```
public class StaticErrorClass {
    private int x;

    public static void staticMethod() {
        x = 1; // compile error
        instanceMethod(); // compile error
    }

    public void instanceMethod() {
        x = 2;
    }
}
```


Static Method Limitations

Static methods can be used before any instances of their enclosing class have been created. Chronologically speaking, this means that in a running Java Virtual Machine, there may not be any occurrences of the containing classes instance variables. Static methods can never access their enclosing classe's instance variables or call their non-static methods.

Calling Static Methods

```
double d = Math.random();
StaticUtilityClass.printMessage();
StaticUtilityClass uc = new StaticUtilityClass();
uc.printMessage(); // works but misleading
sameClassMethod();
```

When calling static methods, you should:

- Qualify the location of the method with a class name if the method is located in a different class than the caller
  - Not required for methods within the same class
- Avoid using an object reference to call a static method

Static Variables

Static variables are variables that can be accessed even if the class they are declared in has not been instantiated. Static variables are:

- Called class variables
- Limited to a single copy per JVM
- Useful for containing shared data
  - Static methods store data in static variables.
  - All object instances share a single copy of any static variables.
- Initialized when the containing class is first loaded

Class Loading

Application developer-supplied classes are typically loaded on demand (first use). Static variables are initialized when their enclosing class is loaded. An attempt to access a static class member can trigger the loading of a class.

Defining Static Variables

```
public class StaticCounter {
    private static int counter = 0;

    public StaticCounter() {
        counter++;
    }

    public static int getCount() {
        return counter;
    }
}
```

Only one copy in memory

Persisting Static Variables

Many technologies that are used to persist application state in Java only save instance variables. Maintaining a single object that keeps track of "shared" state may be used as an alternative to static variables.

Using Static Variables

```
double p = Math.PI;
```

```
new StaticCounter();
new StaticCounter();
System.out.println("count: " + StaticCounter.getCount());
```

When accessing static variables, you should:

- Qualify the location of the variable with a class name if the variable is located in a different class than the caller
  - Not required for variables within the same class
- Avoid using an object reference to access a static variable

Object References to Static Members

Just as using object references to static methods should be avoided, you should also avoid using object references to access static variables. If all the members of a class are static, consider using a private constructor to prevent object instantiation.

Static Imports

A static import statement makes the static members of a class available under their simple name.

- Given either of the following lines:

```
import static java.lang.Math.random;
import static java.lang.Math.*;
```

- Calling the `Math.random()` method can be written as:

```
public class StaticImport {
    public static void main(String[] args) {
        double d = random();
    }
}
```


Overusing static import can negatively affect the readability of your code. Avoid adding multiple static imports to a class.

Quiz

The number of instances of a static variable is related to the number of objects that have been created.

a. True

b. False

Answer: b

Final Methods

A method can be declared `final`. Final methods may not be overridden.

```
public class MethodParentClass {
    public final void printMessage() {
        System.out.println("This is a final method");
    }
}
```

```
public class MethodChildClass extends MethodParentClass {
    // compile-time error
    public void printMessage() {
        System.out.println("Cannot override method");
    }
}
```

ORACLE

Performance Myths

There is little to no performance benefit when you declare a method as final. Methods should be declared as final only to disable method overriding.

Final Classes

A class can be declared `final`. Final classes may not be extended.

```
public final class FinalParentClass { }
```

```
// compile-time error
public class ChildClass extends FinalParentClass { }
```

Final Variables

The `final` modifier can be applied to variables. Final variables may not change their values after they are initialized. Final variables can be:

- Class fields
  - Final fields with compile-time constant expressions are constant variables.
  - Static can be combined with final to create an always-available, never-changing variable.
- Method parameters
- Local variables

Note: Final references must always reference the same object, but the contents of that object may be modified.

Benefits and Drawbacks of Final Variables

Bug Prevention

Final variables can never have their values modified after they are initialized. This behavior functions as a bug-prevention mechanism.

Thread Safety

The immutable nature of final variables eliminates any of the concerns that come with concurrent access by multiple threads.

Final Reference to Objects

A `final` object reference only serves to prevent a reference from pointing to another object. If you are designing immutable objects, you must prevent the object's fields from being modified. Final references also prevent you from assigning a value of `null` to the reference. Maintaining an object's references prevents that object from being available for garbage collection.

Declaring Final Variables

```
public class VariableExampleClass {
    private final int field;
    private final int forgottenField;
    private final Date date = new Date();
    public static final int JAVA_CONSTANT = 10;

    public VariableExampleClass() {
        field = 100;
        // compile-time error – forgottenField
        // not initialized
    }

    public void changeValues(final int param) {
        param = 1; // compile-time error
        date.setTime(0); // allowed
        date = new Date(); // compile-time error
        final int localVar;
        localVar = 42;
        localVar = 43; // compile-time error
    }
}
```


Final Fields

Initializing

Final fields (instance variables) must be either of the following:

- Assigned a value when declared
- Assigned a value in every constructor

Static and Final Combined

A field that is both static and final is considered a constant. By convention, constant fields use identifiers consisting of only uppercase letters and underscores.

Quiz

A final field (instance variable) can be assigned a value either when declared or in all constructors.

a. True
b. False

Answer: a

When to Avoid Constants

`public static final` variables can be very useful, but there is a particular usage pattern you should avoid. Constants may provide a false sense of input validation or value range checking.

- Consider a method that should receive only one of three possible values:

```
Computer comp = new Computer();
comp.setState(Computer.POWER_SUSPEND);
```

This is an `int` constant that equals `2`.

- The following lines of code still compile:

```
Computer comp = new Computer();
comp.setState(42);
```


Runtime Range Checking

In the example in the slide, you must perform a runtime range check when using an `int` to represent state. Within the `setState` method, an `if` statement can be used to validate that only the values `0`, `1`, or `2` are accepted. This type of check is performed every time the `setState` method is called, resulting in additional overhead.

Typesafe Enumerations

Java 5 added a typesafe enum to the language. Enums:

- Are created using a variation of a Java class
- Provide a compile-time range check

```
public enum PowerState {
    OFF,
    ON,
    SUSPEND;
}
```

These are references to the only three `PowerState` objects that can exist.

An enum can be used in the following way:

```
Computer comp = new Computer();
comp.setState(PowerState.SUSPEND);
```

This method takes a `PowerState` reference .

ORACLE

Compile-Time Range Checking

In the example in the slide, the compiler performs a compile-time check to ensure that only valid `PowerState` instances are passed to the `setState` method. No range checking overhead is incurred at run time.

Enum Usage

Enum references can be statically imported.

```
import static com.example.PowerState.*;

public class Computer extends ElectronicDevice {
    private PowerState powerState = OFF;
    //...
}
```

PowerState.OFF

Enums can be used as the expression in a switch statement.

```
public void setState(PowerState state) {
    switch(state) {
        case OFF:
        //...
    }
}
```


Note: When an `enum` is used in a `switch` statement, only the `enum` constants can be used as labels for the case statements.

Complex Enums

Enums can have fields, methods, and private constructors.

```
public enum PowerState {
    OFF("The power is off"),
    ON("The usage power is high"),
    SUSPEND("The power usage is low");

    private String description;
    private PowerState(String d) {
        description = d;
    }
    public String getDescription() {
        return description;
    }
}
```

Call a `PowerState` constructor to initialize the `public static final OFF` reference.

The constructor may not be `public` or `protected`.

Enum Constructors

You may not instantiate an enum instance with `new`.

Quiz

An enum can have a constructor with the following access levels:

a. public
b. protected
c. Default (no declared access level)
d. private

Answer: c, d

Design Patterns

Design patterns are:

- Reusable solutions to common software development problems
- Documented in pattern catalogs
  - *Design Patterns: Elements of Reusable Object-Oriented Software*, written by Erich Gamma et al. (the "Gang of Four")
- A vocabulary used to discuss design

Design Pattern Catalogs

Pattern catalogs are available for many programming languages. Most of the traditional design patterns apply to any object-oriented programming language. One of the most popular books, *Design Patterns: Elements of Reusable Object-Oriented Software*, uses a combination of C++, Smalltalk, and diagrams to show possible pattern implementations. Many Java developers still reference this book because the concepts translate to any object-oriented language.

You learn more about design patterns and other Java best practices in the *Java Design Patterns* course.

Singleton Pattern

The singleton design pattern details a class implementation that can be instantiated only once.

```
public class SingletonClass {
1   private static final SingletonClass instance =
            new SingletonClass();

2   private SingletonClass() {}

    public static SingletonClass getInstance() {
3       return instance;
    }
}
```


Implementing the Singleton Pattern

The singleton design pattern is one of the creational design patterns that are categorized in *Design Patterns: Elements of Reusable Object-Oriented Software*.

To implement the singleton design pattern:

1. Use a static reference to point to the single instance. Making the reference final ensures that it will never reference a different instance.
2. Add a single private constructor to the singleton class. The private modifier allows only "same class" access, which prohibits any attempts to instantiate the singleton class except for the attempt in step 1.
3. A public factory method returns a copy of the singleton reference. This method is declared static to access the static field declared in step 1. Step 1 could use a public variable, eliminating the need for the factory method. Factory methods provide greater flexibility (for example, implementing a per-thread singleton solution) and are typically used in most singleton implementations.

To obtain a singleton reference, call the `getInstance` method:

```
SingletonClass ref = SingletonClass.getInstance();
```

Nested Classes

A nested class is a class declared within the body of another class. Nested classes:

- Have multiple categories
  - Inner classes
    - Member classes
    - Local classes
    - Anonymous classes
  - Static nested classes
- Are commonly used in applications with GUI elements
- Can limit utilization of a "helper class" to the enclosing top-level class

Reasons to Use Nested Classes

The following information is taken from http://download.oracle.com/javase/tutorial/java/javaOO/nested.html.

Logical Grouping of Classes

If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased Encapsulation

Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More Readable, Maintainable Code

Nesting small classes within top-level classes places the code closer to where it is used.

Inner Class: Example

```
public class Car {
    private boolean running = false;
    private Engine engine = new Engine();

    private class Engine {
        public void start() {
            running = true;
        }
    }

    public void start() {
        engine.start();
    }
}
```


Inner Classes Versus Static Nested Classes

An inner class is considered part of the outer class and inherits access to all the private members of the outer class. The example in the slide shows an inner class, which is a member class. Inner classes can also be declared inside a method block (local classes).

A static nested class is not an inner class, but its declaration appears similar with an additional `static` modifier on the nested class. Static nested classes can be instantiated before the enclosing outer class and, therefore, are denied access to all non-static members of the enclosing class.

Anonymous Inner Classes

An anonymous class is used to define a class with no name.

```
public class AnonymousExampleClass {
    public Object o = new Object() {
        @Override
        public String toString() {
            return "In an anonymous class method";
        }
    };
}
```


A Class with No Name

In the example in the slide, the `java.lang.Object` class is being subclassed, and it is that subclass that is being instantiated. When you compile an application with anonymous classes, a separate class file following a naming convention of `Outer$1.class` will be generated, where `1` is the index number of anonymous classes in an enclosing class and `Outer` is the name of the enclosing class.

Anonymous inner classes can also be local classes.

Quiz

Which of the following nested class types are inner classes?

a. anonymous
b. local
c. static
d. member

Answer: a, b, d

Summary

In this lesson, you should have learned how to:

- Design general-purpose base classes by using abstract classes
- Construct abstract Java classes and subclasses
- Model business problems by using the `static` and `final` keywords
- Implement the singleton design pattern
- Distinguish between top-level and nested classes

Practice 5-1 Overview: Applying the Abstract Keyword

This practice covers the following topics:

- Identifying potential problems that can be solved using abstract classes
- Refactoring an existing Java application to use abstract classes and methods

Practice 5-2 Overview: Applying the Singleton Design Pattern

This practice covers using the static and final keywords and refactoring an existing application to implement the singleton design pattern.

Practice 5-3 Overview: (Optional) Using Java Enumerations

This practice covers taking an existing application and refactoring the code to use an enum.

(Optional) Practice 5-4 Overview: Recognizing Nested Classes

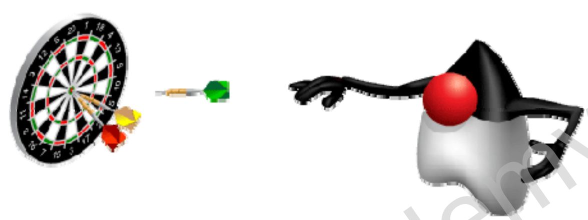
This practice covers analyzing an existing Java application and identifying the number and types of classes present.

Inheritance with Java Interfaces

Objectives

After completing this lesson, you should be able to do the following:

- Model business problems by using interfaces
- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Refactor code to implement the DAO pattern



Implementation Substitution

The ability to outline abstract types is a powerful feature of Java. Abstraction enables:

- Ease of maintenance
  - Classes with logic errors can be substituted with new and improved classes.
- Implementation substitution
  - The `java.sql` package outlines the methods used by developers to communicate with databases, but the implementation is vendor-specific.
- Division of labor
  - Outlining the business API needed by an application's UI allows the UI and the business logic to be developed in tandem.

Abstraction

You just learned how to define abstract types by using classes. There are two ways to define type abstraction in Java: abstract classes and interfaces. By writing code to reference abstract types, you no longer depend on specific implementing classes. Defining these abstract types may seem like extra work in the beginning but can reduce refactoring at a later time if used appropriately.

Java Interfaces

Java interfaces are used to define abstract types. Interfaces:

- Are similar to abstract classes containing only public abstract methods
- Outline methods that must be implemented by a class
  - Methods must not have an implementation {braces}.
- Can contain constant fields
- Can be used as a reference type
- Are an essential component of many design patterns

In Java, an interface outlines a contract for a class. The contract outlined by an interface mandates the methods that must be implemented in a class. Classes implementing the contract must fulfill the entire contract or be declared `abstract`.

Developing Java Interfaces

Public, top-level interfaces are declared in their own `.java` file. You implement interfaces instead of extending them.

```
public interface ElectronicDevice {
    public void turnOn();
    public void turnOff();
}
```

```
public class Television implements ElectronicDevice {
    public void turnOn() { }
    public void turnOff() { }
    public void changeChannel(int channel) {}
    private void initializeScreen() {}
}
```


Rules for Interfaces

Access Modifiers

All methods in an interface are `public`, even if you forget to declare them as `public`. You may not declare methods as `private` or `protected` in an interface. The contract that an interface outlines is a public API that must be provided by a class.

Abstract Modifier

Because all methods are implicitly `abstract`, it is redundant (but allowed) to declare a method `abstract`. Because all interface methods are abstract, you may not provide any method implementation, not even an empty set of braces.

Implements and Extends

A class can extend one parent class and then implement a comma-separated list of interfaces.

Constant Fields

Interfaces can have constant fields.

```
public interface ElectronicDevice {
    public static final String WARNING =
        "Do not open, shock hazard";
    public void turnOn();
    public void turnOff();
}
```


Only constant fields are permitted in an interface. When you declare a field in an interface, it is implicitly `public`, `static`, and `final`. You may redundantly specify these modifiers. Avoid grouping all the constant values for an application in a single interface; good design distributes the constant values of an application across many classes and interfaces. Creating monolithic classes or interfaces that contain large groupings of unrelated code does not follow best practices for object-oriented design.

Interface References

You can use an interface as a reference type. When using an interface reference type, you must use only the methods outlined in the interface.

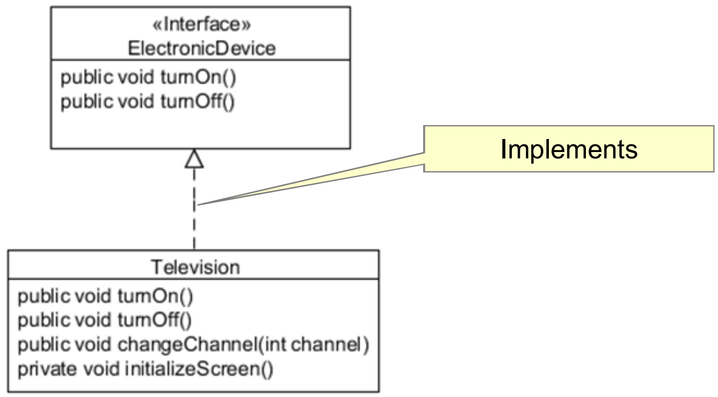
```
ElectronicDevice ed = new Television();
ed.turnOn();
ed.turnOff();
ed.changeChannel(2); // fails to compile
String s = ed.toString();
```


An interface-typed reference can be used only to reference an object that implements that interface. If an object has all the methods outlined in the interface but does not implement the interface, you may not use the interface as a reference type for that object. Interfaces implicitly include all the methods from `java.lang.Object`.

`instanceof` Operator

You can use `instanceof` with interfaces.

```
Television t = new Television();
if (t instanceof ElectronicDevice) { }
```



Television is an instance of an ElectronicDevice.

Previously, you used `instanceof` on class types. Any type that can be used as a reference can be used as an operand for the `instanceof` operator. In the slide, a `Television` implements `ElectronicDevice`. Therefore, a `Television` is an instance of a `Television`, an `ElectronicDevice`, and a `java.lang.Object`.

Marker Interfaces

- Marker interfaces define a type but do not outline any methods that must be implemented by a class.

```
public class Person implements java.io.Serializable { }
```

- The only reason these type of interfaces exist is type checking.

```
Person p = new Person();
if (p instanceof Serializable) {

}
```


`java.io.Serializable` is a marker interface used by Java's I/O library to determine if an object can have its state serialized. When implementing `Serializable`, you are not required to provide method implementations. Testing (in the form of the `instanceof` operator) for the serializability of an object is built into the standard I/O libraries. You use this interface in the lesson titled “Java I/O Fundamentals.”

Casting to Interface Types

You can cast to an interface type.

```
public static void turnObjectOn(Object o) {
    if (o instanceof ElectronicDevice) {
        ElectronicDevice e = (ElectronicDevice)o;
        e.turnOn();
    }
}
```


Casting Guidelines

Just as you do when casting to class types, if you cast to a type that is invalid for that object, your application generates an exception and is even likely to crash. To verify that a cast will succeed, you should use an `instanceof` test.

The example in the slide shows poor design because the `turnObjectOn()` method operates only on `ElectronicDevice`s. Using `instanceof` and casting adds overhead at run time. When possible, use a compile-time test by rewriting the method as:

```
public static void turnObjectOn(ElectronicDevice e) {
    e.turnOn();
}
```

Using Generic Reference Types

- Use the most generic type of reference wherever possible:

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
dao.delete(1);
```

`EmployeeDAOMemoryImpl` implements `EmployeeDAO`

- By using an interface reference type, you can use a different implementing class without running the risk of breaking subsequent lines of code:

```
EmployeeDAOMemoryImpl dao = new EmployeeDAOMemoryImpl();
dao.delete(1);
```

It is possible that you could be using `EmployeeDAOMemoryImpl` only methods here.

When creating references, you should use the most generic type possible. This means that, for the object you are instantiating, you should declare the reference to be of an interface type or of a parent class type. By doing this, all usage of the reference is not tied to a particular implementing class and, if need be, you could use a different implementing class. By using an interface that several classes implement as the reference type, you have the freedom to change the implementation without affecting your code. An `EmployeeDAOMemoryImpl` typed reference could be used to invoke a method that appears only in the `EmployeeDAOMemoryImpl` class.

References typed to a specific class cause your code to be more tightly coupled to that class and potentially cause greater refactoring of your code when changing implementations.

Implementing and Extending

- Classes can extend a parent class and implement an interface:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat { }
```

- You can also implement multiple interfaces:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat, java.io.Serializable { }
```

Use a comma to separate your list of interfaces.

Extends First

If you use both `extends` and `implements`, `extends` must come first.

Extending Interfaces

- Interfaces can extend interfaces:

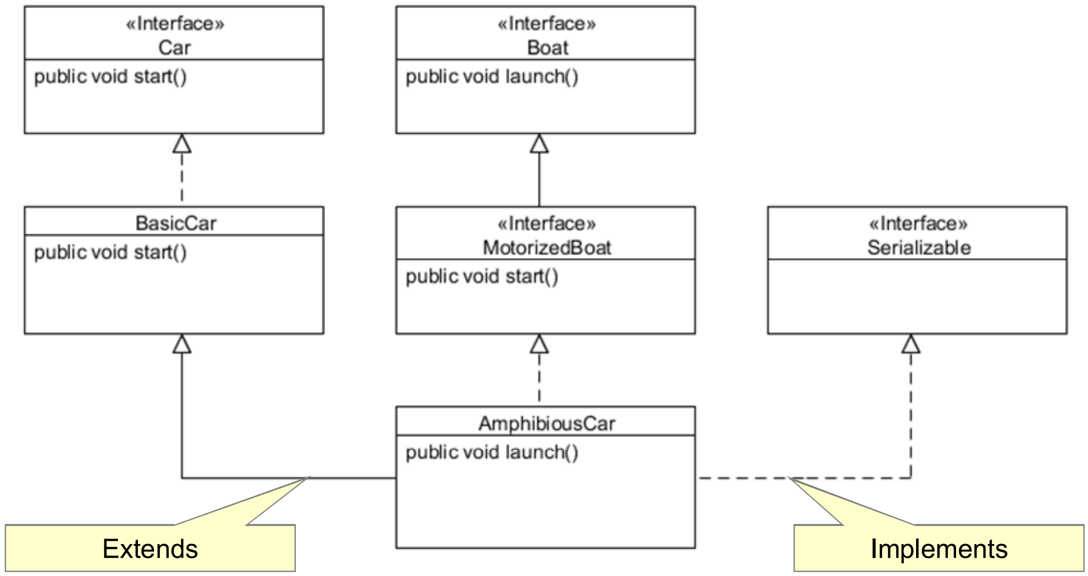
```
public interface Boat { }
```

```
public interface MotorizedBoat extends Boat { }
```

- By implementing `MotorizedBoat`, the `AmphibiousCar` class must fulfill the contract outlined by both `MotorizedBoat` and `Boat`:

```
public class AmphibiousCar extends BasicCar implements
MotorizedBoat, java.io.Serializable { }
```

Interfaces in Inheritance Hierarchies



Interface Inheritances

Interfaces are used for a form of inheritance that is referred to as *interface inheritance*. Java allows multiple interface inheritance but only single class inheritance.

Extending an Implementing Class

If you write a class that extends a class that implements an interface, the class you authored also implements the interface. For example, `AmphibiousCar` extends `BasicCar`. `BasicCar` implements `Car`; therefore, `AmphibiousCar` also implements `Car`.

Interfaces Extending Interfaces

An interface can extend another interface. For example, the interface `MotorizedBoat` can extend the `Boat` interface. If the `AmphibiousCar` class implements `MotorizedBoat`, then it must implement all methods from `Boat` and `MotorizedBoat`.

Duplicate Methods

When you have a class that implements multiple interfaces, directly or indirectly, the same method signature may appear in different implemented interfaces. If the signatures are the same, there is no conflict and only one implementation is required.

Quiz

A class can implement multiple interfaces.

a. True
b. False

Answer: a

Design Patterns and Interfaces

- One of the principles of object-oriented design is to:

  "Program to an interface, not an implementation."
- This is a common theme in many design patterns. This principle plays a role in:
  - The DAO design pattern
  - The Factory design pattern

Object-Oriented Design Principles

"Program to an interface, not an implementation" is a practice that was popularized in the book *Design Patterns: Elements of Reusable Object-Oriented Software*.

You can learn more about object-oriented design principles and design patterns in the *Java Design Patterns* course.

DAO Pattern

The Data Access Object (DAO) pattern is used when creating an application that must persist information. The DAO pattern:

- Separates the problem domain from the persistence mechanism
- Uses an interface to define the methods used for persistence. An interface allows the persistence implementation to be replaced with:
  - Memory-based DAOs as a temporary solution
  - File-based DAOs for an initial release
  - JDBC-based DAOs to support database persistence
  - Java Persistence API (JPA)–based DAOs to support database persistence

Why Separate Business and Persistence Code?

Just as the required functionality of an application will influence the design of your classes, so will other concerns. A desire for ease of maintenance and for the ability to enhance an application also influences its design. Modular code that is separated by functionality is easier to update and maintain.

By separating business and persistence logic, applications become easier to implement and maintain at the expense of additional classes and interfaces. Often these two types of logic have different maintenance cycles. For example, the persistence logic might need to be modified if the database used by the application was migrated from MySQL to Oracle 11*g*.

If you create interfaces for the classes containing the persistence logic, it becomes easier for you to replace your persistence implementation.

Before the DAO Pattern

Notice the persistence methods mixed in with the business methods.

Employee
public int getId() public String getFirstName() public String getLastName() public Date getBirthDate() public float getSalary() public String toString() //persistence methods public void save() public void delete() <u>public static Employee findById(int id)</u> <u>public static Employee[] getAllEmployees()</u>

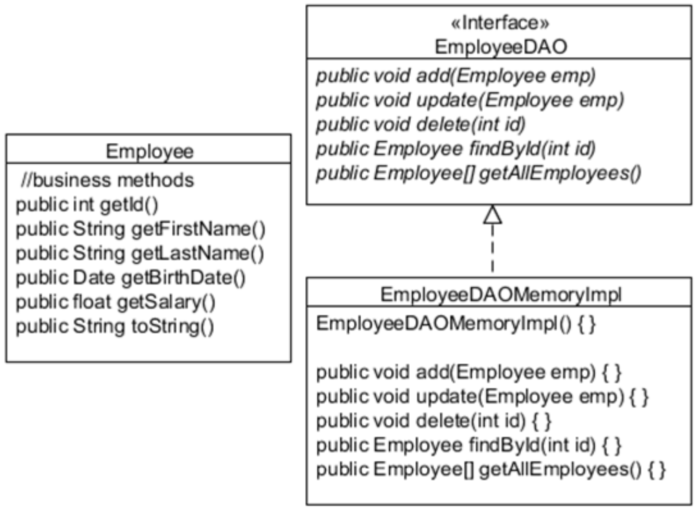
Before the DAO pattern

The Single-Responsibility Principle

The `Employee` class shown in the slide has methods that focus on two different principles or concerns. One set of methods focuses on manipulating the representation of a person, and the other deals with persisting `Employee` objects. Because these two sets of responsibilities may be modified at different points in the lifetime of your applications, it makes sense to split them up into different classes.

After the DAO Pattern

The DAO pattern moves the persistence logic out of the domain classes and into separate classes.



After refactoring to the DAO pattern

DAO Implementations

If you think that you might need to change your DAO implementation at a later time to use a different persistence mechanism, it is best to use an interface to define the contract that all DAO implementations must meet.

Your DAO interfaces outline methods to create, read, update, and delete data, although the method names can vary. When you first implement the DAO pattern, you will not see the benefit immediately. The payoff comes later, when you start modifying or replacing code. In the lesson titled "Building Database Applications with JDBC," we discuss replacing the memory-based DAO with file- and database-enabled DAOs.

The Need for the Factory Pattern

The DAO pattern depends on using interfaces to define an abstraction. Using a DAO implementation's constructor ties you to a specific implementation.

```
EmployeeDAO dao = new EmployeeDAOMemoryImpl();
```

With use of an interface type, any subsequent lines are not tied to a single implementation.

This constructor invocation is tied to an implementation and will appear in many places throughout an application.

Using the Factory Pattern

Using a factory prevents your application from being tightly coupled to a specific DAO implementation.

```
EmployeeDAOFactory factory = new EmployeeDAOFactory();
EmployeeDAO dao = factory.createEmployeeDAO();
```

The `EmployeeDAO` implementation is hidden.

This pattern eliminates direct constructor calls in favor of invoking a method. A factory is often used when implementing the DAO pattern.

In the example in the slide, you have no idea what type of persistence mechanism is used by `EmployeeDAO` because it is just an interface. The factory could return a DAO implementation that uses files or a database to store and retrieve data. As a developer, you want to know what type of persistence is being used because it factors into the performance and reliability of your application. But you do not want the majority of the code you write to be tightly coupled with the type of persistence.

The Factory

The implementation of the factory is the only point in the application that should depend on concrete DAO classes.

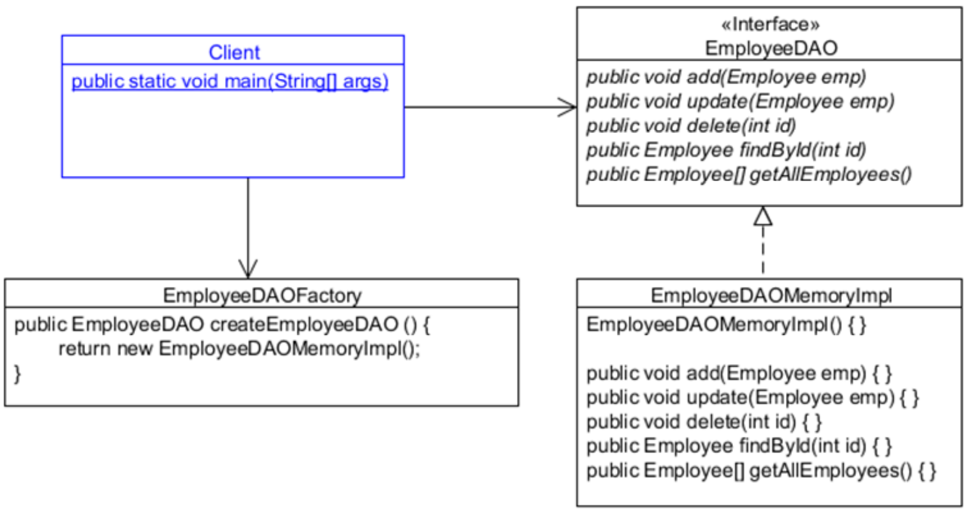
```
public class EmployeeDAOFactory {

    public EmployeeDAO createEmployeeDAO() {
        return new EmployeeDAOMemoryImpl();
    }
}
```

Returns an interface typed reference

For simplicity, this factory hardcodes the name of a concrete class to instantiate. You could enhance this factory by putting the class name in an external source such as a text file and use the `java.lang.Class` class to instantiate the concrete subclass. A basic example of using the `java.lang.Class` is the following:

```
String name = "com.example.dao.EmployeeDAOMemoryImpl";
Class clazz = Class.forName(name);
EmployeDAO dao = (EmployeeDAO)clazz.newInstance();
```

The DAO and Factory Together



Clients depending only on abstract DAOs

Quiz

A typical singleton implementation contains a factory method.

a. True

b. False

Answer: a

Code Reuse

Code duplication (copy and paste) can lead to maintenance problems. You do not want to fix the same bug multiple times.

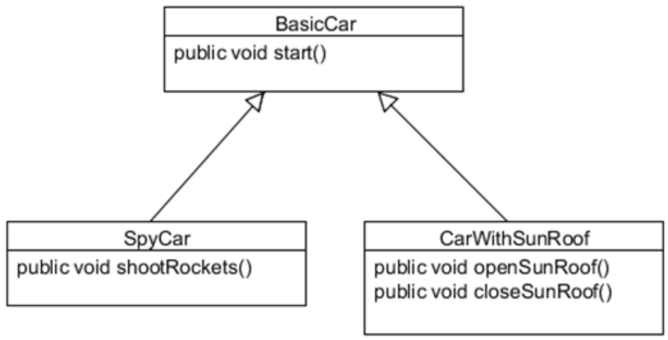
- "Don't repeat yourself!" (DRY principle)
- Reuse code in a good way:
  - Refactor commonly used routines into libraries.
  - Move the behavior shared by sibling classes into their parent class.
  - Create new combinations of behaviors by combining multiple types of objects together (composition).

Copying and pasting code is *not* something that must always be avoided. If duplicated code serves as a starting point and is heavily modified, that may be an acceptable situation for you to copy and paste lines of code. You should be aware of how much copying and pasting is occurring in a project. Besides performing manual code audits, there are tools you can use to detect duplicated code. For one such example, refer to http://pmd.sourceforge.net/cpd.html.

Design Difficulties

Class inheritance allows for code reuse but is not very modular

- How do you create a SpyCarWithSunRoof?



Method implementations located across different classes

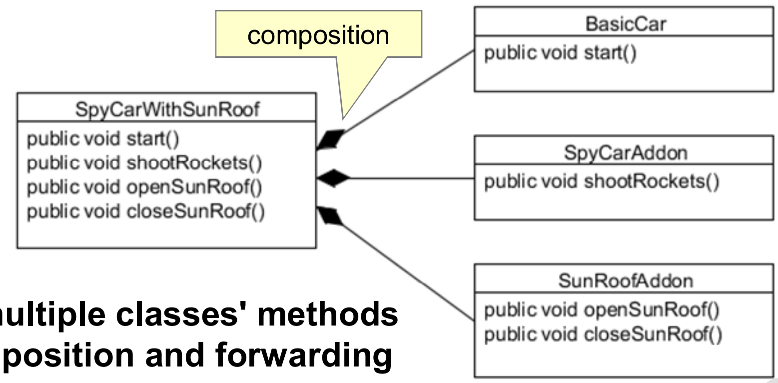
Limitations with Inheritance

Java supports only single class inheritance, which eliminates the possibility of inheriting different implementations of a method with the same signature. Multiple interface inheritance does not pose the same problem as class inheritance because there can be no conflicting method implementations in interfaces.

Composition

Object composition allows you to create more complex objects. To implement composition, you:

1. Create a class with references to other classes.
2. Add same signature methods that forward to the referenced objects.



Combining multiple classes' methods through composition and forwarding

Delegation

Method delegation and method forwarding are two terms that are often used interchangeably. Method forwarding is when you write a method that does nothing except pass execution over to another method. In some cases, delegation may imply more than simple forwarding. For more on the difference between the two, refer to page 20 of the book *Design Patterns: Elements of Reusable Object-Oriented Software*.

Composition Implementation

```
public class SpyCarWithSunRoof {
    private BasicCar car = new BasicCar();
    private SpyCarAddon spyAddon = new SpyCarAddon();
    private SunRoofAddon roofAddon = new SunRoofAddon();

    public void start() {
        car.start();
    }

    // other forwarded methods
}
```

Method forwarding

IDE Wizards Make Implementing Composition Easy

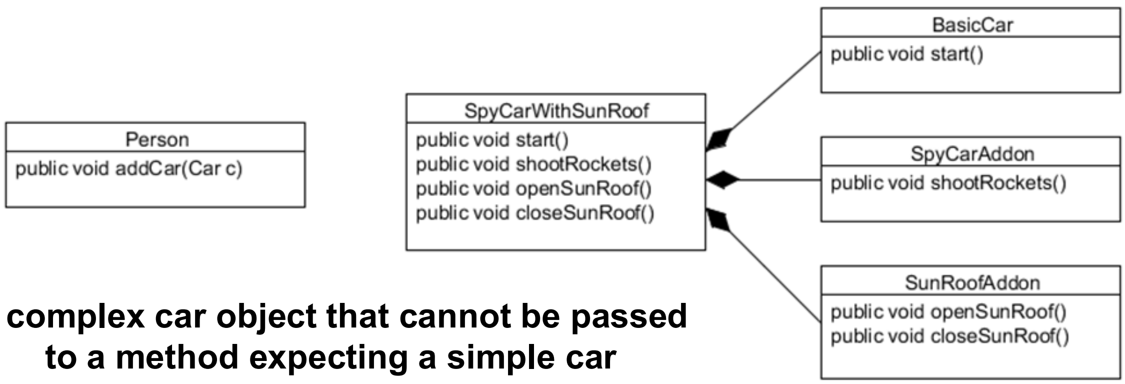
To implement composition with the NetBeans IDE, use the Insert Code tool as follows:

1. Right-click within the braces of the complex class and choose "Insert Code."
2. Select "Delegate Method."
   The Generate Delegate Methods dialog box appears.
3. Select the method calls that you want to forward.
   The methods are inserted for you.

Repeat these steps for each delegate class.

Polymorphism and Composition

Polymorphism should enable us to pass any type of Car to the `addCar` method. Composition does not enable polymorphism unless…



A complex car object that cannot be passed to a method expecting a simple car

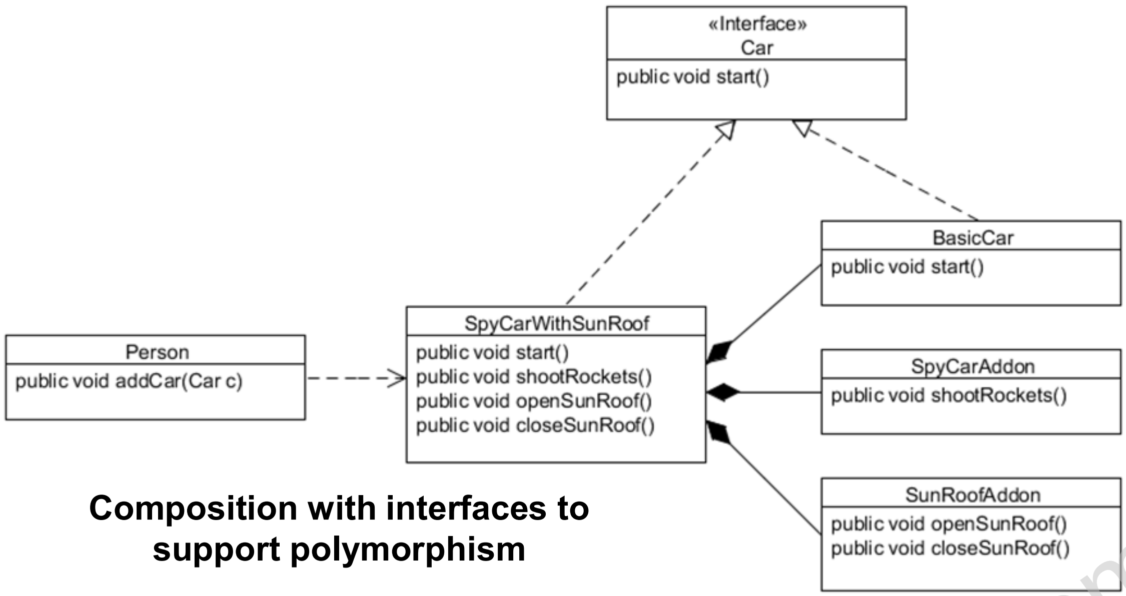
Code Reuse

The ability to use the `addCar` method for any type of Car, no matter how complex, is another form of code reuse. We cannot currently say the following:

```
addCar(new SpyCarWithSunRoof());
```

Polymorphism and Composition

Use interfaces for all delegate classes to support polymorphism.



Composition with interfaces to support polymorphism

Each delegate class that you use in a composition should have an interface defined. When creating the composing class, you declare that it implements all of the delegate interface types. By doing this, you create an object that is a composition of other objects and has many types.

Now we can say:

```
addCar(new SpyCarWithSunRoof());
```

Quiz

Method delegation is required to create complex objects using:

a. Polymorphism
b. Composition

Answer: b

Summary

In this lesson, you should have learned how to:

- Model business problems by using interfaces
- Define a Java interface
- Choose between interface inheritance and class inheritance
- Extend an interface
- Refactor code to implement the DAO pattern

Practice 6-1 Overview: Implementing an Interface

This practice covers the following topics:

- Writing an interface
- Implementing an interface
- Creating references of an interface type
- Casting to interface types

Practice 6-2 Overview: Applying the DAO Pattern

This practice covers the following topics:

- Rewriting an existing domain object with a memory-based persistence implementation using the DAO pattern
- Leveraging an abstract factory to avoid depending on concrete implementations

(Optional) Practice 6-3 Overview: Implementing Composition

This practice covers the following topics:

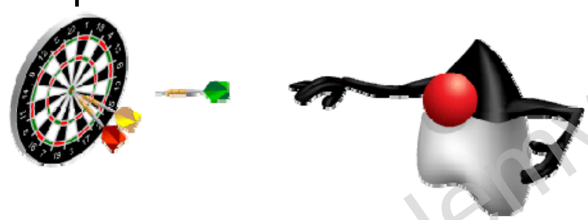
- Rewriting an existing application to better support code reuse through composition
- Using interfaces to enable polymorphism

Generics and Collections

Objectives

After completing this lesson, you should be able to:

- Create a custom generic class
- Use the type inference diamond to create an object
- Create a collection without using generics
- Create a collection by using generics
- Implement an `ArrayList`
- Implement a `Set`
- Implement a `HashMap`
- Implement a stack by using a deque
- Use enumerated types



Generics

- Provide flexible type safety to your code
- Move many common errors from runtime to compile time
- Provide cleaner, easier-to-write code
- Reduce the need for casting with collections
- Are used heavily in the Java Collections API

Simple Cache Class Without Generics

```
public class CacheString {
    private String message = "";

    public void add(String message){
        this.message = message;
    }

    public String get(){
        return this.message;
    }
}
```

```
public class CacheShirt {
    private Shirt shirt;

    public void add(Shirt shirt){
        this.shirt = shirt;
    }

    public Shirt get(){
        return this.shirt;
    }
}
```


The two examples in the slide show very simple caching classes. Even though each class is very simple, a separate class is required for any object type.

Generic Cache Class

```
public class CacheAny <T>{

   private T t;

   public void add(T t){
       this.t = t;
   }

   public T get(){
        return this.t;
    }
}
```


To create a generic version of the `CacheAny` class, a variable named `T` is added to the class definition surrounded by angle brackets. In this case, `T` stands for "type" and can represent any type. As the example shows, the code has changed to use `t` instead of a specific type information. This change allows the `CacheAny` class to store any type of object.

`T` was chosen not by accident but by convention. A number of letters are commonly used with generics.

Note: You can use any identifier you want. The following values are merely strongly suggested.

Here are the conventions:

- `T`: Type
- `E`: Element
- `K`: Key
- `V`: Value
- `S`, `U`: Used if there are second types, third types, or more

Generics in Action

Compare the type-restricted objects to their generic alternatives.

```
public static void main(String args[]){
   CacheString myMessage = new CacheString(); // Type
   CacheShirt myShirt = new CacheShirt();     // Type

   //Generics
   CacheAny<String> myGenericMessage = new CacheAny<String>();
   CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();

   myMessage.add("Save this for me"); // Type
    myGenericMessage.add("Save this for me"); // Generic

}
```


Note how the one generic version of the class can replace any number of type-specific caching classes. The `add()` and `get()` functions work exactly the same way. In fact, if the `myMessage` declaration is changed to generic, no changes need to be made to the remaining code.

The example code can be found in the Generics project in the `TestCacheAny.java` file.

Generics with Type Inference Diamond

- Syntax
  - There is no need to repeat types on the right side of the statement.
  - Angle brackets indicate that type parameters are mirrored.
- Simplifies generic declarations
- Saves typing

```
//Generics
CacheAny<String> myMessage = new CacheAny<>();
}
```

ORACLE

The type inference diamond is a new feature in JDK 7. In the generic code, notice how the right-side type definition is always equivalent to the left-side type definition. In JDK 7, you can use the diamond to indicate that the right type definition is equivalent to the left. This helps to avoid typing redundant information over and over again.

Example: `TestCacheAnyDiamond.java`

Note: In a way, it works in an opposite way from a “normal” Java type assignment. For example, `Employee emp = new Manager();` makes `emp` object an instance of `Manager`.

But in the case of generics:

```
ArrayList<Manager>  managementTeam = new ArrayList<>();
```

It is the left side of the expression (rather than the right side) that determines the type.

Quiz

Which of the following is *not* a conventional abbreviation for use with generics?

a. T: Table
b. E: Element
c. K: Key
d. V: Value

Answer: a

Collections

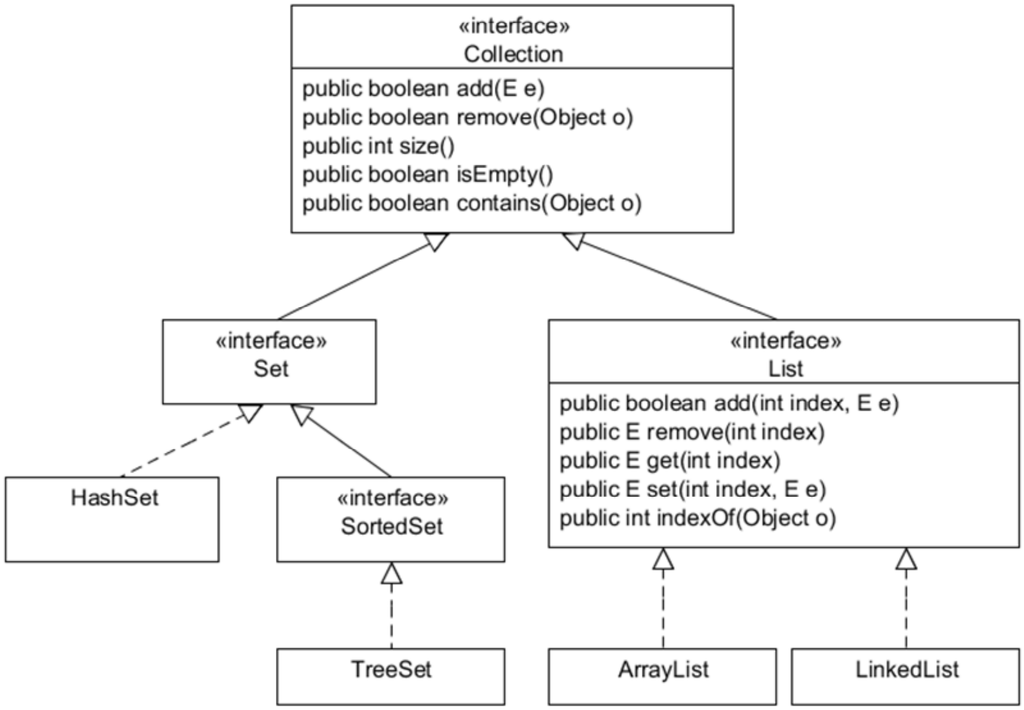
- A collection is a single object designed to manage a group of objects.
  - Objects in a collection are called *elements.*
  - *Primitives are not allowed in a collection.*
- Various collection types implement many common data structures:
  - Stack, queue, dynamic array, hash
- The Collections API relies heavily on generics for its implementation.

A collection is a single object that manages a group of objects. Objects in the collection are called *elements*. Various collection types implement standard data structures including stacks, queues, dynamic arrays, and hashes. All the collection objects have been optimized for use in Java applications.

Note: The Collections classes are all stored in the `java.util` package. The `import` statements are not shown in the following examples, but the `import` statements are required for each collection type:

- `import java.util.List;`
- `import java.util.ArrayList;`
- `import java.util.Map;`

Collection Types



The diagram in the slide shows all the collection types that descend from `Collection`. Some sample methods are provided for both `Collection` and `List`. Note the use of generics.

Characteristics of Implementation Classes

- `HashSet`: A collection of elements that contains no duplicate elements
- `TreeSet`: A sorted collection of elements that contains no duplicate elements
- `ArrayList`: A dynamic array implementation
- `Deque`: A collection that can be used to implement a stack or a queue

Note: The `Map` interface is a separate inheritance tree and is discussed later in the lesson.

List Interface

- List is an interface that defines generic list behavior.
  - An ordered collection of elements
- List behaviors include:
  - Adding elements at a specific index
  - Adding elements to the end of the list
  - Getting an element based on an index
  - Removing an element based on an index
  - Overwriting an element based on an index
  - Getting the size of the list
- Use List as a reference type to hide implementation details.

The List interface is the basis for all Collections classes that exhibit list behavior.

`ArrayList` Implementation Class

- Is a dynamically growable array
  - The list automatically grows if elements exceed initial size.
- Has a numeric index
  - Elements are accessed by index.
  - Elements can be inserted based on index.
  - Elements can be overwritten.
- Allows duplicate items

```
List<Integer> partList = new ArrayList<>(3);
partList.add(new Integer(1111));
partList.add(new Integer(2222));
partList.add(new Integer(3333));
partList.add(new Integer(4444)); // ArrayList auto grows
System.out.println("First Part: " + partList.get(0)); // First item
partList.add(0, new Integer(5555)); // Insert an item by index
```


An `ArrayList` implements a `List` collection. The implementation exhibits characteristics of a dynamically growing array. A “to-do list” application is a good example of an application that can benefit from an `ArrayList`.

`ArrayList` Without Generics

```
1  public class OldStyleArrayList {
2    public static void main(String args[]){
3      List partList = new ArrayList(3);
4
5      partList.add(new Integer(1111));
6      partList.add(new Integer(2222));
7      partList.add(new Integer(3333));
8      partList.add("Oops a string!");
9
10     Iterator elements = partList.iterator();
11     while (elements.hasNext()) {
12       Integer partNumberObject = (Integer)(elements.next()); // error?
13       int partNumber = partNumberObject.intValue();
14
15       System.out.println("Part number: " + partNumber);
16     }
17   }
18 }
```

Java example using syntax prior to Java 1.5.

Runtime error: `ClassCastException`

In the example in the slide, a part number list is created using an `ArrayList`. Using syntax prior to Java version 1.5, there is no type definition. So any type can be added to the list as shown on line 8. So it is up to the programmer to know what objects are in the list and in what order. If an assumption were made that the list was only for `Integer` objects, a runtime error would occur on line 12.

On lines 10–16, with a non-generic collection, an `Iterator` is used to iterate through the list of items. Notice that a lot of casting is required to get the objects back out of the list so you can print the data.

In the end, there is a lot of needless “syntactic sugar” (extra code) working with collections in this way.

If the line that adds the `String` to the `ArrayList` is commented out, the program produces the following output:

```
Part number: 1111
Part number: 2222
Part number: 3333
```

Generic `ArrayList`

```
1  public class GenericArrayList {
2    public static void main(String args[]) {
3      List<Integer> partList = new ArrayList<>(3);
4
5      partList.add(new Integer(1111));
6      partList.add(new Integer(2222));
7      partList.add(new Integer(3333));
8      partList.add("Bad Data");  // compile error now
9
10     Iterator<Integer> elements = partList.iterator();
11     while (elements.hasNext()) {
12       Integer partNumberObject = elements.next();
13       int partNumber = partNumberObject.intValue();
14
15       System.out.println("Part number: " + partNumber);
16     }
17   }
18 }
```

Java example using SE 7 syntax.

No cast required.

With generics, things are much simpler. When the `ArrayList` is initialized on line 3, any attempt to add an invalid value (line 8) results in a compile-time error.

Note: On line 3, the `ArrayList` is assigned to a `List` type. Using this style enables you to swap out the `List` implementation without changing other code.

Generic `ArrayList`: Iteration and Boxing

```
for (Integer partNumberObj:partList){
      int partNumber = partNumberObj; // Demos auto unboxing
      System.out.println("Part number: " + partNumber);
}
```

- The enhanced `for` loop, or `for-each` loop, provides cleaner code.
- No casting is done because of autoboxing and unboxing.

Using the `for-each` loop is much easier and provides much cleaner code. No casts are done because of the autounboxing feature of Java.

Autoboxing and Unboxing

- Simplifies syntax
- Produces cleaner, easier-to-read code

```
1 public class AutoBox {
2    public static void main(String[] args){
3        Integer intObject = new Integer(1);
4        int intPrimitive = 2;
5
6        Integer tempInteger;
7        int tempPrimitive;
8
9        tempInteger = new Integer(intPrimitive);
10        tempPrimitive = intObject.intValue();
11
12        tempInteger = intPrimitive; // Auto box
13        tempPrimitive = intObject;  // Auto unbox
```


Lines 9 and 10 show a traditional method for moving between objects and primitives. Lines 12 and 13 show boxing and unboxing.

Autoboxing and Unboxing

Autoboxing and unboxing are Java language features that enable you to make sensible assignments without formal casting syntax. Java provides the casts for you at compile time.

Note: Be careful when using autoboxing in a loop. There is a performance cost to using this feature.

Quiz

Assuming a valid Employee class, and given this fragment:

```
1 List<Object> staff = new ArrayList<>(3);
2 staff.add(new Employee(101, "Bob Andrews"));
3 staff.add(new Employee(102, "Fred Smith"));
4 staff.add(new Employee(103, "Susan Newman"));
5 staff.add(3, new Employee(104, "Tim Downs"));
6 Iterator<Employee> elements = staff.iterator();
7 while (elements.hasNext())
8   System.out.println(elements.next().getName());
```

What change is required to allow this code to compile?

a. Line 1: The (3) needs to be (4)
b. Line 8: Need to cast `elements.next()` to Employee before invoking `getName()`
c. Line 1: `Object` needs to be `Employee`
d. Line 5: The 3 needs to be 4

Answer: c

`Set` Interface

- A set is a list that contains only unique elements.
- A set has no index.
- Duplicate elements are not allowed in a set.
- You can iterate through elements to access them.
- `TreeSet` provides sorted implementation.

As an example, a set can be used to track a list of unique part numbers.

`Set` Interface: Example

A set is a collection of unique elements.

```
public class SetExample {
    public static void main(String[] args){
        Set<String> set = new TreeSet<>();

        set.add("one");
        set.add("two");
        set.add("three");
        set.add("three"); // not added, only unique

        for (String item:set){
            System.out.println("Item: " + item);
        }
    }
}
```


A set is a collection of unique elements. This example uses a `TreeSet`, which sorts the items in the set. If the program is run, the output is as follows:

```
Item: one
Item: three
Item: two
```

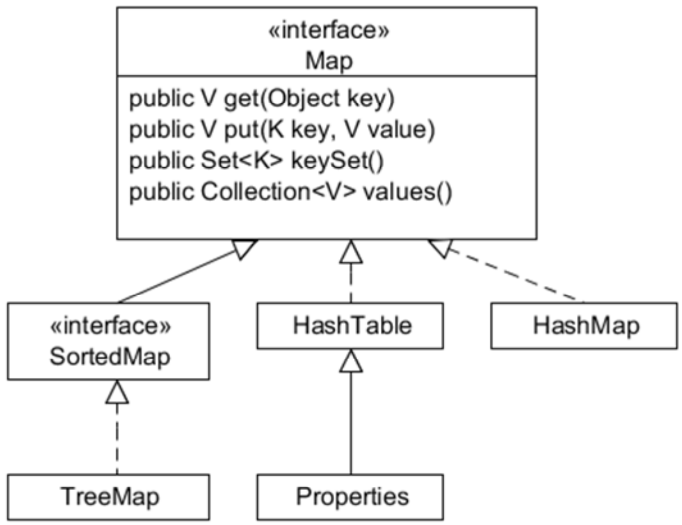
`Map` Interface

- A collection that stores multiple key-value pairs
  - Key: Unique identifier for each element in a collection
  - Value: A value stored in the element associated with the key
- Called “associative arrays” in other languages

Key	Value
101	Blue Shirt
102	Black Shirt
103	Gray Shirt

A `Map` is good for tracking things like part lists and their descriptions (as shown in the slide).

`Map` Types



ORACLE

The `Map` interface does not extend the `Collection` interface because it represents mappings and not a collection of objects. Some of the key implementation classes include:

- `TreeMap`: A map where the keys are automatically sorted
- `HashTable`: A classic associative array implementation with keys and values. `HashTable` is synchronized.
- `HashMap`: An implementation just like `HashTable` except that it accepts null keys and values. Also, it is not synchronized.

`Map` Interface: Example

```
public class MapExample {
   public static void main(String[] args){
       Map <String, String> partList = new TreeMap<>();
       partList.put(“S001", "Blue Polo Shirt");
       partList.put(“S002", "Black Polo Shirt");
       partList.put(“H001", "Duke Hat");

       partList.put(“S002", "Black T-Shirt"); // Overwrite value
       Set<String> keys = partList.keySet();

        System.out.println("=== Part List ===");
        for (String key:keys){
            System.out.println("Part#: " + key + " " +
                                    partList.get(key));
        }
    }
}
```

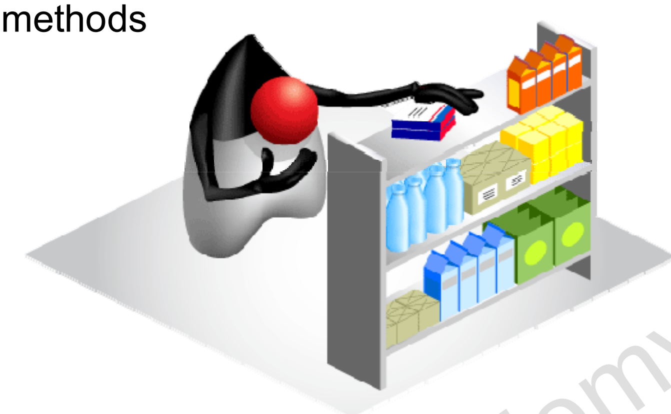

The example shows how to create a `Map` and perform standard operations on it. The output from the program is:

```
=== Part List ===
Part#: 111111 Blue Polo Shirt
Part#: 222222 Black T-Shirt
Part#: 333333 Duke Hat
```

`Deque` Interface

A collection that can be used as a stack or a queue

- Means "double-ended queue" (and is pronounced "deck")
- A queue provides FIFO (first in, first out) operations
  - `add(e)` and `remove()` methods
- A stack provides LIFO (last in, first out) operations
  - `push(e)` and `pop()` methods



`Deque` is a child interface of `Collection` (just like `Set` and `List`).

A queue is often used to track asynchronous message requests so they can be processed in order. A stack can be very useful for traversing a directory tree or similar structures.

Stack with Deque: Example

```
public class TestStack {
   public static void main(String[] args){
       Deque<String> stack = new ArrayDeque<>();
       stack.push("one");
       stack.push("two");
       stack.push("three");

       int size = stack.size() - 1;
       while (size >= 0 ) {
            System.out.println(stack.pop());
            size--;
        }
   }
}
```


A deque (pronounced “deck”) is a “doubled-ended queue.” Essentially this means that a deque can be used as a queue (first in, first out [FIFO] operations) or as a stack (last in, first out [LIFO] operations).

Ordering Collections

- The `Comparable` and `Comparator` interfaces are used to sort collections.
  – Both are implemented using generics.
- Using the `Comparable` interface:
  – Overrides the `compareTo` method
  – Provides only one sort option
- Using the `Comparator` interface:
  – Is implemented by using the `compare` method
  – Enables you to create multiple `Comparator` classes
  – Enables you to create and use numerous sorting options

The Collections API provides two interfaces for ordering elements: `Comparable` and `Comparator`.

The `Comparable` interface is implemented in a class and provides a single sorting option for the class.

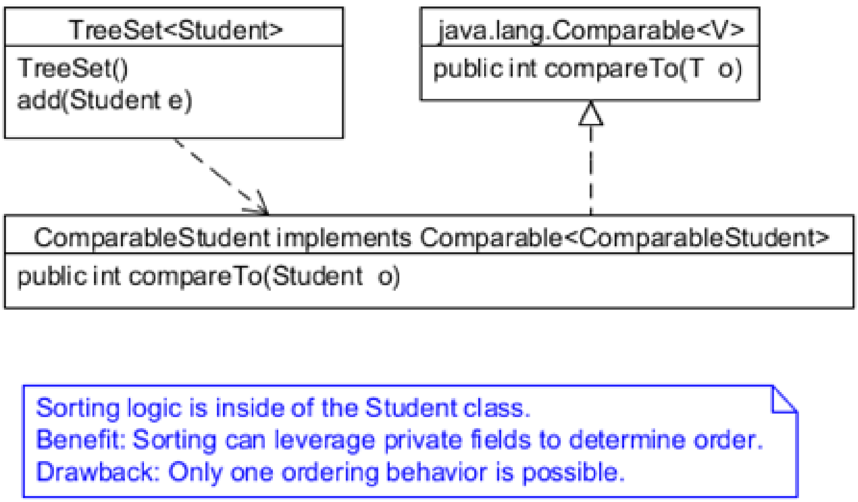
The `Comparator` interface enables you to create multiple sorting options. You plug in the designed option whenever you want.

Both interfaces can be used with sorted collections, such as `TreeSet` and `TreeMap`.

`Comparable` Interface

Using the `Comparable` interface:

- Overrides the `compareTo` method
- Provides only one sort option



The slide shows how the `ComparableStudent` class relates to the `Comparable` interface and `TreeSet`.

`Comparable`: Example

```
public class ComparableStudent implements Comparable<ComparableStudent>{
   private String name; private long id = 0; private double gpa = 0.0;

   public ComparableStudent(String name, long id, double gpa){
      // Additional code here
   }
   public String getName(){ return this.name; }
      // Additional code here

    public int compareTo(ComparableStudent s){
        int result = this.name.compareTo(s.getName());
        if (result > 0) { return 1; }
        else if (result < 0){ return -1; }
        else { return 0; }
    }
}
```


This example in the slide implements the `Comparable` interface and its `compareTo` method. Notice that because the interface is designed using generics, the angle brackets define the class type that is passed into the `compareTo` method. The `if` statements are included to demonstrate the comparisons that take place. You can also merely return a result.

The returned numbers have the following meaning.

- **Negative number:** `s` comes before the current element.
- **Positive number:** `s` comes after the current element.
- **Zero:** `s` is equal to the current element.

In cases where the collection contains equivalent values, replace the code that returns zero with additional code that returns a negative or positive number.

`Comparable` Test: Example

```
public class TestComparable {
  public static void main(String[] args){
    Set<ComparableStudent> studentList = new TreeSet<>();

    studentList.add(new ComparableStudent("Thomas Jefferson", 1111, 3.8));
    studentList.add(new ComparableStudent("John Adams", 2222, 3.9));
    studentList.add(new ComparableStudent("George Washington", 3333, 3.4));

    for(ComparableStudent student:studentList){
      System.out.println(student);
    }
  }
}
```


In the example in the slide, an `ArrayList` of `ComparableStudent` elements is created. After the list is initialized, it is sorted using the `Comparable` interface. The output of the program is as follows:

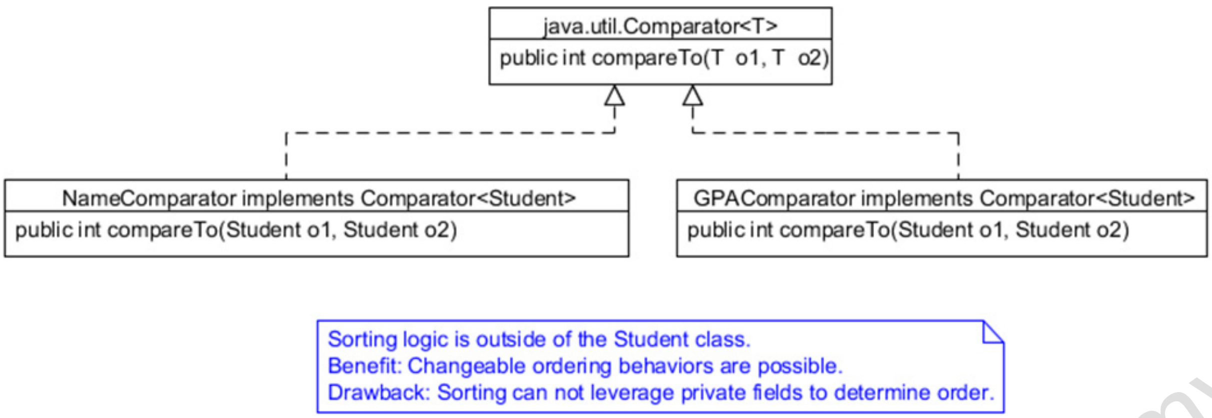
```
Name: George Washington  ID: 3333  GPA:3.4
Name: John Adams  ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
```

Note: The `ComparableStudent` class has overridden the `toString()` method.

`Comparator` Interface

Using the `Comparator` interface:

- Is implemented by using the `compare` method
- Enables you to create multiple `Comparator` classes
- Enables you to create and use numerous sorting options



The slide shows two `Comparator` classes that can be used with the `Student` class. The example in the next slide shows how to use `Comparator` with an unsorted interface like `ArrayList` by using the `Collections` utility class.

`Comparator`: Example

```
public class StudentSortName implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        int result = s1.getName().compareTo(s2.getName());
        if (result != 0) { return result; }
        else {
            return 0;  // Or do more comparing
        }
    }
}
```

```
public class StudentSortGpa implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        if (s1.getGpa() < s2.getGpa()) { return 1; }
        else if (s1.getGpa() > s2.getGpa()) { return -1; }
        else { return 0; }
    }
}
```

Here the compare logic is reversed and results in descending order.

ORACLE

The example in the slide shows the `Comparator` classes that are created to sort based on Name and GPA. For the name comparison, the `if` statements have been simplified.

`Comparator` Test: Example

```
1  public class TestComparator {
2    public static void main(String[] args){
3        List<Student> studentList = new ArrayList<>(3);
4        Comparator<Student> sortName = new StudentSortName();
5        Comparator<Student> sortGpa = new StudentSortGpa();
6
7        // Initialize list here
8
9        Collections.sort(studentList, sortName);
10        for(Student student:studentList){
11            System.out.println(student);
12        }
13
14        Collections.sort(studentList, sortGpa);
15        for(Student student:studentList){
16            System.out.println(student);
17        }
18    }
19 }
```


The example in the slide shows how the two `Comparator` objects are used with a collection.

Note: Some code has been commented out to save space.

Notice how the `Comparator` objects are initialized on lines 4 and 5. After the `sortName` and `sortGpa` variables are created, they can be passed to the `sort()` method by name. Running the program produces the following output.

```
Name: George Washington  ID: 3333  GPA:3.4
Name: John Adams  ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
Name: John Adams  ID: 2222  GPA:3.9
Name: Thomas Jefferson  ID: 1111  GPA:3.8
Name: George Washington  ID: 3333  GPA:3.4
```

Notes

- The `Collections` utility class provides a number of useful methods for various collections. Methods include `min()`, `max()`, `copy()`, and `sort()`.
- The `Student` class has overridden the `toString()` method.

Quiz

Which interface would you use to create multiple sort options for a collection?

a. `Comparable`

b. `Comparison`

c. `Comparator`

d. `Comparinator`

Answer: c

Summary

In this lesson, you should have learned how to:

- Create a custom generic class
- Use the type inference diamond to create an object
- Create a collection without using generics
- Create a collection by using generics
- Implement an `ArrayList`
- Implement a `Set`
- Implement a `HashMap`
- Implement a stack by using a deque
- Use enumerated types

Practice 7-1 Overview: Counting Part Numbers by Using a `HashMap`

This practice covers the following topics:

- Creating a map to store a part number and count
- Creating a map to store a part number and description
- Processing the list of parts and producing a report

Practice 7-2 Overview: Matching Parentheses by Using a Deque

This practice covers processing programming statements to ensure that the number of parentheses matches.

ORACLE

Practice 7-3 Overview:
Counting Inventory and Sorting with `Comparators`

This practice covers processing inventory transactions that generate two reports sorted differently using `Comparators`.

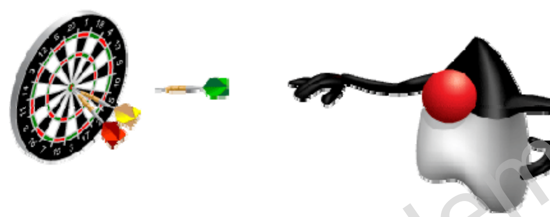
8

String Processing

Objectives

After completing this lesson, you should be able to:

- Read data from the command line
- Search strings
- Parse strings
- Create strings by using a `StringBuilder`
- Search strings by using regular expressions
- Parse strings by using regular expressions
- Replace strings by using regular expressions



Command-Line Arguments

- Any Java technology application can use command-line arguments.
- These string arguments are placed on the command line to launch the Java interpreter after the class name:

```
java TestArgs arg1 arg2 "another arg"
```

- Each command-line argument is placed in the args array that is passed to the static main method:

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


When a Java program is launched from a terminal window, you can provide the program with zero or more command-line arguments.

These command-line arguments enable the user to specify the configuration information for the application. These arguments are strings: either stand-alone tokens (such as arg1) or quoted strings (such as "another arg").

Command-Line Arguments

```
public class TestArgs {
   public static void main(String[] args) {
     for ( int i = 0; i < args.length; i++ ) {
        System.out.println("args[" + i + "] is '" +
                           args[i] + "'");
     }
   }
}
```

Example execution:

```
java TestArgs "Ted Baxter" 45 100.25
args[0] is 'Ted Baxter'
args[1] is '45'
args[2] is '100.25'
```


Command-line arguments are always passed to the main method as strings, regardless of their intended type. If an application requires command-line arguments other than type `String` (for example, numeric values), the application should convert the string arguments to their respective types using the wrapper classes, such as the `Integer.parseInt` method, which can be used to convert the string argument that represents the numeric integer to type `int`.

Properties

- The `java.util.Properties` class is used to load and save key-value pairs in Java.
- Can be stored in a simple text file:

```
hostName = www.example.com
userName = user
password = pass
```

- File name ends in `.properties`.
- File can be anywhere that compiler can find it.

The benefit of a properties file is the ability to set values for your application externally. The properties file is typically read at the start of the application and is used for default values. But the properties file can also be an integral part of a localization scheme, where you store the values of menu labels and text for various languages that your application may support.

The convention for a properties file is `<filename>.properties`, but the file can have any extension you want. The file can be located anywhere that the application can find it.

Loading and Using a Properties File

```
public static void main(String[] args) {
  Properties myProps = new Properties();
  try {
    FileInputStream fis = new FileInputStream("ServerInfo.properties");
    myProps.load(fis);
  } catch (IOException e) {
    System.out.println("Error: " + e.getMessage());
  }

   // Print Values
   System.out.println("Server: " + myProps.getProperty("hostName"));
   System.out.println("User: " + myProps.getProperty("userName"));
   System.out.println("Password: " + myProps.getProperty("password"));
}
```


In the code fragment, you create a `Properties` object. Then, using a `try` statement, you open a file relative to the source files in your NetBeans project. When it is loaded, the name-value pairs are available for use in your application.

Properties files enable you to easily inject configuration information or other application data into the application.

Loading Properties from the Command Line

- Property information can also be passed on the command line.
- Use the –D option to pass key-value pairs:

```
java -Dpropertyname=value -Dpropertyname=value myApp
```

- For example, pass one of the previous values"

```
java -Dusername=user myApp
```

- Get the `Properties` data from the `System` object:

```
String userName = System.getProperty("username");
```


Property information can also be passed on the command line. The advantage to passing properties from the command line is simplicity. You do not have to open a file and read from it. However, if you have more than a few parameters, a properties file is preferable.

`PrintWriter` and the Console

The `PrintWriter` class writes characters instead of bytes. The class implements all of the print methods found in `PrintStream`.

```
import java.io.PrintWriter;

public class PrintWriterExample {
    public static void main(String[] args){
        PrintWriter pw = new PrintWriter(System.out, true);
        pw.println("This is some output.");

    }
}
```


The `PrintStream` class converts characters into bytes using the platform's default character encoding.

Unlike the `PrintStream` class, if automatic flushing is enabled it will be done only when one of the `println`, `printf`, or `format` methods is invoked, rather than whenever a newline character is included in the output.

The example in the slide shows how to create the object using the `autoFlush` option. The `true` option is required to force `PrintWriter` to flush each line printed to the console.

`printf` format

Java provides many ways to format strings:

- `printf` and `String.format`

```
public class PrintfExample {
    public static void main(String[] args){
        PrintWriter pw = new PrintWriter(System.out, true);
        double price = 24.99; int quantity = 2; String color = "Blue";
        System.out.printf("We have %03d %s Polo shirts that cost
    $%3.2f.\n", quantity, color, price);
        System.out.format("We have %03d %s Polo shirts that cost
    $%3.2f.\n", quantity, color, price);
        String out = String.format("We have %03d %s Polo shirts that cost
    $%3.2f.", quantity, color, price);
        System.out.println(out);
        pw.printf("We have %03d %s Polo shirts that cost $%3.2f.\n",
    quantity, color, price);
    }
}
```


You can perform the `printf` format using both the String class and any output stream. The slide shows several different string formatting examples. See the Java API documentation for details about all the options.

- **`%s`:** String
- **`%d`:** Decimal
- **`%f`:** Float

The program output is the following:

```
We have 002 Blue Polo shirts that cost $24.99.
We have 002 Blue Polo shirts that cost $24.99.
We have 002 Blue Polo shirts that cost $24.99.
We have 002 Blue Polo shirts that cost $24.99.
```

Quiz

Which two of the following are valid formatted print statements?

a. `System.out.printf("%s Polo shirts cost $%3.2f.\n", "Red", "35.00");`

b. `System.out.format("%s Polo shirts cost $%3.2f.\n", "Red", "35.00");`

c. `System.out.println("Red Polo shirts cost $35.00.\n");`

d. `System.out.print("Red Polo shirts cost $35.00.\n");`

Answer: a, b

String Processing

- `StringBuilder` for constructing string
- Built-in string methods
  - Searching
  - Parsing
  - Extracting substring
- Parsing with `StringTokenizer`

The first part of this section covers string functions that are not regular expressions. When you perform simple string manipulations, there are a number of very useful built-in methods.

`StringBuilder` and `StringBuffer`

- `StringBuilder` and `StringBuffer` are the preferred tools when string concatenation is nontrivial.
  - More efficient than "+"
- Concurrency
  - `StringBuilder` (not thread-safe)
  - `StringBuffer` (thread-safe)
- Set capacity to the size you actually need.
  - Constant buffer resizing can also lead to performance problems.

The `StringBuilder` and `StringBuffer` classes are the preferred way to concatenate strings.

StringBuilder: Example

```
public class StringBuilding {
    public static void main(String[] args){
        StringBuilder sb = new StringBuilder(500);

        sb.append(", the lightning flashed and the thunder
    rumbled.\n");
        sb.insert(0, "It was a dark and stormy night");

        sb.append("The lightning struck...\n").append("[  ");
        for(int i = 1; i < 11; i++){
            sb.append(i).append("  ");
        }
        sb.append("] times");

        System.out.println(sb.toString());
    }
}
```

ORACLE

The example in the slide shows some common `StringBuilder` methods. You can use `StringBuilder` to insert text in position. Chaining append calls together is a best practice for building strings.

The output from the program is as follows:

```
It was a dark and stormy night, the lightning flashed and the thunder
rumbled.
The lightning struck...
[  1  2  3  4  5  6  7  8  9  10 ] times
```

Sample String Methods

```
public class StringMethodsExample {
    public static void main(String[] args){
        PrintWriter pw = new PrintWriter(System.out, true);
        String tc01 = "It was the best of times";
        String tc02 = "It was the worst of times";

        if (tc01.equals(tc02)){
            pw.println("Strings match..."); }
        if (tc01.contains("It was")){
            pw.println("It was found"); }
        String temp = tc02.replace("w", "zw");
        pw.println(temp);
        pw.println(tc02.substring(5, 12));
    }
}
```


The code in the slide demonstrates some of the more useful string methods of the String class.

- **`equals()`:** Tests the equality of the contents of two strings. This is preferable to `==`, which tests whether two objects point to the same reference.
- **`contains()`:** Searches a string to see if it contains the string provided
- **`replace()`:** Searches for the string provided and replaces all instances with the target string provided. There is a `replaceFirst()` method for replacing only the first instance.
- **`substring()`:** Returns a string based on its position in the string

Running the programs in the slide returns the following output:

```
It was found
It zwas the zworst of times
s the w
```

Using the `split()` Method

```
public class StringSplit {
    public static void main(String[] args){
        String shirts = "Blue Shirt, Red Shirt, Black Shirt, Maroon Shirt";

        String[] results = shirts.split(", ");
        for(String shirtStr:results){
            System.out.println(shirtStr);
        }
    }
}
```


The simplest way to parse a string is using the `split()` method. Call the method with the character (or characters) that will split the string apart. The result is captured in an array.

Note: The delimiter can be defined using regular expressions.

The output of the program in the slide is as follows:

```
Blue Shirt
Red Shirt
Black Shirt
Maroon Shirt
```

Parsing with `StringTokenizer`

```
public class StringTokenizerExample {
    public static void main(String[] args){
        String shirts = "Blue Shirt, Red Shirt, Black Shirt, Maroon Shirt";

        StringTokenizer st = new StringTokenizer(shirts, ", ");

        while(st.hasMoreTokens()){
            System.out.println(st.nextToken());
        }
    }
}
```


The `StringTokenizer` class does the same thing as `split()` but takes a different approach. You must iterate the tokens to get access to them. Also note that the delimiter `", "` in this case means use both commas and spaces as delimiters. Thus, the result from parsing is the following:

```
Blue
Shirt
Red
Shirt
Black
Shirt
Maroon
Shirt
```

Scanner

A `Scanner` can tokenize a string or a stream.

```
1        public static void main(String[] args) {
2            Scanner s = null;
3            StringBuilder sb = new StringBuilder(64);
4            String line01 = "1.1, 2.2, 3.3";
5            float fsum = 0.0f;
6
7            s = new Scanner(line01).useDelimiter(", ");
8            try {
9                while (s.hasNextFloat()) {
10                   float f = s.nextFloat();
11                   fsum += f;
12                   sb.append(f).append(" ");
13               }
14               System.out.println("Values found: " + sb.toString());
15               System.out.println("FSum: " + fsum);
16           } catch (Exception e) {
17               System.out.println(e.getMessage());
18           }
```


A `Scanner` can be used to tokenize an input stream or a string. In addition, a `Scanner` can be used to tokenize numbers and convert them into any primitive number type. Note how the `Scanner` is defined on line 7. The resulting object can be iterated over based on a specific type. In this case, a `float` is used.

The output from this code segment is as follows:

```
Values found: 1.1 2.2 3.3
FSum: 6.6000004
```

Regular Expressions

- A language for matching strings of text
  - Very detailed vocabulary
  - Search, extract, or search and replace
- With Java, the backslash (\) is not fun.
- Java objects
  - `Pattern`
  - `Matcher`
  - `PatternSyntaxException`
  - `java.util.regex`

`Pattern` and `Matcher`

- `Pattern`: Defines a regular expression
- `Matcher`: Specifies a string to search

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternExample {
    public static void main(String[] args){
        String t = "It was the best of times";

        Pattern pattern = Pattern.compile("the");
        Matcher matcher = pattern.matcher(t);

        if (matcher.find()) { System.out.println("Found match!"); }
    }
}
```


The `Pattern` and `Matcher` objects work together to provide a complete solution.

The `Pattern` object defines the regular expression that will be used for the search. As shown in the example, a regular expression can be as simple as a word or phrase.

The `Matcher` object is then used to select the target string to be searched. A number of methods are available for matcher. They are covered in the following slides.

When run, the example produces the following output:

```
Found match!
```

Character Classes

Character	Description
`.`	Matches any single character (letter, digit, or special character), except end-of-line markers
`[abc]`	Would match any “a,” “b,” or “c” in that position
`[^abc]`	Would match any character that is not “a,” “b,” or “c” in that position
`[a-c]`	A range of characters (in this case, “a,” “b,” and “c”)
`|`	Alternation; essentially an “or” indicator

Character classes enable you to match one character in a number of ways.

Character Class: Examples

Target String	It was the best of times

Pattern	Description	Text Matched
`w.s`	Any sequence that starts with a "w" followed by any character followed by "s".	It was the best of times
`w[abc]s`	Any sequence that starts with a "w" followed by "a", "b", or "c" and then "s".	It was the best of times
`t[^aeo]mes`	Any sequence that starts with a "t" followed any character that is not "a", "e", or "o" followed by "mes".	It was the best of times

ORACLE

The code for this example can be found in the `StringExamples` project in the `CustomCharClassExamples.java` file.

Character Class Code: Examples

```
public class CustomCharClassExamples {
    public static void main(String[] args) {
        String t = "It was the best of times";

        Pattern p1 = Pattern.compile("w.s");
        Matcher m1 = p1.matcher(t);
        if (m1.find()) { System.out.println("Found: " + m1.group());
        }

        Pattern p2 = Pattern.compile("w[abc]s");
        Matcher m2 = p2.matcher(t);
        if (m2.find()) { System.out.println("Found: " + m2.group());
        }

        Pattern p3 = Pattern.compile("t[^eou]mes");
        Matcher m3 = p3.matcher(t);
        if (m3.find()) { System.out.println("Found: " + m3.group());
        }
```


The example in the slide shows two ways to find “`was`” and a way to find “`times`”.

To make this happen in Java:

1. Create a `Pattern` object to store the regular expression that you want to search with.
2. Create a `Matcher` object by passing the text to be searched to your `Pattern` object and returning a `Matcher`.
3. Call `Matcher.find()` to search the text with the `Pattern` you defined.
4. Call `Matcher.group()` to display the characters that match your pattern.

Predefined Character Classes

Predefined Character	Character Class	Negated Character	Negated Class
`\d` (digit)	`[0-9]`	`\D`	`[^0-9]`
`\w` (word char)	`[a-zA-Z0-9_]`	`\W`	`[^a-zA-Z0-9_]`
`\s` (white space)	`[ \r\t\n\f\0XB]`	`\S`	`[^ \r\t\n\f\0XB]`

A number of character classes are used repeatedly. These classes are turned into predefined character classes. Classes exist to identify digits, word characters, and white space.

White-Space Characters

`\t`: Tab character

`\n`: New-line character

`\r`: Carriage return

`\f`: Form feed

`\x0B`: Vertical tab

Predefined Character Class: Examples

Target String	Jo told me 20 ways to San Jose in 15 minutes.

Pattern	Description	Text Matched
`\\d\\d`	Find any two digits.**	Jo told me 20 ways to San Jose in 15 minutes.
`\\sin\\s`	Find "in" surrounded by two spaces and then the next three characters.	Jo told me 20 ways to San Jose in 15 minutes.
`\\Sin\\S`	Find "in" surrounded by two non-space characters and then the next three characters.	Jo told me 20 ways to San Jose in 15 minutes.

** If there are additional matches in the current line, additional calls to `find()` will return the next match on that line.

Example:

```
Pattern p1 = Pattern.compile("\\d\\d");
        Matcher m1 = p1.matcher(t);
        while (m1.find()){
            System.out.println("Found: " + m1.group());
        }
```

Produces:

```
Found: 20
Found: 15
```

The code for this example can be found in the `StringExamples` project in the `PredefinedCharClassExample.java` file.

Quantifiers

Quantifier	Description
`*`	The previous character is repeated zero or more times.
`+`	The previous character is repeated one or more times.
`?`	The previous character must appear once or not at all.
`{n}`	The previous character appears exactly *n* times.
`{m,n}`	The previous character appears from *m* to *n* times.
`{m,}`	The previous character appears *m* or more times.
`(xx){n}`	This group of characters repeats *n* times.

Quantifiers enable you to easily select a range of characters in your queries.

Quantifier: Examples

Target String	Longlonglong ago, in a galaxy far far away

Pattern	Description	Text Matched
ago.*	Find “ago” and then 0 or all the characters remaining on the line.	Longlonglong ago, in a galaxy far far away
gal.{3}	Match “gal” plus the next three characters. This replaces “…” as used in a previous example.	Longlonglong ago, in a galaxy far far away
(long){2}	Find “long” repeated twice.	Longlonglong ago, in a galaxy far far away

The code for this example can be found in the `StringExamples` project in the `QuantifierExample.java` file.

Greediness

- A regular expression always tries to grab as many characters as possible.
- Use the `?` operator to limit the search to the shortest possible match.

Target String	Longlonglong ago, in a galaxy far far away.

Pattern	Description	Text Matched
`ago.*far`	A regular expression always grabs the most characters possible.	Longlonglong ago, in a galaxy far far away.
`ago.*?far`	The "?" character essentially turns off greediness.	Longlonglong ago, in a galaxy far far away.

A regular expression always tries to match the characters that return the most characters. This is known as the "greediness principle." Use the `?` operator to limit the result to the fewest characters needed to match the pattern.

The code for this example can be found in the `StringExamples` project in the `GreedinessExample.java` file.

Quiz

Which symbol means that the character is repeated one or more times?

a. *

b. +

c. .

d. ?

Answer: b

Boundary Matchers

Anchor	Description
`^`	Matches the beginning of a line
`$`	Matches the end of a line
`\b`	Matches the start or the end of a word
`\B`	*Does not* match the start or end of a word

Boundary characters can be used to match different parts of a line.

Boundary: Examples

Target String	it was the best of times or it was the worst of times

Pattern	Description	Text Matched
`^it.*?times`	The sequence that starts a line with "it" followed by some characters and "times", with greediness off	it was the best of times or it was the worst of times
`\\sit.*times$`	The sequence that starts with "it" followed by some characters and ends the line with "times"	it was the best of times or it was the worst of times
`\\bor\\b.{3}`	Find "or" surrounded by word boundaries, plus the next three characters.	it was the best of times or it was the worst of times

The code for this example can be found in the `StringExamples` project in the `BoundaryCharExample.java` file.

Quiz

Which symbol matches the end of a line?

a. *

b. +

c. $

d. ^

Answer: c

Matching and Groups

Target String	**george.washington@example.com**
Match 3 Parts	**(george).(washington)@(example.com)**
Group Numbers	**(1).(2)@(3)**
Pattern	**(\\S+?)\\.(\\S+?)@(\\S+)**

With regular expressions, you can use parentheses to identify parts of a string to match. This example matches the component parts of an email address. Notice how each pair of parentheses is numbered. In a regular expression, `group(0)` or `group()` matches all the text matched when groups are used. Here is the source code for the example:

```
public class MatchingExample {
  public static void main(String[] args){
      String email = "george.washington@example.com";

      Pattern p1 = Pattern.compile("(\\S+?)\\.(\\S+?)@(\\S+)");
      Matcher m1 = p1.matcher(email);
      if (m1.find()){
          System.out.println("First: " + m1.group(1));
          System.out.println("Last: " + m1.group(2));
          System.out.println("Domain: " + m1.group(3));
          System.out.println("Everything Matched: " + m1.group(0));
        }
    }
}
```

Using the `replaceAll` Method

Using the `replaceAll` method, you can search and replace.

```
public class ReplacingExample {
    public static void main(String[] args){
        String header = "<h1>This is an H1</h1>";

        Pattern p1 = Pattern.compile("h1");
        Matcher m1 = p1.matcher(header);
        if (m1.find()){
            header = m1.replaceAll("p");
            System.out.println(header);
        }
    }
}
```


You can do a search-and-replace by using the `replaceAll` method after performing a find. The output from the program is as follows:

```
<p>This is an H1</p>
```

Summary

In this lesson, you should have learned how to:

- Read data from the command line
- Search strings
- Parse strings
- Create strings by using a `StringBuilder`
- Search strings by using regular expressions
- Parse strings by using regular expressions
- Replace strings by using regular expressions

Practice 8-1 Overview: Parsing Text with `split()`

This practice covers using the `String.split()` method to parse text.

Practice 8-2 Overview: Creating a Regular Expression Search Program

This practice covers creating a program that searches through a text file using a regular expression.

Practice 8-3 Overview: Transforming HTML by Using Regular Expressions

This practice covers transforming the HTML of a file by using several regular expressions.