| [Print](javascript:window.print()) | [Close](javascript:parent.window.close()) |

**Course Transcript**

Java SE7 Professional: Syntax and Class Design

**Java Basic, Syntax, and Classes**

| [1. Introducing the Java Technology and Classes](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#t3) |

| [2. Java Fields, Constructors, Methods, and Packages](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#t5) |

| [3. Testing Classes and Creating Subclasses](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#t7) |

**Java Class Design**

| [1. Using Access Control and Overriding Methods](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#t9) |

| [2. Comparing Objects, Virtual Methods, and Casting](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#t11) |

| [3. Working with Abstract Classes](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#t13) |

| [4. Advanced Class Design](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#t15) |

| [5. Designing Java Classes](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#t17) |

Introducing the Java Technology and Classes

Learning Objectives

*After completing this topic, you should be able to*

* *identify the features of the Java technology*
* *create simple Java classes*

**1. Features of Java technology**

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, or JRE. A compiler, included in the Java SE Development Kit, or JDK, is used to convert Java source code to Java bytecode.

Graphic

*Java code is processed by the Java compiler and becomes Java bytecode. The Java bytecode provides a JRE for a Solaris OE, a LInux, and a Windows workstation.*

Oracle provides a complete line of Java technology products ranging from kits that create Java technology programs to emulation, or testing, environments for consumer devices, such as cellular phones.  
  
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.  
  
Among other Java technologies, each edition includes a Software Development kit – or SDK – that  
allows programmers to create, compile, and execute Java technology programs on a particular platform.

Graphic

*The figure illustrates the three Java technology product groups and their target device types. There is a table with three rows and six columns. The column headings are Servers, Desktop, Embedded, TV, Mobile, and Card. The row headings are Key APIs, Platform, and Language. In the Servers column of row 1, the API is Java EE, in the Desktop column it is Java FX, the Embedded cell is blank, the TV cell is BD-J and Java TV, the Mobile cell is MSA, and Card is blank. In the second row, the platform for Servers, Desktop, and Embedded is Java SE, Embedded, TV, and Mobile use Java ME, and Card uses Java Card. The third row lists Java Language for all the devices.*

Java ™ Platform, Standard Edition, or Java SE, is used to develop applets and applications that run within web browsers and on desktop computers, respectively. For example, you can use the Java SE SDK to create a word-processing program for a personal computer.   
  
Applets are launched inside a web browser, while applications are launched within an operating system.

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

Code

java -version

Supplement

*Selecting the link title opens the resource in a new browser window.*

**Job Aid**

Access the job aid [Java SE Platform Versions](javascript:doWindow('./jajl_jsep_a01_it_enust301_frame.html')) to learn more about the available Java versions.

You can use these steps to download and install the JDK:

Graphic

*The link to the Oracle download page is: http://www.oracle.com/technetwork/java/javase/downloads/index.html.*

* access the Oracle download page
* select the Java Platform, Standard Edition link
* download the version appropriate for your operating system
* follow the installation instructions, and
* set your PATH

There are standard patterns you need to follow  
when developing and 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, such as GlassFish, or other types of Java Middleware.  
  
Often, you will also need to manipulate information stored inside relational databases such as MySQL or Oracle Database.

Question

Which statements best describe Java technology and programs?

**Options:**

1. Java technology applications are compiled to Java bytecode
2. The java command can be used to determine your Java version
3. The JDK provides a runnable Java platform
4. Java applets and applications launch in the same manner

Answer

***Option 1:****Correct. Java technology applications are written in the Java programming language and compiled to Java bytecode. Bytecode is executed on the Java Platform.*

***Option 2:****Correct. If Java SE is installed on your system, you can detect the version number by running java -version.*

***Option 3:****Incorrect. The JRE provides a runnable Java platform. The compiler is included in the JDK.*

***Option 4:****Incorrect. Applets and applications differ in several ways. Primarily, applets are launched inside a web browser, while applications are launched within an operating system.*

**Correct answer(s):**

1. Java technology applications are compiled to Java bytecode  
2. The java command can be used to determine your Java version

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's very common for you to download and use Java libraries from non-Oracle sources within the Java community. For instance, you can use an Apache-developed JDBC library to access a relational database.

Graphic

*Different members of the Java community are listed. These include OpenJDK, GlassFish, NetBeans, Spring Framework, Apache, and java.net.*

The Java Community Process, or JCP is used to develop new Java standards. The benefits of the JCP include

* free downloads of all Java Specification Requests, or JSRs
* early access to specifications
* public review and feedback opportunities, and
* open membership

Note

*JCP can be found at http://jcp.org.*

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**

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 actively participate in the JCP.

**agility, and**

Slow-moving JSRs are now actively discouraged.

**governance**

The SE and ME expert groups are merging into a single body, to simplify governance.

OpenJDK is the open-source implementation of Java. Some of the benefits of using OpenJDK include

* JDK reference implementation
* new feature development, and
* openness to community contributions

Note

*OpenJDK can be found at http://openjdk.java.net/.*

OpenJDK is an example of a GPL licensed open-source project. As an open-source project, OpenJDK enables users to port Java to operating systems and hardware platforms of their choosing. Ports are underway for many platforms including FreeBSD, OpenBSD, NetBSD, and MacOS X. OpenJDK is also a basis for Oracle JDK.

Java OpenJDK is available for free, both in cost and in license. 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.  
  
The Java SE Support Program provides updates for end-of-life Java versions. The Oracle Java SE Advanced and Oracle Java SE Suite provide JRockit Mission Control, Memory Leak Detection, the Low Latency GC Suite, and the JRockit Virtual Edition Suite.

Additional information, such as product tutorials and documentation, sample code, and the Oracle learning library can be found in the Java section of http://www.oracle.com.

**2. Creating Java Classes**

A Java class is described in a text file with a .java extension. Java uses keywords to define important program components.

**package**

The package keyword defines where this class lives relative to other classes, and provides a level of access control.   
  
**Code**  
  
  
**Syntax**  
package <*package\_name*>;

**import**

The import keyword defines other classes or groups of classes 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. All the Java classes that are part of the package, or in the same folder as the current class, are included by default – there is no need to import them.   
  
**Code**  
  
  
**Syntax**  
import <*other\_packages*>;

**class**

The class keyword precedes the name of this class. It is a good practice for the name of the class and the file name to match when the class is declared public. However, the keyword public in front of the class keyword is a modifier, and is not required.   
  
**Code**  
  
  
**Syntax**  
public class *ClassName* {  
    <*variables*>;  
    <*constructor method(s)*>;  
    <*other methods*>;  
}

To run a Java program, you must define a main method. The main method is automatically called when the class is called from the command line.  
  
In this example, a simple Java class with a main method is created. Command-line arguments are passed to the program through the args[] array.

Code

public class Simple{  
    public static void main(String args[]){  
    }  
}

Java fields, or 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, or left to right. Every class or method declaration is enclosed in a code block.  
  
The Java compiler ignores all whitespace. Line indentation is not required, but makes code much more readable. The line indentation is four spaces – this is the default line indentation used by the NetBeans IDE.

Code

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

There are various primitive data types that you can use in Java, such as

**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 are byte, short, int, and long. Each type has its own range. The int type is the most commonly used. You can append uppercase or lowercase L to a number to specify a long number.  
  
The default integer value is 0.

**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.   
  
The double type is named because it provides twice the precision of float. Whereas float uses 32 bits to store data, double uses 64 bits. You can append uppercase or lowercase F to a number to specify a float number.  
  
The default floating point value is 0.0f.

**character, and**

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.  
  
The default character value is '\u0000'.

**boolean**

The boolean type can hold either true or false. The default boolean value is false.

If a value is not specified, a default value is used. Local variables, or variables declared within methods, don't have a default value. An attempt to use a local variable that has not been assigned a value will cause a compiler error. It's good practice to always supply any variable a default value.

The integer types are identified by the keywords

**byte**

The byte type has a length of 8 bits and a range of -2 to the power of 7 to 2 to the power of 7, minus 1 – or -128 to 127 – which is a range of 256 possible values. If you need to store people's ages, for example, a variable of type byte would work because byte types can accept values in that range.  
*Examples of literal values using the byte type are 2, -114, and 0b10.*

**short**

The short type has a length of 16 bits and a range of -2 to the power of 15 to 2 to the power of 15, minus 1.  
*Examples of literal values using the short type are 2 and -32699.*

**int, and**

The int type is the default type for integral literals. It has a length of 32 bits and a range of -2 to the power of 31 to 2 to the power of 31, minus 1. Integer literals are assumed by the compiler to be of type int unless you specify otherwise using an L to indicate long type.  
*Examples of literal values are 2, 147334778, and 123\_456\_678.*

**long**

The long type has a length of 64 bits and a range of -2 to the power of 63 to 2 to the power of 63, minus 1. When you specify a literal value for a long type, you put an uppercase L to the right of the value to explicitly state that it's a long type.  
*Examples of literal values are 2, -2036854775808L, and 1L.*

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

Code

long creditCardNumber = 1234\_5678\_9012\_3456L;  
long socialSecurityNumber = 999\_99\_9999L;  
float pi = 3.14\_15F;  
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;

When you are using literals you should be aware of the rules in relation to underscores - you can place them 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, or
* in positions where a string of digits is expected

The Java language is case sensitive. In Java, the variable creditCardNumber with upper case C and N is different from CREDITCARDNUMBER all in upper case. Convention indicates that Java variables and method names use lower camel case – that is, lower case for the first element of a variable name, with an upper case for the first letter of subsequent elements.

Code

long creditCardNumber = 1234\_5678\_9012\_3456L;

In Java SE 7 and later, binary literals can also be expressed using the binary system, by adding the prefix 0b or 0B to the number.

Code

// An 8-bit 'byte' value:  
byte aByte = (byte)0b00100001;  
  
// 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.

You use operators when you work with numbers. Operators have definitive precedence, but you can use parentheses to override precedence. The common operators are the arithmetic operators, the unary operators, and the simple assignment operator, which is a single equal sign.

The arithmetic operators are

* additive, represented by +, which is also used for String concatenation
* subtraction, represented by -
* multiplication, represented by \*
* division, represented by /, and
* remainder, represented by %

The unary operators are

* unary plus, represented by +, which indicates positive
* unary minus, represented by -, which negates an expression
* increment, represented by ++, which increments a value by 1
* decrement, represented by --, which decrements a value by 1, and
* logical compliment, represented by !, which inverts the value of a boolean

The code 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 and then combined using the + sign as a concatenation operator.

Code

public class Strings {  
    public static void main(String args[]){  
        char letter = 'a';  
        String string1 = "Hello";  
        String string2 = "World";  
        String string3 = "";  
        String dontDoThis = new String ("Bad Practice");  
        string3 = string1 + string2; // Concatenate strings  
        System.out.println("Output: " + string3 + " " + letter);  
    }  
}

This is the code output.

Code

Output: HelloWorld a.

Strings should always be initialized using the assignment operator = and text in quotation marks. The use of new to initialize a String is strongly discouraged.   
  
This is because "Bad Practice" is itself a String instance and has functionally identical to the object created by the constructor. If this statement appeared inside of a loop that was frequently invoked, there could be a lot of needless String instances created.

Graphic

*The relevant line is  
  
String dontDoThis = new String ("Bad Practice");*

Code

public class Strings {  
    public static void main(String args[]){  
        char letter = 'a';  
        String string1 = "Hello";  
        String string2 = "World";  
        String string3 = "";  
        String dontDoThis = new String ("Bad Practice");  
        string3 = string1 + string2; // Concatenate strings  
        System.out.println("Output: " + string3 + " " + letter);  
    }  
}

String is a class, not a primitive type. Instances of the class String represent sequences of Unicode characters.  
  
Some common String methods include

Code

public class StringOperations {  
    public static void main(String arg[]){  
        String string1 = "Hello";  
        String string2 = "World";  
        String string3 = "";  
        string3 = string1.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());  
    }  
}

**concat()**

The concat method is used to join strings together.  
  
The output is "string3: HelloWorld."

**length()**

The length method is used to return information about the number of characters in the string.  
  
The output is "Length: 5."

**substring(), and**

The substring method is used to return a new string that is a substring of the specified string.  
  
The output is "Sub: Hello."

**toUpperCase()**

The toUpperCase method is used to convert all the characters in the specified string to upper case.  
  
The output is "Upper: HELLOWORLD."  
*In the output, HELLOWORLD is all in uppercase.*

The code in this example demonstrates the use of the if-else statement in Java.  
  
The output of this code is False.

Code

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 different logic and conditional operators you can use in Java are equality and relational operators, conditional operators, and the type comparison operator, which uses instanceof to compare an object to a specified type.

The equality and relational operators are

* equal to, represented by ==
* not equal to, represented by !=
* greater than, represented by >
* greater than or equal to, represented by >=
* less than, represented by <
* less than or equal to, represented by <=

The conditional operators are

* Conditional-AND, represented by &&
* Conditional-OR, represented by ||, and
* Ternary, represented by ?:, and is shorthand for the if-then-else statement

Arrays are objects by default. All arrays support the methods of class Object. Array members include a final field length and a public method clone.   
  
You can define an array in Java by creating the array and then initializing each element separately.

Code

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";

Another method of defining an array is to use a single statement to create the array and initialize the elements.  
  
Each array is iterated through using the Java for-each construct. The loop defines an element that will represent each element of the array and the array to loop through.

Graphic

*The array definition is   
int[] numbers = {100, 200, 300};*

Code

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);  
        }  
    }  
}

There are a range of control flow statements that you can use in Java.

**for**

In this example, a for loop is created. A counter is initialized and incremented for each step of  
the loop. When the condition statement is met, the loop exits.   
  
**Code**  
public class ForLoop {  
  
    public static void main(String args[]){  
  
        for (int i = 0; i < 9; i++ ){  
            System.out.println("i: " + i);  
        }  
    }  
}

**while**

The while loop performs a test and continues if the expression evaluates to true. The while loop, in this example, iterates through an array using a counter.   
  
**Code**  
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++;  
        }  
    }  
}

**do-while**

In the do-while loop, the test occurs after the expression has run at least once.   
  
**Code**  
class DoWhileDemo {  
    public static void main(String[] args){  
        int count = 1;  
        do {  
            System.out.println("Count is: " + count);  
            count++;  
        } while (count <= 11);  
    }  
}

**switch**

In this example, a switch statement using a String has been created. Prior to version 7 of Java, only enums, byte, short, char, and int primitive data types could be used in a switch statement.   
  
**Code**  
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);  
    }  
}

Question

Select the field declarations that use underscore in a valid way within the intial field value.

**Options:**

1. float pi1 = 3\_.1415F;
2. int x1 = 5\_2;
3. long socialSecurityNumber = 999\_99\_9999L;
4. int x2 = 52\_;

Answer

***Option 1:****Incorrect. This is invalid because you can't put an underscore adjacent to a decimal point.*

***Option 2:****Correct. This is valid because underscores can be placed between digits.*

***Option 3:****Correct. This is valid because underscores can be placed between digits.*

***Option 4:****Incorrect. This is invalid because you can't put an underscore at the end of a literal.*

**Correct answer(s):**

2. int x1 = 5\_2;  
3. long socialSecurityNumber = 999\_99\_9999L;

**3. Summary**

In this topic you've identified the features of the Java technology and created simple Java classes.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#top)

Java Fields, Constructors, Methods, and Packages

Learning Objectives

*After completing this topic, you should be able to*

* *use Java fields, constructors, and methods*
* *use package and import statements*

**1. Java fields, constructors, and methods**

Variables, or the data associated with programs, such as integers, strings, arrays, and references to other objects are called *instance fields*, or simply *fields*.  
  
Methods are defined in a class and are functions that can be performed on any object. They are also referred to as *instance methods*.  
  
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*.  
  
An object is a representation in memory of  
a Java class. Constructors are functions called during the creation, or instantiation, of an object.

Java has naming conventions for the different programming objects, including

**class names**

Class names should be nouns in mixed case with the first letter of each internal word capitalized.

**methods, and**

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

**variable names**

Variable names should be short but meaningful. The choice of a variable name should be mnemonic, that is, 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.

Fields, constructors, and methods are written within the body of the class – following the  
class declaration. The curly braces define the class body.  
  
In general, for readability, fields that are used throughout the class are written, or declared, immediately after the class declaration.  
  
Constructors follow the fields. These are optional. If you don't write a constructor, one will be provided for you with no arguments, a no-arg constructor. However, it's good practice to write a constructor method – even a no-arg constructor.   
  
Methods follow constructors.

Syntax

public class *Classname*{  
 *Field 1  
    Field 2*    ...  
  
    public *Constructorname* () {  
    }  
  
    public int *methodName* () {  
        return *methodName*;  
    }  
}

There are many components that are combined to create a class.

Code

package com.example.domain;  
public class Employee {  
    public int empId;  
    public String name;  
    public String ssn;  
    public double salary;  
  
    public Employee () {  
    }  
  
    public int getEmpId () {  
        return empId;  
    }  
}

**public class Employee {**

*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, or a programmatic representation, of an Employee.*

**public int empId;  
   public String name;  
   public String ssn;  
   public double salary;**

*An Employee object defined using this model will contain values for: empId, name, Social Security Number (ssn), and salary.*

**public Employee () {  
   }**

*The constructor method in this class creates an instance of an object called Employee. A constructor method is unique in Java in that the return type of the method is an instance of the class, so constructors are always named the same as the class, and do not declare a return type. You can declare more than one constructor.*

**public int getEmpId () {  
       return empId;  
   }**

*Methods are defined in a class and are functions that can be performed on an object. For example the getEmpID method will return a value for empID.*

When a class has data fields, a common practice is to provide methods for storing data, which are known as setter methods or mutators.  
  
Getter methods or accessors are used to retrieve data from the fields.  
  
The convention is to use set and get and the name of the field with the first letter of the field name capitalized. Most modern Integrated Development Environments, or IDEs, provide an easy way to automatically generate the required methods for you.

Graphic

*An example of the setter method is setEmpId. An example of the getter method is getEmpId.*

Code

package com.example.domain;  
public class Employee {  
// fields  
    public void setEmpId(int empId) { this.empId = empId; }  
    public int getEmpId() { return empId; }  
    public void setName(String name) { this.name = name; }  
    public String getName() { return name; }  
    public void setSsn(String ssn) { this.ssn = ssn; }  
    public void setSalary(double salary){ this.salary = salary;}  
    public double getSalary() { return salary; }  
}

The set methods use the this keyword. 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 compiler would complain: "Assignment to self."  
  
In this example, you could use the setName method to change the employee name and the setSalary method to change the employee salary.

Graphic

*The relevant code snippets are this.empId, setName, and setSalary.*

Code

package com.example.domain;  
public class Employee {  
// fields  
    public void setEmpId(int empId) { this.empId = empId; }  
    public int getEmpId() { return empId; }  
    public void setName(String name) { this.name = name; }  
    public String getName() { return name; }  
    public void setSsn(String ssn) { this.ssn = ssn; }  
    public void setSalary(double salary){ this.salary = salary;}  
    public double getSalary() { return salary; }  
}

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.  
  
All of the fields declared in the class are provided memory space and initialized to their default values. The constructor method is then called. If the memory allocation and constructor are successful, a reference to the object is returned as a result. In the example above, the reference is assigned to a variable called emp.

Graphic

*The relevant code is the new keyword.*

Code

/\* 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);

To store values in the Employee object instance, you could just assign values to each field. However, this is not a good practice and negates the principle of encapsulation. Instead, you should use methods to set the value of each data field.

Graphic

*The relevant code is  
emp.setEmpId(101);   
emp.setName("John Smith");  
emp.setSsn ("011-22-3467");  
emp.setSalary(120345.27);*

Code

/\* 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  
// use a method instead  
emp.setEmpId(101);   
emp.setName("John Smith");  
emp.setSsn ("011-22-3467");  
emp.setSalary(120345.27);

A constructor is a pseudo-method that creates an object. In the Java programming language, constructors are methods with the same name as their class used to create an instance of an object. Constructors are invoked using the new keyword.  
  
The value returned from the constructor is a reference to a Java object of the type created and constructors can take parameters.

Code

Employee emp = new Employee();

Question

Which declaration demonstrates the application of good Java naming conventions?

**Options:**

1. public class repeat { }
2. public void Screencoord (int x, int y) { }
3. private int XCOORD;
4. public int calcOffset (int xCoord, int yCoord) { }

Answer

***Option 1:****Incorrect. This declaration uses a lowercase first letter and a verb for a class name – class names should be nouns with an initial capital letter.*

***Option 2:****Incorrect. This declaration is a method name that uses a first character uppercase, rather than a lower camel case – the first letter lowercase and the first letter of each name element in uppercase. Screencoord also sounds like a noun rather than a verb.*

***Option 3:****Incorrect. This declaration is questionable because it appears to be a constant as it is uppercase, yet there is no assigned value.*

***Option 4:****Correct. This declaration follows the Java naming convention. It clearly identifies its intent – it will calculate the offset between the two coordinates passed as arguments.*

**Correct answer(s):**

4. public int calcOffset (int xCoord, int yCoord) { }

**2. Package and import statements**

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. The namespace creates a logical collection of things, like a directory hierarchy. In this way, packages are more than just a convenience.  
  
In Java, a package is a group of class types. There can be only one package declaration for a class.   
  
It's good practice to always use a package declaration. The package declaration is always at the top of the class.

The given code refers to a class using its fully qualified namespace. However, this often requires lots of extra typing.  
  
Instead, Java provides the import statement to allow you to declare that you want to reference a class in another package.

Code

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

Note

*Modern IDEs, such as NetBeans and Eclipse, will automatically search for and add import  
statements for you. In NetBeans, for example, you can press****Ctrl+Shift+I****to fix imports in your code.*

The import keyword is used to identify classes you wish to reference in your class. The import statement offers various advantages:

* it provides a convenience for identifying classes you wish to reference in your class   
    
  **Code**  
  import java.util.Date;
* you can import a single class, or an entire package, but it's good practice to use the full package and class name rather than the wildcard \* to avoid class name conflicts, and   
    
  **Code**  
  import java.util.\*;
* you can include multiple import statements   
    
  **Code**  
  import java.util.Date;  
  import java.util.Calendar;

Import statements follow the package declaration and precede the class declaration. They are not required statements and by default your class will always import java.lang.\*.   
  
In addition, you do not need to import classes that are in the same package. In this example, the Manager class is already imported as part of the specified package.

Code

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

The Java language uses pass-by-value for all parameter passing. This means that the  
argument on the right side of the equals sign is evaluated, and the value of the argument is  
assigned to the left side of the equals.  
  
For Java primitives, this is straight forward. Java does not pass a reference to a primitive, such as an integer, but rather a copy of the value.  
  
For instance, if you have two variables – x with the value 3 and y with the value x, the value of x is copied and passed to y.  
  
If the value of x changed to 5, the value of y  
remains unchanged.

Code

int x = 3;  
int y = x;

For Java objects, the value of the right-side of an assignment is a reference to memory that stores a Java object. The reference is some address in memory.   
  
After the assignment, the value of y is the same value as x, which is a reference to the same Employee object.  
  
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.   
  
An object is a class instance or an array. The reference values are pointers to these objects. The null value is a special reference which refers to no object.

Code

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

Whenever a new value is created, as in the foo method, a new object is created and the value of the parameter is changed to the new value.  
  
The value of x is simply copied into the parameter for the method, so x remains unchanged during and after the execution of the foo method.

Graphic

*The new value is created using the statement  
e = new Employee();*

Code

Employee x = new Employee();  
foo(x);  
  
public void foo(Employee e) {  
    e = new Employee();  
    e.setSalary (1\_000\_000\_00);   
}

After the call is made to the foo method, a value is returned from the setSalary method.  
  
The value of the reference x is unchanged, and the value returned would be the same as before the call. This is because you used the new keyword to construct a new object and assigned that object instance to e.   
  
The setSalary method is operating on the reference e – not on the object reference passed in to the method.  
  
If you were to remove the new statement, the e.setSalary method would be operating on the x reference.

Graphic

*The call to the foo method is   
foo(x);  
  
The new statement is   
e = new Employee();*

Code

Employee x = new Employee();  
foo(x);  
  
public void foo(Employee e) {  
    e = new Employee();  
    e.setSalary (1\_000\_000\_00);   
}

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

Code

javac –classpath <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, use the Java interpreter, java.

Code

java –classpath <path to other classes> <package name>.<classname>

The CLASSPATH variable is used by both the Java compiler and the Java interpreter, runtime. The classpath can include

* a list of directory names, that are separated by semicolons in Windows and colons in UNIX, or
* a .zip or .jar file name that is fully qualified with its path name

When you are using a list of directories, you should note that the classes are in a package tree relative to any of the directories on the list. And the classpath includes the current working directory (.) by default.  
  
When working with zipped files, you should ensure that the classes in these files are zipped with the path names that are derived from the directories formed by their package names.  
  
The directory containing the root name of the package tree must be added to the classpath. It's good practice to put classpath information in the command window or even in the Java command, rather than hard-coding it in the environment.

Assume that you have created this class and wish to compile and run the application. You can use javac to compile it.  
  
The advantage of an IDE like NetBeans is that management of the classpath, compilation, and running the Java application are handled using the tool.

Graphic

*You use this command to compile the application   
javac –d D:\test D:\test\com\example\HelloWorld.java*

Code

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] + "!");  
        }  
    }  
}

To run the application, you would use the interpreter and the fully qualified classname.

Graphic

*The first run command is   
java –classpath D:\test com.example.HelloWorld  
  
The output is  
Hello World!  
  
The second run command is   
java –classpath D:\test com.example.HelloWorld Tom  
  
The output is   
Hello Tom!*

Question

Which statements best describe packages?

**Options:**

1. They group classes together
2. They are implemented as a folder
3. There can be multiple package declarations for a class
4. The package declaration can be anywhere in a class

Answer

***Option 1:****Correct. In Java, a package is a group of class types. Packages create a namespace, a logical collection of things, like a directory hierarchy.*

***Option 2:****Correct. A package is implemented as a folder, and like a folder, provides a namespace to a class.*

***Option 3:****Incorrect. There can only be one package declaration for a class.*

***Option 4:****Incorrect. The package declaration is always at the top of the class.*

**Correct answer(s):**

1. They group classes together  
2. They are implemented as a folder

**3. Creating Java classes**

You now want to use the NetBeans IDE to create an Employee class, create a class with a main method to test the Employee class, compile and run your application, and print the results to the command-line output.

Graphic

*NetBeans is open. By default, it contains a menu bar, toolbar, and navigation pane that's divided into three tabs: Projects, Files, and Services.*

To create a new project called EmployeeTest in NetBeans, with an EmployeeTest class and main method, you need to follow these steps:

* click **File - New Project**   
  *Some options in the File menu are New Project, Open Project, Open Recent Project, Open File, Open Recent Files, Project Group, and Import Project.*
* select **Java** from Categories, select **Java Application** from Projects, click **Next**, and   
  *The New Project dialog box lists 2 steps. The first step is Choose Project. In the Choose Project section, you can choose Categories and Projects. The available categories are Java, Maven, NetBeans Modules, and Samples. The Projects options are Java Application, Java Desktop Application, Java Class Library, Java Project with Existing Sources, and Java Free-Form Project.*
* enter the values for the project name and project location and click **Finish**on the New Java Application screen   
  *The second step of New java Application is Name and Location.  
    
  The page has a Project Name field with the value EmployeePractice, a Project Location field with the value C:\labs\professional\labs\02-Review\practices, and a Project Folder field with the value C:\labs\professional\labs\02-Review\practices\EmployeeTest. There are three checkboxes: Use Dedicated Folder for Storing Libraries, Create Main Class (selected), and Set as Main Project (selected).*

NetBeans has saved you a great deal of typing by creating a class called EmployeeTest, including the package name of com.example, and writing the skeleton of the main method for you.

Now you need to set the Source/Binary format to JDK 7.   
  
To do this, you access the project properties and select **JDK 7** from the Source/Binary Format drop down list.

Now you can create another package called com.example.domain by using these steps:

* right-click **EmployeeTest** in the **Project** tab and select **New - Java Package**, and   
  *The options in the New submenu are Java Main Class, Java Class, Java Package, Java Interface, JPanel Form, JFrame Form, Entity Class, Entity Classes from Database, and Other.*
* specify com.example.domain as the package name and then click **Finish**   
  *The New Java Package dialog box has 2 steps: Choose File Type, and Name and Location, which is selected. There is a Package Name field, a Project field, a Location drop-down list, and a Created Folder field.*

The icon beside the package name is gray in the project – this is because the package does not contain any classes yet.

Graphic

*The domain package has been added to the Employee Test folder structure.*

Now create a new Java class called Employee in the com.example.domain package using these steps:

* right-click the **com.example.domain** package, select **New - Java Class**, and   
  *The options in the New submenu are Java Main Class, Java Class, Java Package, Java Interface, JPanel Form, JFrame Form, Entity Class, Entity Classes from Database, and Other.*
* enter Employee in the Class Name field and click **Finish** to create the class   
  *The New Java Class dialog box has 2 steps: Choose File Type, and Name and Location, which is selected. There is a Class Name field, a Project field, a Location drop-down list, a Package field, and a Created File field.*

NetBeans has generated a class with the name Employee in the package com.example.domain.

You can follow these steps to code the Employee class:

Graphic

*The code already created is   
package com.example.domain;  
public class Employee*

* add data fields for Employee id, Employee name, Employee Social Security Number, and Employee salary respectively, and   
  *The code to add the data fields is  
     public int empId;  
     public String name;  
     public String ssn;  
     public double salary;*
* add a constructor   
  *The code to add a constructor is   
  public Employee() {  
     }*

You can create getter and setter methods for each field using these steps:

* click in your class where you want to insert the methods, and then right-click and select **Getter and Setter** from the Generate menu   
  *The options are Constructor, Getter, Setter, Getter and Setter, equals() and hashCode(), toString(), Delegate Method, and Override Method.*
* view the fields for which you can generate getter and setter methods, and   
  *The Generate Getters and Setters dialog box displays the Employee class and lists its fields – empId: int, name : String, salary : Double, and ssn : String. Each option has an associated checkbox.*
* select the fields that you want to generate getter and setter methods for, and click **Generate** to insert the code   
  *You select empId: int, name : String, salary : Double, and ssn : String.*

The code for the getter and setter methods for each field is automatically generated for the Employee class.

Graphic

*The code for the getter and setter methods is  
  
   public int getEmpId() {  
       return empId;  
   }  
  
   public void setEmpId(int empId) {  
       this.empId = empId;  
   }  
  
   public String getName() {  
       return name;  
   }  
  
   public void setName(String name) {  
       this.name = name;  
   }  
  
   public double getSalary() {  
       return salary;  
   }  
  
   public void setSalary(double salary) {  
       this.salary = salary;  
   }  
  
   public String getSsn() {  
       return ssn;  
   }  
  
   public void setSsn(String ssn) {  
       this.ssn = ssn;*

To continue, you can use these steps to write the code in the EmployeeTest main class to test your Employee class:

* add an import statement to your class for the Employee object   
  *The code to add an import statement is   
  import com.example.domain.Employee;*
* create an instance of your Employee class in the main method of EmployeeTest   
  *The code to add an instance is   
  Employee emp = new Employee ();*
* use the Employee object instance to add data to the object using the setter methods, and   
  *The code to add the data is  
         emp.setEmpId(101);  
         emp.setName("Jane Smith");  
         emp.setSsn ("012-34-5678");  
         emp.setSalary(120\_345.27);*
* in the body of the main method, use the System.out.println method to write messages to the console output and save the EmployeeTest class   
  *The code to write messages is  
  System.out.println ("Employee id:         " + emp.getEmpId());  
         System.out.println ("Employee name:       " + emp.getName());  
         System.out.println ("Employee Soc Sec #:  " + emp.getSsn());  
         System.out.println ("Employee salary:     " + emp.getSalary());*

Now you can build and run your project. To do this, you access the Compiling properties of the project.  
  
**Compile on Save** is selected by default. This means that as soon as you saved the Employee and EmployeeTest classes, they were compiled.

Graphic

*The Project Properties - EmployeePractice dialog box has a Categories pane, listing the categories Sources, Libraries, Build, Run, Application, and Formatting. Build has  the subcategories Compiling, Packaging, and Documenting.  
  
The Compiling options include Compile on Save, Generate Debugging Info, Reported Uses of Deprecated APIs, and Track Java Dependencies.*

When you select **Run**, you'll notice that the Main Class is com.example.EmployeeTest. This is the class the Java interpreter will execute.   
  
The next field is Arguments, used for passing arguments to the main method.

Graphic

*The Run options include a Configuration drop-down list box, which has an associated New button, and Main Class, Arguments, and Working Directory text boxes.*

When you run the project, this is the output that is expected, if there are no errors.

Code

run:  
Employee id:        101  
Employee name:      Jane Smith  
Employee Soc Sec #: 012-34-5678  
Employee salary:    120345.27  
BUILD SUCCESSFUL (total time: 1 second)

**4. Summary**

In this topic you've used Java fields, constructors, and methods, as well as package and import statements.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#top)

Testing Classes and Creating Subclasses

Learning Objectives

*After completing this topic, you should be able to*

* *test Java classes*
* *create Java subclasses*

**1. Testing Java classes**

The term encapsulation means to enclose in a capsule, or wrap something around an object to cover it.  
  
Encapsulation is defined by the Java 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 would be the steering wheel of a car. When you drive a car, you don't worry about how the steering wheel implements a turn right or turn left function. As long as the car steers properly when you turn the wheel, the steering wheel encapsulates the functions you need – the implementation does not concern you.

In object-oriented programming, encapsulation covers, or wraps, the internal workings of a Java object. Some encapsulation features include

* hiding data variables, or fields, from the user of the object
* using methods to provide services to the objects users, but hiding the implementation of the methods, and
* modifying implementations without any impact on a user, provided that the services do not change

Suppose you're creating a model of a generic employee, or an employee record in a database.   
  
To create this model, you will need to create an object to encapsulate data fields and operations such as

**Employee ID**

You can use the Employee ID as a unique identifier for the employee.

**Name**

You can use the Name field to store the employee name.

**Social Security Number**

You can use a Social Security Number data field as an identifier for American employees. You may need to use an alternative identifier for employees of other nationalities.

**Salary**

You should record the Salary details for each employee.

**Change Name, and**

You can use a Change Name operation to allow changes to the employee name in the event of marriage or divorce.

**Raise Salary**

You can use the Raise Salary operation to specify if an employee qualifies for an increase based on any specified criteria.

One way to hide implementation details is to declare all of the fields private.  
  
In this example, the fields custID, name, and amount are marked private, making them invisible outside of the methods in the class itself.

Code

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  
}

You can use public and private access modifiers to apply different levels of access to different elements:

Code

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  
}

* the public keyword, applied to fields and methods, allows any class in any package to access the field or method   
    
  **Code**  
  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  
  }
* the private keyword, applied to fields and methods, allows access only to other methods within the class itself, and   
    
  **Code**  
  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   
    
  **Code**  
  // Called when a withdrawal exceeds the available funds  
  private void applyOverdraftFee () {  
      amount += fee;  
  }

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

Code

package com.example.model;  
    public class Employee {  
        private int emp\_id;  
        private String name;  
        private String ssn;  
        private double salary;  
            //... constructor and methods  
    }  
  
    public Employee () {  
    }  
    public int getEmpId () {  
        return empId;  
    }  
}

Although the fields are now hidden using private access, there are three main issues with the current Employee class:

Code

public void setEmpId(int empId) { this.empId = empId; }  
public int getEmpId() { return empId; }  
public void setName(String name) { this.name = name; }  
public String getName() { return name; }  
public void setSsn(String ssn) { this.ssn = ssn; }  
public void setSalary(double salary){ this.salary = salary;}  
public double getSalary() { return salary; }

* the setter methods, which are currently public access, allow any other class to change the ID, SSN, and salary
* the current class does not really represent the operations defined in the original Employee class design, and
* the original model for the Employee class had a change name and increase salary operation

Just as fields should clearly define the type of data they store, methods should clearly identify  
the operations that they perform.   
  
One of the easiest ways to improve the readability of your code is to write method names that clearly identify the method's function. Another good practice when coding methods is to hide as many of the implementation details as possible.

Code

package com.example.model;  
    public class Employee {  
        // fields  
            public void setEmpId(int empId) { this.empId = empId; }  
            public int getEmpId() { return empId; }  
            public void setName(String name) { this.name = name; }  
            public String getName() { return name; }  
            public void setSsn(String ssn) { this.ssn = ssn; }  
            public void setSalary(double salary){ this.salary = salary;}  
            public double getSalary() { return salary; }  
   }

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 – that is, no changes should be allowed.   
  
The Employee model that was previously defined had only two operations – one for altering an employee name and one for increasing an employee's salary.

Code

package com.example.model;  
    public class Employee {  
        // fields  
            public void setEmpId(int empId) { this.empId = empId; }  
            public int getEmpId() { return empId; }  
            public void setName(String name) { this.name = name; }  
            public String getName() { return name; }  
            public void setSsn(String ssn) { this.ssn = ssn; }  
            public void setSalary(double salary){ this.salary = salary;}  
            public double getSalary() { return salary; }  
   }

To refine the Employee class, the first step is to remove the setter methods and create methods that clearly identify their purpose.   
  
This code contains two methods – setName to change an employee name and raiseSalary to increase an employee salary.  
  
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 could do further checking as necessary.

Code

package com.example.model;  
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;  
    }  
}

Finally, because the class no longer has setter methods, you need a way to set the initial value of the fields.  
  
You can do this by passing 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. Removing the setter methods and replacing the no-arg constructor also guarantees that an instance of Employee has immutable ID and SSN fields.

Code

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) { ... }  
}

Question

Which statements best describe encapsulation in Java?

**Options:**

1. Data variables are visible to the user of the object
2. Methods provide an explicit service to the user of the object but hide the implementation
3. As long as the services do not change, the implementation can be modified without impacting the user
4. Modifying the implementation means that code using the object must be changed.

Answer

***Option 1:****Incorrect. Data variables, or fields, are hidden from the user of the object.*

***Option 2:****Correct. Encapsulation means to enclose in a capsule or to wrap something around an object to cover it. Methods provide an explicit service to the user of the object but hide the implementation.*

***Option 3:****Correct. With encapsulation, as long as the services provided by the object remain the same, the implementation can be modified without impacting the user.*

***Option 4:****Incorrect. 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.*

**Correct answer(s):**

2. Methods provide an explicit service to the user of the object but hide the implementation  
3. As long as the services do not change, the implementation can be modified without impacting the user

**2. Creating Java 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.  
  
The Manager class 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 you defined Java classes this way, there would be a lot of redundant  
coding.

Code

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...

In an object-oriented language like Java, subclassing is used to define a new class in terms of an existing one.   
  
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.  
  
For example, the Manager class gets empId, name, SSN, salary and all of the setter and getter methods from Employee. It is important to remember that although Manager specializes Employee, a Manager still is an Employee.

Graphic

*The Manager class is a subclass of the Employee class.  
  
The fields of the Employee class are:   
       private int empId;  
       private String name;  
       private String ssn;  
       private double salary;  
  
Its methods are:  
   public Employee(int empId, String name, String                                             ssn, double salary){}  
   public void setName(String newName){}  
   public void raiseSalary(double increase){}      
  
The fields of the Manager class are:       
        private String deptName;  
  
Its methods are:  
   public Manager(int empId, String name, String                                             ssn, double salary                                                            String dept){}  
   public String getdeptName(){}*

Note

*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.Java.*

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 Manager is an Employee, it follows that Manager has all of the same attributes and operations of an Employee.  
  
However, the Manager class has to declare its own constructor as it is not inherited from the parent class.   
  
The constructor that Manager declares, calls the constructor of its parent class, Employee, using the super keyword. This is done to set the value of all the Employee fields.

Graphic

*The line of code to create to subclass is  
public class Manager extends Employee.  
  
The constructor that is declared is   
public Manager (int empId, String name,  
String ssn, double salary, String dept)*

Code

package com.example.domain;  
    public class Manager extends Employee {  
        private String deptName;  
            public Manager (int empId, String name,  
                        String ssn, double salary, String dept) {  
            super (empId, name, ssn, salary);  
            this.deptName = dept;  
            }  
    public String getDeptName () {  
        return deptName;  
    }  
}

Manager is a specialization of Employee, so constructing a Manager requires a department name, which is assigned to the deptName field.  
  
You can also consider other methods that you can use in this model of a manager. For example, you can add Employees to this manager and then using an array or a Collection to keep track of them.

Graphic

*The relevant line of code is  
  
this.deptName = dept;*

Code

package com.example.domain;  
    public class Manager extends Employee {  
        private String deptName;  
            public Manager (int empId, String name,  
                        String ssn, double salary, String dept) {  
            super (empId, name, ssn, salary);  
            this.deptName = dept;  
            }  
    public String getDeptName () {  
        return deptName;  
    }  
}

Although a subclass inherits all of the methods and fields from a parent class, it does not inherit constructors. If the parent class does not include a constructor, a no-arg constructor is provided by default, as otherwise there would be no way to create an object of that type.  
  
If the parent class declares a constructor with arguments, the subclass must provide its own implementation.  
  
In its constructor, Manager calls the constructor of Employee. This way, when a Manager instance is created, the Manager must have a department to manage. The super keyword is used to call a parent's constructor, method, or access a non-private parent field.

Code

super (empId, name, ssn, salary);

Note

*The super call of the parent's constructor must appear first in the constructor.*

Even though the Manager.java file does not contain all of the methods from the Employee.java class, they are included in the definition of the object.   
  
Here are some considerations when creating a Manager object:

* creating a Manager object is the same as creating an Employee object   
    
  **Code**  
  Manager mgr = new Manager (102, "Barbara Jones",  
  "107-99-9078", 109345.67, "Marketing");
* all of the Employee methods are available to Manager, and   
    
  **Code**  
  mgr.raiseSalary (10000.00);
* the Manager class defines a new method to get the Department Name and you can call methods that are specific to the Manager class   
    
  **Code**  
  String dept = mgr.getDeptName();

Question

Which statements most accurately describe subclasses?

**Options:**

1. A subclass is a specialization of the superclass
2. The subclass contains less code than the parent class
3. The keyword extends is used to create a subclass
4. Private fields and methods from the superclass are part of the subclass

Answer

***Option 1:****Correct. When an existing class is subclassed, the new class is a specialization of the superclass.*

***Option 2:****Incorrect. 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.*

***Option 3:****Correct. In an object-oriented language, subclassing is used to define a new class in terms of an existing one. The keyword extends is used to create a subclass.*

***Option 4:****Incorrect. All of the non-private fields and methods from the superclass are part of the subclass.*

**Correct answer(s):**

1. A subclass is a specialization of the superclass  
3. The keyword extends is used to create a subclass

Polymorphism means "many forms." In Java and object-oriented programming languages, polymorphism refers to the fact that while an object has only one form, a variable can have many forms because it can refer to an object using either its actual form, or a parent form.   
  
This is useful when creating a general purpose business method – for example, to be able to 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.  
  
Another example is this assignment that shows that an Employee can be Manager.

Code

Employee emp = new Manager();

In this case, however, the given assignment will generate an error.  
  
The Java compiler recognizes the variable emp only as an Employee object, and because the Employee class does not have a setDeptName method, it shows an error.

Code

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

Your design may call for several methods in the same class with the same name, but with different arguments.  
  
For example, you want to design methods with the same intent, or method name, like print, to print out several different types.  
  
However, this is time-consuming and doesn't meet the goals of object-oriented programming.  
  
Fortunately, Java permits you to reuse a method name for more than one method. This process is called *overloading*.

Code

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

When overloading methods, the argument lists must be different – either in order, number or type. And the return types can be different.  
  
In this example, the given methods contain the same argument list, and differ only in the return type used. This is not allowed.

Code

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

A variation of method overloading is when you need a method that takes any number of arguments of the same type. For example, you want to have methods to calculate an average for 2, 3, or 4 integers.  
  
Each of these methods would perform a similar type of computation – the average of the arguments passed in.  
  
Because these three overloaded methods share the same functionality, it would be useful to collapse these methods into one method.

Code

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 the three methods into one method, and provides for any number of arguments using the varargs or variable arguments feature.  
  
The nums argument is actually an array object of type int[]. This permits the method to iterate over and allow any number of elements.  
  
The average method shown here will take any number of integer arguments, from 1 to as many as you want. The notation(int... nums)converts the list of arguments passed to the average method into array object of type int.   
  
This notation takes some getting used to, and isn't very readable. Another way of solving this problem would be to create a method that requires an int [].

Graphic

*The average method is  
  
public float average(int... nums)*

Code

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);  
    }  
}

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 subclass, the language does provide features that enable multiple classes to implement the features of other classes.  
  
Single inheritance does not prevent continued refinement and specialization of classes. For example, a Manager can have Employees, and a Director has a budget and can approve expenses.

Graphic

*The Employee class is the parent class of the Manager class. The Director, Engineer, and Admin classes are also linked to the Employee class.*

Note

*You should use inheritance only when it's completely valid or unavoidable. Check appropriateness with the "is a" phrase. For example, the phrase "a shirt is a piece of clothing" expresses a valid inheritance link, while the phrase "a hat is a sock" does not.*

**3. Working with subclasses**

You want to create subclasses of Employee, including Manager, Engineer, and Administrative assistant – Admin.  
  
You also need to create a subclass of Manager called Director, and create a test class with a main method to test your new classes.

You begin by adding encapsulation to the existing Employee class using these steps:

Code

package com.example.domain;  
public class Employee {  
    public int empId;  
    public String name;  
    public String ssn;  
    public double salary;  
  
    public Employee() {  
    }  
    public int getEmpId() {  
        return empId;}     
    public void setEmpId(int empId) {  
        this.empId = empId;}  
    public String getName() {  
        return name;}  
    public void setName(String name) {  
        this.name = name;}  
    public double getSalary() {  
        return salary;}  
    public void setSalary(double salary) {  
        this.salary = salary;}  
    public String getSsn() {  
        return ssn;}  
    public void setSsn(String ssn) {  
        this.ssn = ssn;}  
}

* make the fields of the Employee class private   
    
  **Code**  
  package com.example.domain;  
  public class Employee {  
      private int empId;  
      private String name;  
      private String ssn;  
      private double salary;  
    
      public Employee() {  
      }  
      public int getEmpId() {  
          return empId;}     
      ...  
  }
* replace the no-arg constructor in Employee with a constructor that takes empId, name, ssn, and salary   
    
  **Code**  
  package com.example.domain;  
  public class Employee {  
      private int empId;  
      private String name;  
      private String ssn;  
      private double salary;  
    
      public Employee(int empId, String name, String ssn, double salary) {  
          this.empId = empId;  
          this.name = name;  
          this.ssn = ssn;  
          this.salary = salary;  
      }  
    ...  
  }
* remove all the setter methods except setName, and   
    
  **Code**  
  package com.example.domain;  
  public class Employee {  
      private int empId;  
      private String name;  
      private String ssn;  
      private double salary;  
    
      public Employee(int empId, String name, String ssn, double salary) {  
          this.empId = empId;  
          this.name = name;  
          this.ssn = ssn;  
          this.salary = salary;  
      }  
      public int getEmpId() {  
          return empId;}     
      public String getName() {  
          return name;}  
      public void setName(String name) {  
          this.name = name;}  
      public double getSalary() {  
          return salary;}  
      public String getSsn() {  
          return ssn;}  
     }
* add a method named raiseSalary with a parameter of type double named increase to increment the salary, and then save Employee.java   
    
  **Code**  
      public void raiseSalary(double increase) {  
      salary += increase;  
      }  
  }

Now you need to create a subclass of Employee called Manager. You begin by creating the Manager class.

Code

package com.example.domain;  
  
public class Manager

Then you follow these steps:

Code

package com.example.domain;  
  
public class Manager

* modify the class to subclass Employee   
    
  **Code**  
  package com.example.domain;  
    
  public class Manager extends Employee {
* add a private String field to store the department name in a field called deptName   
    
  **Code**  
      private String deptName;
* add a constructor that sets the value of the fields inherited from the parent class, and   
    
  **Code**  
      public Manager(int empId, String name, String ssn, double salary, String deptName) {  
          super(empId, name, ssn, salary);  
          this.deptName = deptName;  
      }
* add a getter method for deptName and save the Manager class   
    
  **Code**  
      public String getDeptName() {  
          return deptName;  
      }  
  }

Note

*Constructors aren't inherited from the parent class, so you need to add a constructor that sets the value of the fields inherited from the parent class –the easiest way is to write a constructor that calls the parent constructor using the super keyword.*

You now create another subclass of Employee:Engineer in the com.example.domain package. No fields or methods are needed.  
  
Because an Engineer is an Employee, you need to add a constructor for the class, which will construct the class as an instance of an Employee.

Code

package com.example.domain;  
  
public class Engineer extends Employee {  
  
    public Engineer(int empId, String name, String ssn, double salary) {  
        super(empId, name, ssn, salary);  
    }  
}

Note

*Use the super keyword as you did in the Manager class.*

Similarly, you can create an Admin subclass and then save the classes.

Code

package com.example.domain;  
  
public class Admin extends Employee {  
  
    public Admin(int empId, String name, String ssn, double salary) {  
        super(empId, name, ssn, salary);  
    }  
}

To continue, you create a subclass of Manager called Director in the com.example.domain package:

Code

package com.example.domain;  
  
public class Director extends Manager {

* add a private field to store a double value budget   
    
  **Code**  
      private double budget;
* add the appropriate constructors for Director – use the super keyword to construct a Manager instance and set the value of budget, and   
    
  **Code**  
      public Director(int empId, String name, String ssn, double salary, String department, double budget) {  
          super(empId, name, ssn, salary, department);  
          this.budget = budget;  
      }
* create a getter method for budget, before saving the class   
    
  **Code**  
      public double getBudget() {  
          return budget;  
      }  
  }

Now you can test your subclasses by modifying the existing EmployeeTest class. You can do this by adding in the additional import statements that are required and removing the code that creates an instance of the "Jane Smith" Employee.

Code

package com.example;  
  
import com.example.domain.Employee;  
import com.example.domain.Engineer;  
import com.example.domain.Manager;  
import com.example.domain.Admin;  
import com.example.domain.Director;  
import java.text.NumberFormat;  
  
public class EmployeeTest {  
  
    public static void main(String[] args) {

To continue, you need to use these steps:

Code

package com.example;  
  
import com.example.domain.Employee;  
import com.example.domain.Engineer;  
import com.example.domain.Manager;  
import com.example.domain.Admin;  
import com.example.domain.Director;  
import java.text.NumberFormat;  
  
public class EmployeeTest {  
  
    public static void main(String[] args) {

* create an instance of an Engineer   
    
  **Code**  
          Engineer eng = new Engineer(101, "Jane Smith", "012-34-5678", 120\_345.27);
* create an instance of a Manager   
    
  **Code**  
          Manager mgr = new Manager(207, "Barbara Johnson", "054-12-2367", 109\_501.36, "Marketing");
* create an instance of an Admin, and   
    
  **Code**  
          Admin adm = new Admin(304, "Bill Munroe", "108-23-6509", 75\_002.34);
* create an instance of a Director and save EmployeePractice   
    
  **Code**  
          Director dir = new Director(12, "Susan Wheeler", "099-45-2340", 120\_567.36, "Global Marketing",                     100\_000.00);

You now want to print out information about your classes. Adding System.out.println methods after each of the instances you created will create a lot of redundant code.   
  
Instead, you can add a printEmployee method to EmployeeTest that takes an Employee object as the parameter.  
  
Because all the object instances you're creating are Employee objects, regardless of which subclass you create, the printEmployee method will work – however, the Employee class can't know about the specialization of its subclasses.

Code

package com.example;  
...  
public class EmployeeTest {  
  
    public static void main(String[] args) {  
  
       ...  
  
    }  
  
        public static void printEmployee (Employee emp) {  
            System.out.println(); // Print a blank line as a separator  
            // Print out the data in this Employee object  
                System.out.println ("Employee id:         " + emp.getEmpId());  
                System.out.println ("Employee name:       " + emp.getName());  
                System.out.println ("Employee Soc Sec #:  " + emp.getSsn());  
                System.out.println ("Employee salary:     " + emp.getSalary());  
        }  
}

You can also perform optional tasks on the classes by following these steps:

* use the raiseSalary and setName methods on some of your objects to make sure those methods work   
    
  **Code**  
          System.out.println("\nTesting raiseSalary and setName on Manager:");  
          mgr.setName ("Barbara Johnson-Smythe");  
          mgr.raiseSalary(10000);  
          printEmployee(mgr);
* improve the look of the salary print output using the NumberFormat class – for example, you can format the salary to look like a standard US dollar currency   
    
  **Code**  
      System.out.println("Employee salary:" + NumberFormat.getCurrencyInstance().format((double)                          emp.getSalary()));
* prevent a negative value for the raiseSalary method, and   
    
  **Code**  
          public void raiseSalary(double increase) {  
          // Make sure the increase is not less than 0  
          if (increase > 0) {  
              salary += increase;
* prevent a null or empty value for the setName method   
    
  **Code**  
      public void setName(String name) {  
          // Make sure the name value is not null or empty  
          if (name != null && !name.equals("")) {  
              this.name = name;  
          }  
      }

**4. Summary**

In this topic, you've learned how to test Java classes and create Java subclasses.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#top)

Using Access Control and Overriding Methods

Learning Objectives

*After completing this topic, you should be able to*

* *use access levels*
* *override methods*

**1. Using access levels**

There are four access levels that can be applied to data fields and methods.

**private**

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.

**default**

When a keyword is absent, the default access modifier is applied. Classes can be default, that is have no modifier, or public.

**protected**

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. The protected access level 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 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.

**public**

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

Supplement

*Selecting the link title opens the resource in a new browser window.*

**Learning Aid**

Access the learning aid [Keyword modifier access levels](javascript:doWindow('./lajsep_a01_it_enust901_frame.html')) to find out how they apply in different parts of a program.

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, the resetResult method sets the value of this.result to 0. This code compiles fine, but the value of results is never getting set to 0.

Code

package demo;  
public class Foo {  
    protected int result;  
}  
  
...  
package test;  
import demo.Foo;  
public class Bar extends Foo {  
    private int results = 10;  
    public void resetResult () {  
    this.result = 0;  
    }  
}

This is because the value of this.result is the protected data field declared in class Foo. Because Bar extends Foo, the protected field is accessible, even in a different package. The programmer probably meant to modify the value of this.results in the class Foo, but since the code compiled hours later, it was hard to track down this error.  
  
A good practice is to make each class and class member, data field, and method as inaccessible as possible. In other words, always err on the side of hiding as much as possible.

Code

package demo;  
public class Foo {  
    protected int result;  
}  
  
...  
package test;  
import demo.Foo;  
public class Bar extends Foo {  
    private int results = 10;  
    public void resetResult () {  
    this.result = 0;  
    }  
}

This is a slightly modified version of the example using the protected keyword. If the idea is to limit access of the field result to classes within the package and the subclasses, which are package-protected, you should make the access explicit by defining a method purposefully written for package and subclass-level access.   
  
This does not resolve the issue of confusion between results and result, but addresses the intended use of a protected member.

Code

package demo;  
public class Foo2 {  
    private int result;  
    protected setResult (int value) { result = value; }  
}  
  
...  
package test;  
import demo.Foo2;  
public class Bar2 extends Foo2 {  
    public void resetResult () {  
        setResult(0); // Much better  
    }  
}

Question

Match the access level to its description.

**Options:**

1. private
2. default
3. protected
4. public

**Targets:**

1. Is applied to provide the greatest control over access to fields and methods
2. Is applied when a keyword is absent
3. Is applied to keep access within the package and subclass
4. Is applied to provide the greatest access to fields and methods

Answer

*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.*

*When an access modifier keyword is absent, the default access modifier is applied. Default fields and methods are accessible from the same class and the same package.*

*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.*

*The public keyword provides the greatest access to fields and methods. It makes them accessible in the class, package, subclasses, and any other class.*

**Correct answer(s):**

Target 1 = Option A

Target 2 = Option B

Target 3 = Option C

Target 4 = Option D

**2. Using override methods**

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

Code

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, or superclass. You may want to specialize this method to describe a Manager object.

Code

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

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.

Code

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

Note

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

When a method is overridden, it replaces the method in the parent class. This method will be called for any Manager instance. A call of the form super.getDetails() invokes the getDetails method of the parent class.

Code

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

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.*

During run time, the Java Virtual Machine will invoke the getDetails method of the appropriate class. Say you add this code to the program using the Employee and Manager classes. The output will differ, depending on the situation:

Code

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());

* When you run the program, the correct getDetails method of each class is called, producing this output.   
    
  **Code**  
  Employee id: 101 Employee name: Jim Smith  
  Employee id: 102 Employee name: Joan Kern Department: Marketing
* If you comment out the getDetails method in the Manager class, the m.getDetails() would produce different output. Methods are inherited from the parent class, so at run time, the getDetails method of the parent class, Employee, would be executed.   
    
  **Code**  
  Employee id: 101 Employee name: Jim Smith  
  Employee id: 102 Employee name: Joan Kern

Suppose you enter this code in the program.

Code

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

During execution, the object’s run time type is determined to be a Manager object. 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.

Code

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

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 – that is, the behavior of an object is determined by its run time 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.

Code

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

Note

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

An overriding method cannot be less accessible than the method in the parent class. To override a method, the name and the order of arguments must be identical.  
  
By changing the access of the Manager getDetails method to private, only that class can execute the method. However, language semantics for the execution of e.getDetails dictate that the Manager getDetails method is executed. This result is a compiler error.

Graphic

*The getDetails method in the Manager class is set to private.*

Code

public class Employee {  
    //... other fields and methods  
    public String getDetails() { ... }  
}  
  
public class Manager extends Employee {  
    //... other fields and methods  
    private String getDetails() { ... }  
}  
  
public class OverriddingTest {  
    Employee e = new Manager(102, "Joan Kern", "012-23-4567", 110\_450.54, "Marketing");  
    e.getDetails(); // illegal – method is private  
    }  
}

When you make the Employee method getDetails private and the Manager  
method public, the compiler does type checking, The compiler will complain that you are attempting to access a private method in Employee.

Graphic

*The Employee getDetails method is set to private and the Manager getDetails method set to public.*

Code

public class Employee {  
    //... other fields and methods  
    private String getDetails() { ... }  
}  
  
public class Manager extends Employee {  
    //... other fields and methods  
    public String getDetails() { ... }  
}  
  
public class OverriddingTest {  
    Employee e = new Manager(102, "Joan Kern", "012-23-4567", 110\_450.54, "Marketing");  
    e.getDetails(); // illegal – method is private  
    }  
}

Question

Which statements best describe overriding methods?

**Options:**

1. An overriding method cannot be less accessible than the method in the parent class
2. To override a method, the order of arguments must be identical
3. When a method is overridden, the calls to the method by the subclass will be replaced
4. A subclass can invoke a parent method using the parent keyword

Answer

***Option 1:****Correct. An overriding method cannot be less accessible than the method in the parent class. The compiler does type checking, and will complain if you attempt to access a private method.*

***Option 2:****Correct. To override a method, the name and the order of arguments must be identical.*

***Option 3:****Correct. When a subclass overrides a method, it will no longer use the method of the parent class. The method associated with that subclass will be replaced with the overriding method. For example, by creating a method with the same signature as a method in the parent class, you are overriding the method from that parent class.*

***Option 4:****Incorrect. A subclass can invoke a parent method using the super keyword.*

**Correct answer(s):**

1. An overriding method cannot be less accessible than the method in the parent class  
2. To override a method, the order of arguments must be identical  
3. When a method is overridden, the calls to the method by the subclass will be replaced

**3. Summary**

In this topic you've learned how to use access levels, and how to override methods.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#top)

Comparing Objects, Virtual Methods, and Casting

Learning Objective

*After completing this topic, you should be able to*

* *use constructors and casts in Java*

**1. Using constructors and casts in Java**

Suppose that you've created an Employee class that among other things, can print employee details on screen. You're then asked to create a new class that calculates a stock grant for employees, based on their salary and the type of employee they are – manager, engineer, and admin.

Code

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

This is a possible solution and each method performs the calculation based on the type of employee passed in, and returns the number of shares. Besides not being very object-orientated, this solution also presents a design problem.   
  
Consider what happens if you were to add two or three more employee types. You would need to add three additional methods, and possibly replicate code depending on 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 a lot of duplicate code.

Code

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  
}

A good practice is to pass parameters and write methods that use the most generic form of your object as possible. In this case, Employee is a good base class to start from, but you still have to determine what object type is passed in to the method.

Code

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);  
...

The Java language provides the keyword instanceof to determine an object's class type at run time.  
  
In this modified EmployeeStockPlan, the grantStock method uses instanceof to determine what type of Employee was passed to the method.

Code

public class EmployeeStockPlan {  
    public int grantStock (Employee e) {  
        // perform a calculation based on Employee data  
        if (e instanceof Manager) {  
            // process Manager stock grant  
        } else if (e instanceof Engineer) {  
            // process Engineer stock grant  
        } else if (e instanceof Admin) {  
            // process Admin stock grant  
        } else {  
            // perhaps an error – a generic Employee?  
        }  
    }  
}

After using the instanceof operator to verify that the object you received as an argument is a subclass, you can then access the full functionality of the object by casting the reference. Without the cast to Manager, the setDeptName method would not compile.

Graphic

*The casting is done by adding (Manager) to the declaration.*

Code

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

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

Code

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

Casting can be done upwards and downwards however, there are rules for casting references.

**Upward casting**

Upward casts are always permitted and do not require a cast operator.  
*The graphic shows an Employee class that has a Manager subclass, that has a Director subclass. Directional arrows show a Manager being upward cast as an Employee, a Director as a Manager, and a Director as an Employee.*   
  
**Code**  
Director d = new Director();  
Manager m = new Manager();

**Downward casting**

With a downward cast, the compiler simply determines if the cast is possible. If the cast down is to a subclass, it is possible that the cast will succeed.  
*The graphic shows an Employee class that has a Manager and an Engineer subclass. The Manager subclass has a Director subclass. Directional arrows show an Employee class being downward cast to the Manager class, and a Manager class downward cast to the Director class. A cast from Manager to Engineer fails however.*   
  
**Code**  
Employee e = new Manager();  
Manager m = new Manager();

At run time a downward 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 outside class hierarchy, such as the cast from a Manager instance to an Engineer, fails. A Manager and an Engineer are both employees, but a Manager is not an Engineer.

One of the advantages of single inheritance is that every class has a parent object by default. The java.lang.Object class is the root class of all classes in the Java programming language. Furthermore, you don't have to declare that your class extends Object. The compiler does that for you and this means that these two lines of code are equivalent.

Code

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

Object defines several non-final methods that are designed to be overridden by your class. These are equals, hashCode, toString, clone, and finalize. The clone method is used to create and return a copy of an object. And the finalize method is called by the garbage collector when it is determined that there are no more references to the object.

The toString method returns a String representaion of your object. This includes instance information such as the name of the class and a hexadecimal representation of the hash code of the object.

Code

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

You can also use toString to provide instance information. This is a better approach to getting details about your class than creating your own getDetails method.

Code

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

Sometimes you may want to be able to print out the name of the class that is executing a method. The getClass() method is used to return the Class object instance, and the getName() method provides the fully qualified name of the run time class. These methods are in the Object class.

Code

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

The Object equals method by default only determines if the value of two object references point to the same object. Basically, the test in the Object class is simply if x == y, return true.  
  
For an object that contains values, like the Employee object, this comparison is not sufficient, particularly if we want to make sure there is one and only one employee with a particular ID.

Code

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!

The equals method only determines if the value of two object references point to the same object. Given that what we really want is to test the contents of the Employee object, we need to override the equals method.  
  
This simple equals test first tests to make sure the object passed in is not null, and then that it is an instance of an Employee class. This will work because all subclasses are also employees. Then the Object is cast to Employee and each field in Employee is checked for equality.

Code

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;  
}

Note

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

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

Code

// 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;  
}

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 IDE’s like  
NetBeans can generate hashCode for you.

Code

// 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;  
}

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

Code

// Generated by NetBeans  
public int hashCode() {

* Start with an integer non-zero constant. Prime numbers result in fewer hashcode collisions.   
    
  **Code**  
  int hash = 7;
* For each field used in the equals method, compute an int hash code for the field. Note that for the Strings, you can use the hashCode of the String.   
    
  **Code**  
  hash = 83 \* hash + this.empId;  
  hash = 83 \* hash + Objects.hashCode(this.name);  
  hash = 83 \* hash + Objects.hashCode(this.ssn);  
  hash = 83 \* hash + (int)
* Add the computed hash codes together.   
    
  **Code**  
      (Double.doubleToLongBits(this.salary) ^  
      (Double.doubleToLongBits(this.salary) >>> 32));
* Return the result.   
    
  **Code**  
  return hash;  
  }

Question

What must be considered when overriding Object methods?

**Options:**

1. The equals, hashCode, and toString methods should not be overridden
2. An advantage of single inheritance is that every class has a parent object by default
3. You must declare that your class extends Object
4. The root class contains several non-final methods that are designed to be overridden

Answer

***Option 1:****Incorrect. Object defines several non-final methods that are designed to be overridden by your class. These are equals, hashCode, toString, clone, and finalize.*

***Option 2:****Correct. 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.*

***Option 3:****Incorrect. You do not have to declare that your class extends Object – the compiler does that for you.*

***Option 4:****Correct. The root class contains several non-final methods, but you should consider overriding toString, equals, and hashCode.*

**Correct answer(s):**

2. An advantage of single inheritance is that every class has a parent object by default  
4. The root class contains several non-final methods that are designed to be overridden

Question

Which statement best describes the toString method?

**Options:**

1. It determines if two object references point to the same object
2. It is used in hash-based collections
3. It can be used to provide instance information
4. It is called whenever an instance of your class is passed to a method that takes a String

Answer

***Option 1:****Incorrect. The equals method by default only determines if the value of two object references point to the same object.*

***Option 2:****Incorrect. The hashCode method is used in conjunction with the equals method in hash-based collections, such as HashMap, HashSet, and Hashtable.*

***Option 3:****Correct. You can use the toString method to provide instance information. This is a better approach to getting details about your class than creating your own getDetails method.*

***Option 4:****Incorrect. The toString method is only automatically called whenever an instance of your class is passed to the println method.*

**Correct answer(s):**

3. It can be used to provide instance information

**2. Overriding methods and virtual invocation**

Now you will override the toString method of the Object class in the Employee class and in the Manager class. You will create an EmployeeStockPlan class with a grantStock method that uses the instanceof operator to determine how much stock to grant based on the employee type.

You'll use the EmployeePractice project, and apply all code changes to the existing program files.

You now want to edit the Employee class to override the toString() method from the Object class. The Object's toString method returns a String. These are the steps:

Code

    public void setName(String name) {  
        // Make sure the name value is not null or empty  
        if (name != null && !name.equals("")) {  
            this.name = name;  
        }  
    }  
  
    public void raiseSalary(double increase) {  
        // Make sure the increase is not less than 0  
        if (increase > 0) {  
            salary += increase;  
  
        }  
   }

* add the toString method to the Employee class with this signature   
    
  **Code**  
          public String toString() {
* add a return statement that returns a string that includes the employee information – ID, name, SSN, and a formatted salary – and   
    
  **Code**  
              return "Employee ID: " + getEmpId() + "\n" +  
                  "Employee Name: " + getName() + "\n" +  
                  "Employee SSN: " + getSsn() + "\n" +  
                  "Employee Salary: " +  
                  NumberFormat.getCurrencyInstance().format(getSalary());
* save the Employee class

Now you want to override the toString method in the Manager class to include the deptName field value. You follow these steps:

* Open the Manager class.   
    
  **Code**  
  package com.example.domain;  
    
      public class Manager extends Employee {  
          private String deptName;  
    
          public Manager(int empId, String name, String ssn, double salary, String deptName) {  
              super(empId, name, ssn, salary);  
              this.deptName = deptName;  
          }  
    
          public String getDeptName() {  
              return deptName;  
         }
* Add a toString method with the same signature as the Employee toString method. The toString method in the Manager class overrides the toString method inherited from the Employee class.   
    
  **Code**  
    
          @Override  
          public String toString() {
* Call the parent class method by using the super keyword and add the department name.   
    
  **Code**  
          return super.toString() + "\nDepartment: " + getDeptName();  
      }  
  }

Note the Green circle icon with the "o" in the center beside the method signature in the Manager class. This indicates that NetBeans is aware that this method overrides the method from the parent class, Employee. If you hold the cursor over the icon, you can read what it represents. If you click the icon, NetBeans will open the Employee class and position the view to the toString() method.  
  
Finally, you save the Manager class.

Note

*Optionally, you can also override the toString method in the Director class as well to display all the fields of a director and the available budget.*

You now want to create a new class called EmployeeStockPlan in the package com.example.business. This class will include a single method, grantStock, which takes an Employee object as a parameter and returns an integer number of stock options based on the employee type.

Graphic

*The table consists of two columns and four rows. The columns are named Employee type and Stock options. The remaining three rows list the Employee types, the first is Director who has a Stock options value of 1000. The Manager type has a Stock options value of 100, and the Other employees Employee type has a Stock options value of 10.*

You follow these steps:

* create the new package and class in one step by right-clicking **Source Packages**, and selecting **New** – **Java Class**, and   
  *The right-click menu is open on the New submenu and Java Class is highlighted. Other options include Properties File, Java Interface, Java Package, and Java Main Class.*
* enter EmployeeStockPlan as the class name, com.example.business as the package, and then click **Finish**   
  *The New Java Class wizard is open on the second step – Name and Location. There are several fields:Class Name is set to EmployeeStockPlan, Project is EmployeePractice, Location is Source Packaged, and the Package is com.example.business. The Created File field shows the path to the file, which is named EmployeeStockPlan.java.*

Ensure the EmployeeStockPlan.java class is open, and then implement these steps:

Code

package com.example.business;  
  
public class EmployeeStockPlan {

* in the new class, add these fields to the class to define the stock levels   
    
  **Code**  
      private final int employeeShares = 10;  
      private final int managerShares = 100;  
      private final int directorShares = 1000;
* add a grantStock method that takes an Employee object reference as a parameter and returns an integer   
    
  **Code**  
      public int grantStock(Employee emp) {
* in the method body, determine what employee type was passed in using the instanceof keyword, and return the appropriate number of stock options based on that type   
    
  **Code**  
          // Stock is granted based on the employee type  
          if (emp instanceof Director) {  
              return directorShares;  
          } else {  
          if (emp instanceof Manager) {  
              return managerShares;  
          } else {  
              return employeeShares;  
     }  
  }
* resolve any missing import statements, and   
    
  **Code**  
  package com.example.business;  
    
  import com.example.domain.Director;  
  import com.example.domain.Employee;  
  import com.example.domain.Manager;  
    
  public class EmployeeStockPlan {  
    
      private final int employeeShares = 10;  
      private final int managerShares = 100;  
      private final int directorShares = 1000;  
    
      public int grantStock(Employee emp) {  
          // Stock is granted based on the employee type  
          if (emp instanceof Director) {  
              return directorShares;  
          } else {  
          if (emp instanceof Manager) {  
              return managerShares;  
          } else {  
              return employeeShares;  
     }  
  }  
  }
* save the EmployeeStockPlan class   
    
  **Code**  
  package com.example.business;  
    
  import com.example.domain.Director;  
  import com.example.domain.Employee;  
  import com.example.domain.Manager;  
    
  public class EmployeeStockPlan {  
    
      private final int employeeShares = 10;  
      private final int managerShares = 100;  
      private final int directorShares = 1000;  
    
      public int grantStock(Employee emp) {  
          // Stock is granted based on the employee type  
          if (emp instanceof Director) {  
              return directorShares;  
          } else {  
          if (emp instanceof Manager) {  
              return managerShares;  
          } else {  
              return employeeShares;  
     }  
  }  
  }

Now you modify the EmployeeTest class. Replace the four print statements in the printEmployee method with a single print statement that uses the toString method you created. To do this, you follow these steps:

Code

    public static void printEmployee(Employee emp) {  
        System.out.println(); // Print a blank line as a separator  
        // Print out the data in this Employee object  
        System.out.println("Employee type:   " + emp.getClass().getSimpleName());  
        System.out.println("Employee id: " + emp.getEmpId());  
        System.out.println("Employee name: " + emp.getName());  
        System.out.println("Employee Soc Sec #: " + emp.getSsn());  
        System.out.println("Employee salary: " +  
        NumberFormat.getCurrencyInstance().format((double)  
        emp.getSalary()));  
   }

* replace this code block   
    
  **Code**  
          System.out.println("Employee id: " + emp.getEmpId());  
          System.out.println("Employee name: " + emp.getName());  
          System.out.println("Employee Soc Sec #: " + emp.getSsn());  
          System.out.println("Employee salary: " +  
          NumberFormat.getCurrencyInstance().format((double)  
          emp.getSalary()));
* with this line of code   
    
  **Code**  
          System.out.println(emp);

Now overload the printEmployee method to take a second parameter, EmployeeStockPlan, and print out the number of stock options this employee will receive. Follow these steps:

Code

    public static void printEmployee(Employee emp) {  
        System.out.println(); // Print a blank line as a separator  
        // Print out the data in this Employee object  
        System.out.println(emp);  
    }

* create another printEmployee method that takes an instance of the EmployeeStockPlan class   
    
  **Code**  
          public static void printEmployee(Employee emp, EmployeeStockPlan esp) {
* add this method that first calls the original printEmployee method   
    
  **Code**  
              printEmployee(emp);
* add a print statement to print out the number of stock options the employee is entitled to, and   
    
  **Code**  
              System.out.println("Stock Options:   " + esp.grantStock(emp));
* above the printEmployee method calls in the main method, create an instance of the EmployeeStockPlan and pass that instance to the each of the printEmployee methods, and resolve any missing import statements   
    
  **Code**  
          EmployeeStockPlan esp = new EmployeeStockPlan();  
          printEmployee(eng, esp);  
          ...

Save the EmployeeTest class and run the application. This should produce output for each employee that includes the number of Stock Options.

Graphic

*The output produced is:  
  
Employee id: 101  
Employee name: Jane Smith  
Employee Soc Sec #: 012-34-5678  
Employee salary: $120,345.27  
Stock Options: 10*

If you want to know what type of employee each employee is, you add this code to your original printEmployee method above the print statement that prints the employee data fields.

Graphic

*The code for the print statement is:  
  
System.out.println("Employee type:   " + emp.getClass().getSimpleName());*

Code

    public static void printEmployee(Employee emp) {  
        System.out.println(); // Print a blank line as a separator  
        // Print out the data in this Employee object  
        System.out.println("Employee type:   " + emp.getClass().getSimpleName());  
        System.out.println(emp);  
    }

This will print out the simple name of the class, such as Manager or Engineer. The output of the first employee record should now produce this.

Graphic

*The output produced is:  
  
Employee type: Engineer  
Employee id: 101  
Employee name: Jane Smith  
Employee Soc Sec #: 012-34-5678  
Employee salary: $120,345.27  
Stock Options: 10*

**3. Summary**

In this topic you've learned how to use constructors and casts in Java.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#top)

Working with Abstract Classes

Learning Objective

*After completing this topic, you should be able to*

* *work with abstract Java classes and subclasses*

**1. Working with abstract Java classes**

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.

Graphic

*Electronic Device is depicted as the parent class of Mobile Phone and Television.*

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, or 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.

Inheritance or sub-classing is an essential feature of the Java programming language. Inheritance provides code reuse through

Graphic

*A Class Inheritance Diagram shows four objects, Caller, ElectronicDevice, MobilePhone, and Television. MobilePhone and Television inherits the methods of ElectronicDevice, and these methods are public void turnOn() and public void turnOff(). The diagram shows a Caller object using an ElectronicDevice object.*

**method inheritance, and**

Inheritance provides code reuse through method inheritance, because subclasses avoid code duplication by inheriting method implementations.

**generalization**

Inheritance provides code reuse through generalization, since code designed to rely on the most generic type possible is easier to maintain.

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. You should 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.

Code

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

Subclasses may not need to inherit a method implementation if the method is specialized. 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 need to be overridden with a specialized implementation. In these cases, inclusion of the method in a parent class is a mixed blessing. It allows the use of generic reference types, but developers can easily forget to supply the specialized implementation in the subclasses.

Code

public class Television extends ElectronicDevice {  
    public void turnOn() {  
        changeChannel(1);  
        initializeScreen();  
    }  
    public void turnOff() {}  
    public void changeChannel(int channel) {}  
    public void initializeScreen() {}  
}

You can declare a class as abstract by using the abstract class level modifier but note the following:

Code

public abstract class ElectronicDevice { }

* An abstract class will typically be extended by a child class and may be used as a reference type, but   
    
  **Code**  
  public class Television extends ElectronicDevice { }
* an abstract class cannot be instantiated   
    
  **Code**  
  ElectronicDevice dev = new ElectronicDevice(); // error

A method can be declared as abstract by using the abstract method level modifier. An abstract method cannot have a method body. It must be declared in an abstract class, and is overridden in sub-classes.  
  
When a child class inherits an abstract method, it's 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.

Code

public abstract class ElectronicDevice {  
  
    public abstract void turnOn ();  
    public abstract void turnOff ();  
}

The following additional rules apply when using abstract classes and methods:

Code

public abstract class ElectronicDevice {  
  
    public abstract void turnOn;  
    public abstract void turnOff;  
}

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

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

Question

Identify the rules for defining an abstract method.

**Options:**

1. It cannot be declared in an abstract class
2. It must have a method body
3. It is overridden in subclasses
4. It requires the abstract method level modifier

Answer

***Option 1:****Incorrect. When an abstract method is defined, it must be declared in an abstract class.*

***Option 2:****Incorrect. When a child class inherits an abstract method, it is inheriting a method signature but no implementation.*

***Option 3:****Correct. An abstract method cannot have a method body, must be declared in an abstract class, and is overridden in subclasses.*

***Option 4:****Correct. A method can be declared as abstract using the abstract method level modifier.*

**Correct answer(s):**

3. It is overridden in subclasses  
4. It requires the abstract method level modifier

Question

When using abstract classes and methods, which actions must you take when inheriting from an abstract class?

**Options:**

1. Declare the child class as abstract
2. Declare the child class as private
3. Override all methods inherited from the parent class
4. Override all abstract methods inherited from the parent class

Answer

***Option 1:****Correct. When inheriting from an abstract class, you must either declare the child as abstract or override all abstract methods inherited from the parent class.*

***Option 2:****Incorrect. You should declare the child class as abstract, not private, or override all abstract methods inherited from the parent class.*

***Option 3:****Incorrect. When inheriting from an abstract class, you must override all abstract methods inherited from the parent class or declare the child as abstract.*

***Option 4:****Correct. When inheriting from an abstract class, you must either declare the child as abstract or override all abstract methods inherited from the parent class.*

**Correct answer(s):**

1. Declare the child class as abstract  
4. Override all abstract methods inherited from the parent class

**2. Using abstract classes**

Next you will re-factor the code to use the abstract keyword in an existing application.

You have been given a project that implements the logic for a bank. The banking software supports only the creation of time deposit accounts. Time deposit accounts only allow withdrawal after a maturity date. Time deposit accounts are also known as term deposit, certificate of deposit , or CD, or fixed deposit accounts. You will enhance the software to support checking accounts.

A checking account and a time deposit account have some similarities and some differences. Your class design should reflect this. Additional types of accounts might be added in the future.

You first open the main project. To do this you

Graphic

*NetBeans IDE is open and the Open Project dialog box displays. It includes the Look in drop-down list, which currently is set to practices. It also displays the list of projects available in this folder: AbstractBanking (selected), NestedClasses, and SingletonBanking. The File name text box displays D:\labs\04-Advanced\_Class\_Design\practices. The Files of Type drop-down list is set to Project Folder. There are two buttons to move to the higher folder in the hierarchy, and to create a new folder. The Project Name text box displays AbstractBanking. The Open as Main Project and the Open Required Projects checkboxes are cleared. There are also Open Project and Cancel buttons.*

* click **File** – **Open Project**
* browse to the project folder
* select the project and select the **Open as Main Project** check box, and
* click the **Open Project** button

Next you expand the project directories.

Graphic

*The AbstractBanking project has been expanded and contains two child nodes – Source Packages and Libraries. Source Packages has been fully expanded and contains the com sub-node, which contains the example sub-node. The example node contains a number of child files: AbstractBankingMain.java, Bank.java, Customer.java, CustomerReport.java, and TimeDepositAccount.java.*

After you run the project, you should see a report of all customers and their accounts.

Graphic

*The nodes have been collapsed again, but the program has produced output as shown in the Output window.*

Code

                                CUSTOMERS REPORT  
                                ================  
  
Customer: Simms, Jane  
    Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current balance is 500.0  
  
Customer: Bryant, Owen  
  
Customer: Soley, Tim  
    Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current balance is 1500.0  
  
Customer: Soley, Maria  
    Time Deposit Account Fri Mar 09 12:04:28 CST 2012: current balance is 150.0

You now want to review the TimeDepositAccount class. Complete these steps:

* Open the **TimeDepositAccount.java** file found under the com.example package.   
    
  **Code**  
  package com.example;  
    
  import java.util.Date;  
    
  public class TimeDepositAccount {  
        
      private double balance;  
      private final Date maturityDate;  
        
      public TimeDepositAccount(double balance, Date maturityDate) {  
          this.balance = balance;  
          this.maturityDate = maturityDate;  
      }  
    
      public double getBalance() {  
          return balance;  
      }  
        
      public void deposit(double amount) {  
          balance += amount;  
     }  
       
  ...
* Identify the fields and method implementations of TimeDepositAccount that are related to time or are in some other way specific to TimeDepositAccount. These include for example, the maturityDate field.   
    
  **Code**  
      private double balance;  
      //time deposit account specific code  
      private final Date maturityDate;  
    
  ...  
    
      public boolean withdraw(double amount) {  
          //time deposit account specific code  
          Date today = new Date();  
          if(today.after(maturityDate)) {  
              if(amount <= balance) {  
                  balance -= amount;  
                  return true;  
              } else {  
                  return false;  
              }  
          } else {  
              return false;  
          }  
      }  
    
      public String getDescription() {  
          //time deposit account specific code  
          return "Time Deposit Account " + maturityDate;  
     }
* Identify the fields and method implementations of TimeDepositAccount, which could be used by any type of account. These include the balance field and the getBalance, getDescription and toString methods.   
    
  **Code**  
      public double getBalance() {  
          // generic account code  
          return balance;  
      }  
        
      public void deposit(double amount) {  
          // generic account code  
          balance += amount;  
      }  
        
      @Override  
      public String toString() {  
          // generic account code  
          return getDescription() + ": current balance is " + balance;  
     }

Now create a new Java class called Account in the com.example package.

Graphic

*The example node is selected and the right-click menu is open. This menu has a number of options: New, Find, Cut, Copy, Paste, Delete, Refactor, Local History, Tools, and Properties. The New sub-menu is open and the Java Class option is selected.*

You now complete the following steps to code the Account class:

Code

package com.example;  
  
public class Account {  
  
}

* Declare this class as abstract.   
    
  **Code**  
  package com.example;  
    
  public abstract class Account {
* Assign the Account class a protected access level balance field. Move this field from TimeDepositAccount and change the access level.   
    
  **Code**  
      protected double balance;
* Add an Account class constructor, which has a double balance parameter.   
    
  **Code**  
      public Account(double balance) {  
          this.balance = balance;  
      }
* You want to move field or method implementations from TimeDepositAccount that could be used in any type of account. So you move the getBalance, deposit, and toString methods from TimeDepositAccount to the Account class. The fields and methods should be removed from TimeDepositAccount.   
    
  **Code**  
      public double getBalance() {  
          return balance;  
      }  
        
      public void deposit(double amount) {  
          balance += amount;  
      }  
        
      @Override  
      public String toString() {  
          return getDescription() + ": current balance is " + balance;  
     }
* You want to add abstract methods for any methods that are time related in TimeDepositAccount, but that would make sense in any type of account. So you add abstract getDescription and withdraw methods to the Account class.   
    
  **Code**  
      public abstract boolean withdraw(double amount);  
        
      public abstract String getDescription();  
    
  }

Next you complete these steps to modify the TimeDepositAccount class:

* Make TimeDepositAccount a subclass of Account.   
    
  **Code**  
  public class TimeDepositAccount extends Account {
* Modify the TimeDepositAccount constructor to call the parent class constructor with the balance value.   
    
  **Code**  
    
      private final Date maturityDate;  
    
      public **TimeDepositAccount** (double balance, Date maturityDate) {  
    
          super(balance);  
          this.maturityDate = maturityDate;  
      }
* Make sure you are overriding the abstract withdraw and getDescription methods inherited from the Account class by using the @Override annotation.   
    
  **Code**  
    
      @override  
      public boolean **withdraw**(double amount) {  
          Date today = new Date();  
          if(today.after(maturityDate)) (  
              if(amount <= balance) (  
                  balance -= amount;  
                  return true;  
              ) else (  
                  return false;  
              )  
          )  
      @Override  
      public String **getDescription**() {  
          return "Time Deposit Account " + maturityDate;  
     }

You want to modify the Customer and CustomerReport classes to use Account references. To do this, follow these steps:

* Open the **Customer.java** file under the com.example package, and find all references to TimeDepositAccount   
    
  **Code**  
  package com.example;  
    
  public class Customer {  
    
      private String firstName;  
      private String lastName;  
      private TimeDepositAccount[] accounts;  
      private int numberOfAccounts;  
    
      public Customer(String f, String l) {  
          firstName = f;  
          lastName = l;  
          // initialize accounts array  
          accounts = new TimeDepositAccount[10];  
          numberOfAccounts = 0;  
      }  
  ...  
      public void addAccount(TimeDepositAccount acct) {  
          int i = numberOfAccounts++;  
          accounts[i] = acct;  
      }  
  ...  
      public TimeDepositAccount getAccount(int account\_index) {  
          return accounts[account\_index];  
     }
* Change all TimeDepositAccount references to Account type references.   
    
  **Code**  
  package com.example;  
    
  public class Customer {  
    
      private String firstName;  
      private String lastName;  
      private Account[] accounts;  
      private int numberOfAccounts;  
    
      public Customer(String f, String l) {  
          firstName = f;  
          lastName = l;  
          // initialize accounts array  
          accounts = new Account[10];  
          numberOfAccounts = 0;  
      }  
  ...  
      public void addAccount(Account acct) {  
          int i = numberOfAccounts++;  
          accounts[i] = acct;  
      }  
  ...  
      public Account getAccount(int account\_index) {  
          return accounts[account\_index];  
     }
* Now open the **CustomerReport.java** file under the com.example package, and find the references to TimeDepositAccount in the generateReport method   
    
  **Code**  
      public void generateReport() {  
    
          // Print report header  
          System.out.println("\t\t\tCUSTOMERS REPORT");  
          System.out.println("\t\t\t================");  
    
          // For each customer...  
          for (int cust\_idx = 0; cust\_idx < bank.getNumOfCustomers(); cust\_idx++) {  
              Customer customer = bank.getCustomer(cust\_idx);  
    
              // Print the customer's name  
              System.out.println();  
              System.out.println("Customer: "  
                      + customer.getLastName() + ", "  
                      + customer.getFirstName());  
    
              // For each account for this customer...  
              for (int acct\_idx = 0; acct\_idx < customer.getNumOfAccounts(); acct\_idx++) {  
                  TimeDepositAccount account = customer.getAccount(acct\_idx);  
    
                  // Print the current balance of the account  
                  System.out.println("    " + account);  
              }  
          }  
     }
* Repeat the second step and change all TimeDepositAccount references to Account type references.   
    
  **Code**  
      public void generateReport() {  
    
          // Print report header  
          System.out.println("\t\t\tCUSTOMERS REPORT");  
          System.out.println("\t\t\t================");  
    
          // For each customer...  
          for (int cust\_idx = 0; cust\_idx < bank.getNumOfCustomers(); cust\_idx++) {  
              Customer customer = bank.getCustomer(cust\_idx);  
    
              // Print the customer's name  
              System.out.println();  
              System.out.println("Customer: "  
                      + customer.getLastName() + ", "  
                      + customer.getFirstName());  
    
              // For each account for this customer...  
              for (int acct\_idx = 0; acct\_idx < customer.getNumOfAccounts(); acct\_idx++) {  
                  Account account = customer.getAccount(acct\_idx);  
    
                  // Print the current balance of the account  
                  System.out.println("    " + account);  
              }  
          }  
     }

You now run the project. You should see a report of all customers and their accounts.

Code

                                CUSTOMERS REPORT  
                                ================  
  
Customer: Simms, Jane  
    Time Deposit Account Fri Mar 09 13:04:59 CST 2012: current balance is 500.0  
  
Customer: Bryant, Owen  
  
Customer: Soley, Tim  
    Time Deposit Account Fri Mar 09 13:04:59 CST 2012: current balance is 1500.0  
  
Customer: Soley, Maria  
    Time Deposit Account Fri Mar 09 13:04:59 CST 2012: current balance is 150.0

You now create a new Java class called CheckingAccount in the com.example package. Complete these steps to write the code:

Code

package com.example;  
  
public class CheckingAccount {

* Define the CheckingAccount class as a subclass of Account.   
    
  **Code**  
  package com.example;  
    
  public class CheckingAccount extends Account {
* Add an overDraftLimit field to the CheckingAccount class.   
    
  **Code**  
      private final double overDraftLimit;
* Add a CheckingAccount constructor that has one parameter.   
    
  **Code**  
  public CheckingAccount(double balance) {  
      this(balance, 0);  
  }
* Add another CheckingAccount constructor using these two parameters.   
    
  **Code**  
      public CheckingAccount(double balance, double overDraftLimit) {  
          super(balance);  
          this.overDraftLimit = overDraftLimit;  
     }
* Override the abstract getDescription method inherited from the Account class. It is good practice to add @Override to any method, which should be overriding a parent class method.   
    
  **Code**  
      @Override  
      public String getDescription() {  
          return "Checking Account";  
      }
* Override the abstract withdraw method inherited from the Account class. The withdraw method should allow an account to go negative on the balance up to the amount specified in the overDraftLimit field.   
    
  **Code**  
      @Override  
      public boolean withdraw(double amount) {  
          if(amount <= balance + overDraftLimit) {  
              balance -= amount;  
              return true;  
          } else {  
              return false;  
          }  
     }

Now modify the AbstractBankingMain class to create checking accounts for the customers. Both Customer and CustomerReport can use CheckingAccount instances because you previously modified them to use Account type references.

Code

// Create several customers and their accounts  
bank.addCustomer("Jane", "Simms");  
customer = bank.getCustomer(0);  
customer.addAccount(new TimeDepositAccount(500.00, cal.getTime()));  
customer.addAccount(new CheckingAccount(200.00, 400.00));  
  
bank.addCustomer("Owen", "Bryant");  
customer = bank.getCustomer(1);  
customer.addAccount(new CheckingAccount(200.00));  
  
bank.addCustomer("Tim", "Soley");  
customer = bank.getCustomer(2);  
customer.addAccount(new TimeDepositAccount(1500.00, cal.getTime()));  
customer.addAccount(new CheckingAccount(200.00));  
  
bank.addCustomer("Maria", "Soley");  
customer = bank.getCustomer(3);  
// Maria and Tim have a shared checking account  
customer.addAccount(bank.getCustomer(2).getAccount(1));  
customer.addAccount(new TimeDepositAccount(150.00, cal.getTime()));

Now run the project. You should see a report of all customers and their accounts. Note that the date displayed should be 180 days in the future.

Code

                                CUSTOMERS REPORT  
                                ================  
  
Customer: Simms, Jane  
    Time Deposit Account Wed Sep 05 13:34:59 CST 2012: current balance is 500.0  
    Checking Account: current balance is 200.0  
  
Customer: Bryant, Owen  
    Checking Account: current balance is 200.0  
  
Customer: Soley, Tim  
    Time Deposit Account Wed Sep 05 13:34:59 CST 2012: current balance is 1500.0  
    Checking Account: current balance is 200.0  
  
Customer: Soley, Maria  
    Checking Account: current balance is 200.0  
    Time Deposit Account Wed Sep 05 13:34:59 CST 2012: current balance is 150.0

**3. Summary**

In this topic you've learned to work with abstract Java classes and subclasses.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#top)

Advanced Class Design

Learning Objective

*After completing this topic, you should be able to*

* *use the static and final keywords*

**1. Using the static keyword**

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.

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, and
* should not be used to bypass the object-oriented features of Java unless there is good reason to

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 anytime a constructor is called. A static factory method could maintain a cache of objects for reuse or call new if the cache was depleted. A factory method may also produce an object which subclasses the method’s return type.

Code

NumberFormat nf = NumberFormat.getInstance();

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 object initialization-related tasks
* cannot access non-static members within the same class, and
* can be hidden in sub-classes but not overridden and thus allows no virtual method invocation

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

Code

public class StaticErrorClass {  
    private int x;  
  
    public static void staticMethod() {  
        x = 1; // compile error  
        instanceMethod(); // compile error  
    }  
    public void instanceMethod() {  
    x = 2;  
    }  
}

When calling static methods, you qualify the location of the method with a class name if the method is located in a different class than the caller. This isn't required for methods within the same class. You should avoid using an object reference to call a static method.

Code

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

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
* initialized when the containing class is first loaded, and
* useful for containing shared data because static methods store data in static variables and all object instances share a single copy of any static variables

Application developer-supplied classes are typically loaded on demand at 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.

Many technologies 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.

Code

public class StaticCounter {  
    private static int counter = 0;  
    public StaticCounter() {  
        counter++;  
    }  
    public static int getCount() {  
        return counter;  
    }  
}

When accessing static variables, there are two things you should do.

Code

double p = Math.PI;  
  
new StaticCounter();  
new StaticCounter();  
System.out.println("count: " + StaticCounter.getCount());

**double p = Math.PI;**

*You qualify the location of the variable with a class name if the variable is located in a different class than the caller. This is not required for variables within the same class.*

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

*You should avoid using an object reference to access a static variable. If all the members of a class are static, consider using a private constructor to prevent object instantiation.*

A static import statement makes the static members of a class available under their simple name. So, you can use either of these import lines to call the Math.random() method using its simple name.  
  
Overusing static import can negatively impact the readability of your code, and you should avoid adding multiple static imports to a class.

Code

import static java.lang.Math.random;  
import static java.lang.Math.\*;  
  
public class StaticImport {  
    public static void main(String[] args) {  
        double d = random();  
    }  
}

Question

Which statements best describe static class members?

**Options:**

1. They can be used without object instances
2. They are used when a problem is best solved with objects
3. They are used when objects of different types need to share fields
4. They should not be used to bypass the object-oriented features of Java

Answer

***Option 1:****Correct. The static modifier is used to declare fields and methods as class-level resources. Static class members can be used without object instances.*

***Option 2:****Incorrect. Static class members are used when a problem is best solved without objects.*

***Option 3:****Incorrect. Static class members are used when objects of the same type need to share fields.*

***Option 4:****Correct. Unless there is a good reason, static class members should not be used to bypass the object-oriented features of Java.*

**Correct answer(s):**

1. They can be used without object instances  
4. They should not be used to bypass the object-oriented features of Java

**2. Using the final keyword**

A method can be declared final. Final methods may not be overridden. There is little to no performance benefit to declaring a method as final. Methods should be declared as final only to disable method overriding.

Code

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");  
    }  
}

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

Code

public final class FinalParentClass { }  
// compile time error  
public class ChildClass extends FinalParentClass { }

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

* class fields
* method parameters, and
* local variables

Final references must always reference the same object but that object's contents may be modified.  
  
Final fields with compile-time constant expressions are constant variables, and static can be combined with final to create an always-available, never-changing variable.

There are some benefits to using final variables, including

**bug prevention, and**

Final variables can never have their values modified once 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.

A final object reference, however, 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.

Final fields must either be assigned a value when declared, or assigned a value in every constructor. A field that is both static and final is considered a constant. By convention, constant fields use identifiers consisting only of uppercase letters and underscores.

Code

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;  
    }  
  
    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  
    }  
}

While public static final variables can be very useful, 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 only receive one of three possible values. These lines of code would still compile.  
  
In the example, when using an int to represent state, you must perform a runtime range check. Within the setState method, an if statement could be used to validate that only the values 0, 1, or 2 are passed. This type of check would be performed every time the setState method was called, resulting in additional overhead.

Code

Computer comp = new Computer();  
comp.setState(Computer.POWER\_SUSPEND);  
  
Computer comp = new Computer();  
comp.setState(42);

Note

*The POWER\_SUSPEND constant has a value of 2.*

Java 5 added a typesafe enum to the language. Enums are created using a variation of a Java class and provide a compile-time range check. An enum can be used as a solution in the power state example :

* by creating references that reflect the three PowerState objects that can exist, and   
    
  **Code**  
  public enum PowerState {  
      OFF,  
      ON,  
      SUSPEND;  
  }
* by sending the enum as a parameter reflecting the correct state to a method   
    
  **Code**  
  Computer comp = new Computer();  
  comp.setState(PowerState.SUSPEND);

Enum references can be statically imported. In this example, code effectively uses the PowerState.OFF value to set powerState.

Code

import static com.example.PowerState.\*;  
public class Computer extends ElectronicDevice {  
    private Object powerState = OFF;  
    //...  
}

Enums can be used as the expression in a switch statement. In this example, OFFis statically imported.

Code

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

Enums may have fields, methods, and private constructors.   
  
In this example, a call to the PowerState constructor initializes the public static final OFF reference.

Graphic

*The code is:  
  
OFF("The power is off"),*

Code

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;  
    }  
}

Question

Which statements accurately describe final variables?

**Options:**

1. Final variables may not change their value once initialized
2. Final variables can be method parameters
3. Final references need not always reference the same object
4. Final variables may not be local variables

Answer

***Option 1:****Correct. Final variables can never have their values modified once initialized. This behavior functions as a bug prevention mechanism.*

***Option 2:****Correct. Final variables can be class fields, method parameters, and local variables.*

***Option 3:****Incorrect. Final references must always reference the same object, but that object’s contents may be modified.*

***Option 4:****Incorrect. Final variables can be class fields, method parameters, and local variables.*

**Correct answer(s):**

1. Final variables may not change their value once initialized  
2. Final variables can be method parameters

Design patterns are reusable solutions to common software development problems that are documented in 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 OO language.

The singleton design pattern is one of the creational design patterns as categorized by *Design Patterns: Elements of Reusable object-oriented Software*. There are three steps to implement the singleton design pattern.

Code

public class SingletonClass {  
    private static final SingletonClass instance =  
        new SingletonClass();  
  
    private SingletonClass() {}  
  
    public static SingletonClass getInstance() {  
        return instance;  
    }  
}

**private static final SingletonClass instance =  
       new SingletonClass();**

*First use a static reference to point to the single instance. Making the reference final ensures that it will never reference a different instance.*

**private SingletonClass() {}**

*Second, add a single private constructor to the singleton class. The private modifier only allows "same class" access, which prohibits any attempts to instantiate the singleton class except for the attempt in step 1.*

**public static SingletonClass getInstance() {  
       return instance;  
   }**

*Finally, a public factory method returns a copy of the singleton reference. This method is declared static in order 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, implementing a per-thread singleton solution for example, and are typically used in most singleton implementations.*

To obtain a singleton reference, call the getInstance method.

Code

SingletonClass ref = SingletonClass.getInstance();

A nested class is a class declared within the body of another class. Nested classes have multiple categories – inner classes, which include member, local, and anonymous classes, and static nested classes. Nested classes are commonly used in applications with GUI elements, and can limit utilization of a "helper class" to the enclosing top level class.

You would use nested classes because they

**are a logical grouping of classes**

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

**provide increased encapsulation, and**

Nested classes provide better 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.

**are more readable, maintainable code**

Nesting small classes within top-level classes places the code closer to where it is used, making it more readable and maintainable.

An inner class is considered part of the outer class and inherits access to all the private members of the outer class. The example shows an inner class, which is a member class. Inner classes can also be declared inside of a method block – in such a case, they are called *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.

Code

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();  
    }  
}

An anonymous class can be used to define a class with no name. In this example, the java.lang.Object class is being subclassed, and that subclass is being instantiated.  
  
When compiling 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.

Code

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

**3. Applying the singleton design pattern**

You now take an existing application and refactor the code to implement the Singleton design pattern.  
  
You have been given a project that implements the logic for a bank. The application currently allows the creation of an unlimited number of Bank instances. Using the static and final keywords, you will limit the number of Bank instances to one per JVM.

Code

Bank bank = new Bank();  
Bank bank2 = new Bank();  
Bank bank3 = new Bank();

You've opened the SingletonBanking project and expanded the project directory structure.

Graphic

*The SingletonBanking project directory structure has been expanded to the example node. This node contains a number of files: Account.java, Bank.java, CheckingAccount.java, Customer.java, CustomerReport.java, SingletonBankingMain.java, and TimeDepositAccount.java.*

Now you run the project. You should see a report of all customers and their accounts.

Code

CUSTOMERS REPORT  
==================  
  
Customer: Simms, Jane  
Time Deposit Account Sat Aug 11 08:30:20 ADT 2012: current balance is 500.00  
Checking Account: current balance is 200.0  
  
Customer: Bryant, Owen  
Checking Account: current balance is 200.0  
  
Customer: Soley, Tim  
Time Deposit Account Sat Aug 11 08:30:20 ADT 2012: current balance is 1500.00  
Checking Account: current balance is 200.0  
  
Customer: Soley, Maria  
Time Deposit Account Sat Aug 11 08:30:20 ADT 2012: current balance is 200.0  
Checking Account: current balance is 150.0  
BUILD SUCCESSFUL (total time: 0 seconds)

You've opened the Bank.java file and now want to modify the Bank class to implement the Singleton design pattern. Follow these steps:

Code

package com.example;  
  
public class Bank {  
  
    private Customer[] customers;  
    private int numberOfCustomers;  
  
    public Bank() {  
        customers = new Customer[10];  
        numberOfCustomers = 0;  
    }  
  
    public void addCustomer(String f, String l) {  
        int i = numberOfCustomers++;  
        customers[i] = new Customer(f, l);  
    }  
  
    public int getNumOfCustomers() {  
        return numberOfCustomers;  
    }  
  
    public Customer getCustomer(int customer\_index) {  
        return customers[customer\_index];  
   }  
}

* change the constructor’s access level to private   
    
  **Code**  
  public class Bank {  
       
      private Customer[] customers;  
      private int numberOfCustomers;  
    
      private Bank() {  
          customers = new Customer[10];  
          numberOfCustomers = 0;  
     }
* add a new static final field named instance  of type Bank and initialize it to a new Bank instance, and   
    
  **Code**  
  public class Bank {  
      private static final Bank instance = new Bank;  
      private Customer[] customers;  
      private int numberOfCustomers;  
    
      private Bank() {  
          customers = new Customer[10];  
          numberOfCustomers = 0;  
     }
* create a static method named getInstance that returns the value stored in the instance field   
    
  **Code**  
  public class Bank {  
      private static final Bank instance = new Bank;  
      private Customer[] customers;  
      private int numberOfCustomers;  
    
      private Bank() {  
          customers = new Customer[10];  
          numberOfCustomers = 0;  
      }  
    
      public static Bank **getInstance()** {  
          return instance;  
     }

Supplement

*Selecting the link title opens the resource in a new browser window.*

**Learning Aid**

Access the learning aid, [Bank.java](javascript:doWindow('../t15/misc/cw_jsep_a01_it_enus_t15002.html'))[,](javascript:doWindow('../t15/misc/cw_jsep_a01_it_enus_1502.html')) to view the full Bank.java file.

You now modify the SingletonBankingMain class to use the Bank singleton. In the SingletonBankingMain.java file, do the following:

Code

package com.example;  
  
import java.util.Calendar;  
  
public class SingletonBankingMain {  
  
    public static void main(String[] args) {  
        Bank bank = new Bank();  
        initializeCustomers(bank);  
  
        // run the customer report  
        CustomerReport report = new CustomerReport();  
        report.setBank(bank);  
        report.generateReport();  
    }  
  
    private static void initializeCustomers(Bank bank) {  
        Customer customer;  
          
        //90 day term  
        Calendar cal = Calendar.getInstance();  
        cal.add(Calendar.DAY\_OF\_YEAR, 180);  
...

* replace any calls to the Bank constructor with calls to the getInstance method   
    
  **Code**  
  package com.example;  
    
  import java.util.Calendar;  
    
  public class SingletonBankingMain {  
    
      public static void main(String[] args) {  
          Bank bank = Bank.getInstance();
* create a second local Bank reference named bank2 in the main method, and initialize it using the getInstance method   
    
  **Code**  
          Bank bank2 = Bank.getInstance();
* use reference equality checking to determine if bank and bank2 reference the same object, and   
    
  **Code**  
          if(bank == bank2) {  
              System.out.println("bank and bank2 are the same object");  
          }
* initialize only the second Bank, but run the report on the first Bank   
    
  **Code**  
          initializeCustomers(bank2);  
    
          // run the customer report  
          CustomerReport report = new CustomerReport();  
          report.setBank(bank);  
          report.generateReport();

Supplement

*Selecting the link title opens the resource in a new browser window.*

**Learning Aid**

Access the learning aid, [SingletonBankingMain.java](javascript:doWindow('../t15/misc/cw_jsep_a01_it_enus_1503.html')), to access the full SingletonBankingMain.java file.

Now you run the project. A report of all customers and their accounts should be displayed.

Graphic

*The report is as follows:  
  
bank and bank2 are the same object  
CUSTOMERS REPORT  
==================  
  
Customer: Simms, Jane  
Time Deposit Account Sat Aug 11 08:33:09 ADT 2012: current balance is 500.00  
Checking Account: current balance is 200.0  
  
Customer: Bryant, Owen  
Checking Account: current balance is 200.0  
  
Customer: Soley, Tim  
Time Deposit Account Sat Aug 11 08:33:09 ADT 2012: current balance is 1500.00  
Checking Account: current balance is 200.0  
  
Customer: Soley, Maria  
Time Deposit Account Sat Aug 11 08:33:09 ADT 2012: current balance is 200.0  
Checking Account: current balance is 150.0  
BUILD SUCCESSFUL (total time: 0 seconds)*

**4. Recognizing nested classes**

You now take an existing application and attempt to recognize the declaration and use of various types of nested classes.   
  
You have been given a small project that contains only two .java files. Although there are only two .java files, there may be multiple Java classes being created. You want to determine the number of classes being created.

You open the NestedClasses project and expand the project directory structure.

Graphic

*The NestedClasses project is open in NetBeans and the OuterClass.java file is open in the code editor. The project includes one other file, which is NestedClassesMain.java.*

When you run the project, this is the output.

Code

In local class  
42  
In an anonymous local class method  
42  
In an anonymous class method  
42  
In a inner class method  
43  
In a static nested class method  
BUILD SUCCESSFUL (total time: 0 seconds)

You want to count the number of classes created in the OuterClass.java file. This will include the total number of

* classes created in this file
* top-level classes created in this file
* nested classes created in this file
* inner classes
* member classes
* local classes
* anonymous classes, and
* static nested classes

You can use the Files tab in NetBeans to access the .class files created in the build\classes folder for a project.

Graphic

*There are 11 .class files in the project. They are  
  
NestedClassesMain.class  
OuterClass$1.class  
OuterClass$1LocalClass.class  
OuterClass$2.class  
OuterClass$3.class  
OuterClass$A$B$1C.class  
OuterClass$A$B.class  
OuterClass$A.class  
OuterClass$InnerClass.class  
OuterClass$StaticNestedClass.class  
OuterClass.class*

When you study this code, you'll find 10 classes in total consisting of one top-level class and nine nested classes. The nine nested classes include eight inner classes and one static nested class. The inner classes consist of three member classes, two local classes, and three anonymous classes.

Graphic

*The graphic provides a graphical illustration of the hierarchical structure of the classes.*

Supplement

*Selecting the link title opens the resource in a new browser window.*

**Code window**

Access the code for the [OuterClass.java](javascript:doWindow('../t15/misc/cw_jsep_a01_it_enus_1501.html')) file.

There are several examples of classes included in this code:

* a local inner class   
  *The local inner class declaration is:  
    
  class LocalClass*   
    
  **Code**  
      public void method1() {  
          // local classes are inner classes but not members of the outer class  
          class LocalClass {  
    
              public void localPrint() {  
                  System.out.println("In local class");  
                  System.out.println(x);  
              }  
          }
* an anonymous local inner class   
  *The anonymous local inner class declaration is:  
    
  Runnable r = new Runnable()*   
    
  **Code**  
      public void method2() {  
          Runnable r = new Runnable() {  
    
              @Override  
              public void run() {  
                  System.out.println("In an anonymous local class method");  
                  System.out.println(x);  
              }  
          };  
          r.run();  
     }
* an anonymous inner class   
  *The anonymous inner class declaration is:  
    
  public Runnable r = new Runnable()*   
    
  **Code**  
      public Runnable r = new Runnable() {  
    
          @Override  
          public void run() {  
              System.out.println("In an anonymous class method");  
              System.out.println(x);  
          }  
     };
* a member inner class, and   
  *The member inner class declaration is:  
    
  public class InnerClass*   
    
  **Code**  
      public class InnerClass {  
    
          // hides OuterClass x  
          public int x = 43;  
          //static requires final  
          public static final int y = 44;  
    
          public void innerPrint() {  
              System.out.println("In a inner class method");  
              System.out.println(x);  
          }  
     }
* a static nested class   
    
  **Code**  
    
      public static class StaticNestedClass {  
    
          public void staticNestedPrint() {  
              System.out.println("In a static nested class method");  
              //compile error  
              //System.out.println(x);  
          }  
     }

**5. Summary**

In this topic you've learned how to use the static and final keywords.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#top)

Designing Java Classes

Learning Objectives

*After completing this topic, you should be able to*

* *to override methods and utilize the instanceof operator*
* *apply the abstract keyword*
* *apply singleton design and recognize nested classes*

**1. Exercise overview**

In this exercise, you’re required to design Java classes.

This involves the following tasks:

* overriding methods and utilize the instanceof operator
* applying the abstract keyword, and
* applying singleton design and recognize nested classes

**2. Overriding methods**

You've written a program to manage the personnel information of your company.

Question

You have created a class named Manager that extends Employee. You want to override the toString() method defined in Employee to add Manager specific information. Which segment of code ensures that information from the base class Employee is also displayed?

**Code**  
@Override  
    public String toString() {  
        return INSERT THE MISSING CODE +  
             "\nDepartment: " + getDeptName();  
    }

**Options:**

1. super.toString()
2. new super.toString()
3. Employee.toString()
4. toString()

Answer

***Option 1:****Correct. You use the super keyword to refer to the base, or super class that Manager is based on.*

***Option 2:****Incorrect. To call the parent class, you need to use the super keyword, which does not require the new keyword.*

***Option 3:****Incorrect. The toString method of Employee is not a static method and cannot be called in this way.*

***Option 4:****Incorrect. You need to use either a class instance or the super keyword to invoke this method.*

**Correct answer(s):**

1. super.toString()

Question

You have written a class that compares the various subclasses of Employee. Which statement determines if the variable newEmp is of the type Director?

**Options:**

1. if (newEmp == Director)
2. if (newEmp.equals(Director))
3. if (newEmp instanceof Employee)
4. if (newEmp instanceof Director)

Answer

***Option 1:****Incorrect. The == operator is used to compare two object references to see if they are referring to the same object.*

***Option 2:****Incorrect. Unless overridden, the equals method behaves the same as the == operator.*

***Option 3:****Incorrect. While this is the correct keyword, this comparison would not determine if newEmp id is of type Director. Also note that all subclasses of Employee would return a true result for this comparison.*

***Option 4:****Correct. Using the instanceof keyword allows you to determine if an object is of a particular type.*

**Correct answer(s):**

4. if (newEmp instanceof Director)

Question

Given that Manager and Engineer are each direct subclasses of Employee, which code segment is correct?

**Code**  
Employee e = new Employee();  
Manager m = new Manager();  
Engineer en = new Engineer();

**Options:**

1. Employee e2= (Employee)m;
2. Manager m2 = (Manager)e;
3. Employee e3 = (Employee)en;
4. Engineer en2 = (Engineer)m;

Answer

***Option 1:****Correct. Since Manager is a direct subclass of Employee, this cast is possible.*

***Option 2:****Incorrect. While this would compile, it would throw a runtime error since Employee cannot be cast down to Manager.*

***Option 3:****Correct. Since Engineer is a direct subclass of Employee, this cast is possible.*

***Option 4:****Incorrect. While both Manager and Engineer are each direct subclasses of Employee, you cannot cast across class hierarchies.*

**Correct answer(s):**

1. Employee e2= (Employee)m;  
3. Employee e3 = (Employee)en;

**3. Applying the abstract keyword**

Question

Which statement correctly defines an abstract method?

**Options:**

1. public String getDescription();
2. public abstract String getDescription();
3. public String abstract getDescription();
4. public abstract String getDescription(){ };

Answer

***Option 1:****Incorrect. You must specify that the method is abstract by using the abstract keyword.*

***Option 2:****Correct. This statement correctly defines the abstract method.*

***Option 3:****Incorrect. The abstract keyword must precede the return type.*

***Option 4:****Incorrect. You cannot define a method body as noted by the pair of parentheses.*

**Correct answer(s):**

2. public abstract String getDescription();

Question

You want to use the random method of the Math class in your program without having to refer to it by its full name every time. Which segment completes the sample so that you can do this?

**Code**  
INSERT THE MISSING CODE  
  
public class StaticImportClass {  
    public static void main(String[] args) {  
        double d = random();  
    }  
}

**Options:**

1. import static java.lang.Math.\*;
2. import java.lang.Math.\*;
3. static import java.lang.Math.\*;
4. import static.java.lang.Math.\*;

Answer

***Option 1:****Correct. This defines a static import statement that makes the static members of a class available under their simple name.*

***Option 2:****Incorrect. You need to use the static keyword with the import statement to define a static import.*

***Option 3:****Incorrect. The static keyword must come after the import keyword.*

***Option 4:****Incorrect. The static keyword is not part of the reference to the package.*

**Correct answer(s):**

1. import static java.lang.Math.\*;

**4. Using Singleton design**

Question

You decide to implement the Singleton pattern in your class definition of a user account. Which segment correctly completes the sample to implement the Singleton pattern?

**Code**  
public class UserAccount{  
INSERT THE MISSING CODE =   
new UserAccount();  
   
      private UserAccount () { }  
   
      public static UserAccount getInstance() {  
       return instance;  
        }  
}

**Options:**

1. private static final UserAccount instance
2. private static Singleton UserAccount instance
3. public static final UserAccount instance
4. private static final UserAccount getInstance

Answer

***Option 1:****Correct. To complete the implementation of the pattern, you need to define a private variable named instance that cannot be overridden.*

***Option 2:****Incorrect. Singleton does not actually refer to a type in Java.*

***Option 3:****Incorrect. You need to use the private access modifier to only allow in class access to the value.*

***Option 4:****Incorrect. The field is named instance. The getInstance method is used to allow public access to the private constructor.*

**Correct answer(s):**

1. private static final UserAccount instance

Question

Which statements best describe why you would want to use an inner class?

**Options:**

1. They prevent people from modifying the values in the inner class once initialized
2. They provide a logical grouping of classes
3. They are used to prevent any instances of that class from being created
4. They make for more readable, maintainable code

Answer

***Option 1:****Incorrect. You use the final modifier to prevent the change of variables or methods.*

***Option 2:****Correct. If a class is useful to only one other class, it is logical to embed it in that class and keep the two together.*

***Option 3:****Incorrect. You actually use abstract classes to prevent instances of a class from being created.*

***Option 4:****Correct. Nesting small classes within top-level classes places the code closer to where it is used*

**Correct answer(s):**

2. They provide a logical grouping of classes  
4. They make for more readable, maintainable code

A method has been overridden and instanceof used to determine a class type. An abstract method was used and a class imported using the static keyword. The Singleton pattern has been recognized and the use of nested classes considered.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsep_a01_it_enus/output/html/course_transcript.html#top)

© 2016 Skillsoft Ireland Limited