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

**Course Transcript**

Java SE7 Fundamentals: Overloading, Validation, and Object-Oriented Concepts

**Methods, Overloading, and Encapsulation in Java**

| [1. Methods and Overloading](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#t3) |

| [2. Java Encapsulation and Validation](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#t24) |

| [3. Creating Constructors to Initialize Objects](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#t25) |

| [4. Using Methods, Overloading, and Encapsulation in Java](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#t8) |

**Advanced Object-Oriented Concepts in Java**

| [1. Using Advanced Object-Oriented Concepts](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#t15) |

| [2. Polymorphism, Interfaces, and the Object class](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#t17) |

Methods and Overloading

Learning Objectives

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

* *create and invoke methods*
* *use method overloading*

**1. Creating and invoking methods**

You can create and invoke most methods using this basic syntax.

Syntax

*[modifiers] return\_type method\_identifier ([arguments]) {  
    method\_code\_block  
}*

For simple methods, such as the one in this example, don't receive any arguments or return a value.

Code

public void display () {  
    System.out.println("Shirt ID: " + shirtID);  
    System.out.println("Shirt description:" + description);  
    System.out.println("Color Code: " + colorCode);  
    System.out.println("Shirt price: " + price);  
} // end of display method

**void**

*The void keyword indicates that the method doesn’t return a value.*

**()**

*Empty parentheses indicate that no arguments are passed to the method.*

In this example, a method in a different class is being invoked. The display() method is being called, but the Shirt object has not had any of its fields set.

Graphic

*The relevant code is display().*

Code

public class ShirtTest {  
    public static void main (String args[]) {  
        Shirt myShirt;  
        myShirt = new Shirt();  
        myShirt.display();  
    }  
}

Because none of the fields for the Shirt object have been set, the default values for those fields display in the output.

Graphic

*The highlighted code shows the default field values: Item ID is 0, Item description is description required, Color Code is U, and Item price is 0.0*

Code

Item ID: 0  
Item description:-description required-  
Color Code: U  
Item price: 0.0

The ShirtTest class calls the display() method from within another method - the main method. Therefore, the main method is referred to as the calling method because it is invoking or "calling"' another method to do some work. Conversely, the display method is referred to as the worker method because it does some work for the main method.   
  
When a calling method calls a worker method, the calling method stops execution until the worker method is done. After the worker method has completed, program flow returns to the point after the method invocation in the calling method.  
  
There is no limit to the number of method calls that a calling method can make, and the calling method and the worker method can be in the same class or in different classes.

Graphic

*The relevant code is 'public static void main (String args[]) {' and 'myShirt.display();'.*

Code

public class ShirtTest {  
    public static void main (String args[]) {  
        Shirt myShirt;  
        myShirt = new Shirt();  
        myShirt.display();  
    }  
}

Methods that pass arguments and return values typically do so in three steps.

**Step 1**

In step 1, the value is passed from the calling method to the called method.

**Step 2**

In step 2, the value is received by the called method.

**Step 3**

In step 3, the value is returned to the calling method.

You can also create methods with parameters. For example, the setFloor() method receives a value of int type and gives it the name of desiredFloor.   
  
desiredFloor becomes a local variable whose scope is the method. It is called - in this case, from a calling method in another class - by using the dot notation and including the argument.

Code

Elevator theElevator = new Elevator();  
  
theElevator.setFloor( 4 ); // Send elevator to the fourth floor  
  
public void setFloor( int desiredFloor ) {  
    while (currentFloor != desiredFloor){  
    if (currentFloor < desiredFloor){  
        goUp();  
    }  
    else {  
        goDown();  
    }  
}

Note

*A variable defined in the method declaration is called a method parameter, whereas a value passed into the method call is called an argument.*

In this example, there is a caller method and a worker method.

Code

Elevator theElevator = new Elevator();  
  
theElevator.setFloor( 4 ); // Send elevator to the fourth floor  
  
public void setFloor( int desiredFloor ) {  
    while (currentFloor != desiredFloor){  
    if (currentFloor < desiredFloor){  
        goUp();  
    }  
    else {  
        goDown();  
    }  
}

**setFloor( 4**

*In this caller method, a call to the setFloor() method passes the value 4, of type int.*

**int desiredFloor**

*In this worker method, the setFloor() method receives an argument of type int, naming it desiredFloor.*

You can also create methods with return values, such as in this example, which shows the checkDoorStatus() method being called by the caller method.   
  
The checkDoorStatus() defines that it will return a Boolean. Any single type can be defined here, or the keyword void is used if the method does not return a value.  
  
The value is returned to the calling statement by the return statement. Because the method has been declared with a return type of boolean, an error will occur if there is no return or if the return is of an incorrect type.

Code

... < lines of code omitted > ...  
  
boolean isOpen = theElevator.checkDoorStatus() // Is door open?  
  
public class Elevator {  
    public boolean doorOpen=false;  
    public int currentFloor = 1;  
  
    ... < lines of code omitted > ...  
  
    public boolean checkDoorStatus() {  
        return doorOpen ;  
    }  
}

This example contains variables and fields to support a value being returned.

Code

... < lines of code omitted > ...  
  
boolean isOpen = theElevator.checkDoorStatus() // Is door open?  
  
public class Elevator {  
    public boolean doorOpen=false;  
    public int currentFloor = 1;  
  
    ... < lines of code omitted > ...  
  
    public boolean checkDoorStatus() {  
        return doorOpen ;  
    }  
}

**isOpen**

*The local variable isOpen indicates if the elevator door is open.*

**boolean doorOpen=false**

*The Elevator class has a doorOpen field to indicate the state of the elevator door.*

**boolean**

*The type returned by the method is defined before the method name.*

**doorOpen**

*The return statement returns the value in doorOpen.*

Calling a method in the same class is more straightforward than calling a method in a different class. You can simply use the method name without a reference and dot notation. This is the same as when accessing a field - you can simply use the field name.  
  
However, if you have local variables with similar names and you want to make it obvious that your code is accessing a field or method of the current object, you can use the this keyword with dot notation. The keyword this is a reference to the current object. For example, you can replace !checkDoorStatus() with !(this.checkDoorStatus()).

Graphic

*The highlighted code is !checkDoorStatus() .*

Code

public class Elevator {  
public boolean doorOpen=false;  
public int currentFloor = 1;  
public final int TOP\_FLOOR = 5;  
public final int BOTTOM\_FLOOR = 1;  
public void openDoor() {  
    // Check if door already open  
    if ( !checkDoorStatus() ) {  
        // door opening code  
    }  
}

There are various advantages to using methods:

* they make programs more readable and easier to maintain
* they make development and maintenance quicker
* they are central to reusable software, and
* they allow separate objects to communicate and to distribute the work performed by the program

You want to create a class to order more than one shirt and then display the total order value of all the shirts.  
  
In this case you have already coded the Shirt and Order classes. You now create a new Java Main Class called OrderTest.

Code

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

Next you add code to the main method that will add a shirt to a new order and display the total amount of the order. These are the first three steps in this procedure:

* add the code for creating and initializing the two new objects - the shirt and the order   
  *The code for this is:  
  Order order = new Order();  
  Shirt shirt = new Shirt();*   
    
  **Code**  
  public static void main(String[] args){   
                      Order order = new Order();  
                      Shirt shirt = new Shirt();  
  }
* add code to declare and initialize a local variable of type double, and   
  *The relevant code is:  
  double totalCost = 0.0;*   
    
  **Code**  
  public static void main(String[] args){   
                      Order order = new Order();  
                      Shirt shirt = new Shirt();  
                      double totalCost = 0.0;  
  }
* assign a value to the price field of the Shirt object   
  *The code for this is:  
  shirt.price = 14.99;*   
    
  **Code**  
  public static void main(String[] args){   
                      Order order = new Order();  
                      Shirt shirt = new Shirt();  
                      double totalCost = 0.0;  
                      shirt.price = 14.99;  
  }

You continue adding code to the class using these steps:

Code

public static void main(String[] args){   
                    Order order = new Order();  
                    Shirt shirt = new Shirt();  
                    double totalCost = 0.0;  
                    shirt.price = 14.99;  
}

* invoke the addShirt method of the Order object using the method argument, and   
  *The code that does this is:  
  totalCost = order.addShirt(shirt);*  
    
  **Code**  
  public static void main(String[] args){   
                      Order order = new Order();  
                      Shirt shirt = new Shirt();  
                      double totalCost = 0.0;  
                      shirt.price = 14.99;  
                      totalCost = order.addShirt(shirt);   
  }
* display the return value with a suitable label Example output   
  *The relevant code is:  
  System.out.println("Total amount for the order is $" +totalCost);*   
    
  **Code**  
  public static void main(String[] args){   
                      Order order = new Order();  
                      Shirt shirt = new Shirt();  
                      double totalCost = 0.0;  
                      shirt.price = 14.99;  
                      totalCost = order.addShirt(shirt);   
                      System.out.println("Total amount for the order is $" + totalCost);  
  }

You then save and compile your program and test the order process by running the OrderTest class.

Code

run:  
Total amount for the order is $14.99  
BUILD SUCCESSFUL (total time: 0 seconds)

In the main method of OrderTest, you create additional Shirt objects, assign values to the price field of each new Shirt object, and add the Shirt objects to your order by invoking the addShirt method.  
  
You can then save and compile the program and once again, run the OrderTest class to test it. You must make sure that the amount displayed is the total of all of the shirt prices.

Graphic

*The code to create the objects is:  
shirt2 = new Shirt(), shirt3 = new Shirt();  
The highlighted piece of code to assign values is:  
shirt2.price = 23.55;  
shirt3.price = 49.99;   
The highlighted piece of code to add objects to the order is:  
order.addShirt(shirt2);  
totalCost = order.addShirt(shirt3);*

Code

public static void main(String[] args){   
                    Order order = new Order();  
                    Shirt shirt = new Shirt(), shirt2 = new Shirt(), shirt3 = new Shirt();  
                    double totalCost = 0.0  
                    shirt.price = 14.99;  
                    shirt2.price = 23.55;   
                    shirt3.price = 49.99;   
                    order.addShirt(shirt);   
                    order.addShirt(shirt2);  
                    totalCost = order.addShirt(shirt3);  
                    System.out.println("Total amount for the order is $" + totalCost);

Note

*The addShirt method adds the price of the shirt argument object to the totalPrice field of the Order object. Therefore, the totalPrice value grows with each addition of a shirt. You only need to capture the return value of the final addShirt method call to get the totalCost value.*

Question

Which statements best describe invoking methods?

**Options:**

1. Calling methods don’t have limitations on method calls
2. The calling method and worker method must be in different classes
3. You can invoke methods in any order
4. You invoke the worker method and the calling method in the same way

Answer

***Option 1:****Correct. There is no limit to the number of method calls that a calling method can make. Caller methods invoke other methods to do some work.*

***Option 2:****Incorrect. The calling method and the worker method can be in the same class or in different classes.*

***Option 3:****Correct. You can invoke methods in any order. Methods do not need to be completed in the order in which they are listed in the class where they are declared.*

***Option 4:****Incorrect. The way you invoke the worker method is different depending on whether it is in the same class or in a different class from the calling method.*

**Correct answer(s):**

1. Calling methods don’t have limitations on method calls  
  
3. You can invoke methods in any order

**2. Using method overloading**

ASCII character values encode lowercase a to z from 97 to 122. You can use the Maths utilities in Java, specifically Math.random(), to generate a number in that  
range and put it into a char, and you can then use the concatenation operator to build a String as shown here.

Graphic

*The code that illustrates this is:  
char asciiChar = (char)(Math.random() \* 26 + 97);*

Code

String name = "Lenny";  
String guess = "";  
int numTries = 0;  
while (!guess.equals(name.toLowerCase())) {  
    guess = "";  
    while (guess.length() < name.length()) {  
        char asciiChar = (char)(Math.random() \* 26 + 97);  
        guess = guess + asciiChar;  
    }  
    numTries++;  
}  
System.out.println(name + " found after " + numTries + " tries!");

Note

*Java actually uses Unicode, not ASCII, but the first 128 characters in Unicode and ASCII are the same.*

Here are some methods including random() from the Math class.

**pow(double a, double b)**

The pow(double a, double b) method returns the value of the first argument raised to the power of the second argument.

**random()**

The random()method returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

**rint(double a)**

The rint(double a) method returns the double value that is closest in value to the argument and is equal to a mathematical integer.

**round(double a)**

The round(double a)method returns the closest long to the argument, with ties rounding up.

The random() method returns a double between 0 and 1. So to generate a double between 0 and 10, you multiply by 10. Or to generate a double between 1 and 10, you multiply by 9 and add 1.

Code

Math.random \* 10

Often you will want an integer rather than a double. Then all you need to do is cast to int or, in this case, to char.

Graphic

*The relevant code is (char).*

Code

String name = "Lenny";  
String guess = "";  
int numTries = 0;  
    while (!guess.equals(name.toLowerCase())) {  
        guess = "";  
        while (guess.length() < name.length()) {  
            char asciiChar = (char)(Math.random() \* 26 + 97);  
            guess = guess + asciiChar;  
        }      
        numTries++;  
}  
System.out.println(name + " found after " + numTries + " tries!");

The random() method is static, as are all the methods in Math. That means that Math does not need to be instantiated to call any of its methods. In fact, Math cannot be instantiated.  
  
You can call the static methods of a class with this syntax.

Syntax

*classname.method\_name*

You can access methods and variables by creating an object of the class that the method or variable belongs to, and invoking the method or accessing the variable - if it is a public variable. Methods and variables that are unique to an instance are called instance methods and instance variables.  
  
However, some methods do not require object instantiation, such as the main method. These are called class methods or static methods. You can invoke them without creating an object first.  
  
Similarly, the Java programming language allows you to create static variables or class variables, which you can use without creating an object.

Methods and non-local variables can be static.They belong to the class and not the object, and are declared using the static keyword.

Code

static Properties getProperties()

You can work with static methods and variables in the following ways:

* by invoking static methods   
  **Syntax**  
  *Classname.method();*
* by accessing static variables in another class, and   
  **Syntax**  
   *Classname.attribute\_name;*
* by accessing static variables in the same class   
  **Syntax**  
  *attribute\_name;*

This is an example of a method that could be added to the Shirt class to convert numerical shirt sizes to sizes such as small, medium, or large.  
  
This method is a static method because it does not directly use any attributes of the Shirt class, and you might want to invoke the method even if you do not have a Shirt object.  
  
The convertShirtSize method accepts a numerical size, determines the corresponding character size, and returns the character size.

Code

public static char convertShirtSize(int numericalSize) {  
    if (numericalSize < 10) {  
        return 'S';  
    }  
    else if (numericalSize < 14) {  
        return 'M';  
    }  
    else if (numericalSize < 18) {  
        return 'L';  
    }  
    else {  
        return 'X';  
    }  
}

To access the static method convertShirtSize() of the Shirt class, you use this code.

Code

char size = Shirt.convertShirtSize(16);

You can also use the static keyword to declare a class variable. This means that there can be only one copy of the variable in memory associated with a class, not a copy for each object instance.  
  
In this example, salesTAX is a static variable. You can access it from any method in any class by using the class name of its class.

Code

static double salesTAX = 8.25;

If salesTAX is in a class called TaxUtilities, you could access it by using this code.

Code

TaxUtilities.salesTAX

If TaxUtilities has static or instance methods, those methods can access the variable by name without having to use the class name.

Code

salesTAX

Variables can have both the static and final modifier to indicate that there is only one copy of the variable and that the contents of the variable cannot be changed. The PI variable in the Math class is a static final variable.

Code

double myPI;  
myPI = Math.PI;

Certain Java class libraries contain only static methods and variables. These are

**System, and**

The System class contains utility methods for handling operating system-specific tasks – they do not operate on an object instance. For example, the getProperties method of the System class gets information about the computer that you are using.  
  
Functionalities of the System class include retrieving environment variables, accessing the standard input and output streams, and exiting the current program.

**Math**

The Math class contains utility methods for math operations, such as those that are exponential, logarithmic, trigonometric, random, as well as access to common mathematical constants such as the value PI.

You should declare a static method or variable when

* performing the operation on an individual object or associating the variable with a specific object type is not important
* accessing the variable or method before instantiating an object is important, and
* the method or variable does not logically belong to an object, but possibly belongs to a utility class, such as the Math class, included in the Java API

This code includes a method that has two parameters - the initial sum of money and the interest rate - and returns the number of years required to double that initial sum.  
  
The method signature of a method is the unique combination of the method name and the number, types, and order of its parameters. The method signature does not include the return type.

Graphic

*The method signature is getYearsToDouble(int initialSum, int interest).*

Code

public int getYearsToDouble(int initialSum, int interest) {  
    int interest = 7; // per cent  
    int years = 0;  
    int currentSum = initialSum \* 100; // Convert to pennies  
    int desiredSum = currentSum \* 2;  
    while( currentSum <= desiredSum) {   
        currentSum += currentSum \* interest/100;   
        years++;   
    }   
    return years;   
}

In the Java programming language, there can be several methods in a class that have the same name but different arguments - so the method signature is different. This concept is called method overloading. Overloaded methods may have different or similar functionality and are widely used in the Java foundation classes.  
  
Just as you can distinguish between two students named Jim in the same class by calling them "Jim in the green shirt" and "Jim with the beeper," you can distinguish between two methods by their name and arguments.

With method overloading, you can create several methods with the same name and different signatures.  
  
This code shows three methods to add two numbers, such as two int types or two float  
types.

Code

public final class Calculator {  
  
    public static int sum(int numberOne, int numberTwo){  
        System.out.println("Method One");  
        return numberOne + numberTwo;  
    }  
  
    public static float sum(float numberOne, float numberTwo) {  
        System.out.println("Method Two");  
        return numberOne + numberTwo;  
    }  
  
    public static float sum(int numberOne, float numberTwo) {  
        System.out.println("Method Three");  
        return numberOne + numberTwo;  
   }

**public static int sum(int numberOne, int numberTwo){  
       System.out.println("Method One");  
       return numberOne + numberTwo;**

*This sum method accepts two int arguments and returns an int value*

**public static float sum(float numberOne, float numberTwo) {  
       System.out.println("Method Two");  
       return numberOne + numberTwo;**

*This sum method accepts two float arguments and returns a float value.*

**public static float sum(int numberOne, float numberTwo) {  
       System.out.println("Method Three");  
       return numberOne + numberTwo;**

*This sum method accepts an int and a float as arguments and returns a float.*

To invoke any of the sum methods, the compiler compares the method signature in your method invocation against the method signatures in a class.

Code

public final class Calculator {  
  
    public static int sum(int numberOne, int numberTwo){  
        System.out.println("Method One");  
        return numberOne + numberTwo;  
    }  
  
    public static float sum(float numberOne, float numberTwo) {  
        System.out.println("Method Two");  
        return numberOne + numberTwo;  
    }  
  
    public static float sum(int numberOne, float numberTwo) {  
        System.out.println("Method Three");  
        return numberOne + numberTwo;  
   }

This example has a main method that invokes each of the previous sum methods of the Calculator class.

Code

public class CalculatorTest {  
  
    public static void main(String [] args) {  
  
        int totalOne = Calculator.sum(2,3);  
        System.out.println("The total is " + totalOne);  
  
        float totalTwo = Calculator.sum(15.99F, 12.85F);  
        System.out.println(totalTwo);  
  
        float totalThree = Calculator.sum(2, 12.85F);  
        System.out.println(totalThree);  
    }  
}

Many methods in the Java API are overloaded including the System.out.println method.  These are some of the variations of the println method.

**void println()**

The void println() method terminates the current line by writing the line separator string.

**void println(boolean x)**

The void println(boolean x) method prints a Boolean value and then terminates the line.

**void println(char x)**

The void PrintLn(char x) method prints a character and then terminates the line.

**void println(char[] x)**

The void println(char[] x) method prints an array of characters and then terminates the line.

**void println(double x)**

The void println(double x) method prints a double and then terminates the line.

There are several other variations of the println method in the Java API that are overloaded.

**void println(float x)**

The void println(float x) method prints a float and then terminates the line.

**void println(int x)**

The void println(int x) method prints an int and then terminates the line.

**void println(long x)**

The void println(long x) method prints a long and then terminates the line.

**void println(Object x)**

The void println(Object x) method prints an object and then terminates the line.

**void println(String x)**

The void println(String x) method prints a string and then terminates the line.

You want to write a Customer class with an overloaded method called setCustomerInfo.   
  
To begin, you create a new Java Class called Customer. Within this class, you declare five fields and initialize them.

Code

public class Customer {  
    public int customerID = 0;  
    public String name = "-name required-";  
    public String address = "-address required-";  
    public String phoneNumber = "-phone required-";  
    public String eMail = "-email required-";  
      
}

Within the Customer class, you add two overloaded methods called setCustomerInfo. Within each method, you assign each argument in the method to its corresponding field.

Code

public void setCustomerInfo(int ID, String nm, String addr, String phNum)  
public void setCustomerInfo(int ID, String nm, String addr, String phNum, String email)

Then you create a display method to display the values of all the member fields of the Customer class, and save and compile the program.

Code

public void display(){  
        System.out.println("Customer ID: "+ customerID);  
        System.out.println("Customer name: " + name);  
        System.out.println("Customer address: "+ address);  
        System.out.println("Customer phone: "+ phoneNumber);  
        System.out.println("Customer email: "+ eMail);  
}

In the CustomerTest class, you modify the main method so that it can be used to test the overloaded methods of the Customer class. You use these steps:

* add code that creates two object references to different Customer objects   
  *The code for this is:  
  Customer c1 = new Customer(), c2 = new Customer();*   
    
  **Code**  
  public static void main(String[] args){   
      Customer c1 = new Customer(), c2 = new Customer();  
  }
* use each variation of the setCustomerInfo method to provide information for each customer object, and   
  *The code that does this is:  
  c1.setCustomerInfo(1,”Harry”,”234 Maple St”,”505-123-4545”);  
  c2.setCustomerInfo(2,”Sally”,”567 Oak St”, ”505-123-2323”,”sally@gmail.com”);*   
    
  **Code**  
  public static void main(String[] args){   
      Customer c1 = new Customer(), c2 = new Customer();  
      c1.setCustomerInfo(1,"Harry","234 Maple St","505-123-4545");  
      c2.setCustomerInfo(2,"Sally","567 Oak St",   
                           "505-123-2323","sally@gmail.com");
* add code to display the contents of each Customer object   
  *The relevant code is:  
  c1.display();  
  c2.display();*   
    
  **Code**  
  public static void main(String[] args){   
      Customer c1 = new Customer(), c2 = new Customer();  
      c1.setCustomerInfo(1,"Harry","234 Maple St","505-123-4545");  
      c2.setCustomerInfo(2,"Sally","567 Oak St",   
                           "505-123-2323","sally@gmail.com");  
      c1.display();  
      c2.display();

You then save and compile the program and run the CustomerTest file to test the program.

Code

run:  
customer ID: 1  
Customer name: Harry  
Customer address: 234 Maple St  
Customer phone: 505-123-4545  
Customer email: -email required-  
customer ID: 2  
Customer name: Sally  
Customer address: 567 Oak St  
Customer phone: 505-123-2323  
Customer email: sally@gmail.com  
BUILD SUCCESSFUL (total time: 0 seconds)

Question

Which statements best describe overloaded methods?

**Options:**

1. They have the same name and signature
2. They may have different functionality
3. They are not available in the fundamental classes
4. The compiler uses method signatures to determine which method to invoke

Answer

***Option 1:****Incorrect. In the Java programming language, there can be several methods in a class that have the same name but different arguments, so the method signature is different.*

***Option 2:****Correct. Overloaded methods may have different or similar functionality. You can differentiate between two methods by their arguments.*

***Option 3:****Incorrect. Overloaded methods are widely used in the fundamental prewritten class libraries available in Java.*

***Option 4:****Correct. To invoke overloaded methods, the compiler compares the method signature in your method invocation against the method signatures in a class.*

**Correct answer(s):**

2. They may have different functionality  
4. The compiler uses method signatures to determine which method to invoke

**3. Summary**

In this topic, you've learned how to create and invoke methods, and use method overloading.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#top)

Java Encapsulation and Validation

Learning Objective

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

* *use encapsulation in Java*

**1. Java encapsulation and validation**

Encapsulation means hiding object fields by making all fields private. This is done using getter and setter methods. In setter methods, you use code to check that values are valid.  
  
Encapsulation mandates programming to the interface. The data type of the field is irrelevant to the caller method, and the class can be changed as long as the interface remains the same.  
  
Encapsulation also encourages good Object- Orientated, or OO, design.

Note

*An interface is a type of contract specifying how a software object interacts with another object. In Java, an interface contains only method signatures constants, and nested types. It cannot be instantiated but can only be implemented by a class.*

This code shows the goUp() method and the currentFloor field. It protects an elevator from trying to go above the top floor.  
  
However, public fields, such as this one, can cause problems. The goUp() method can be circumvented - there is nothing to stop the currentFloor field from being modified directly.

Graphic

*The code example is public int currentFloor = 1;*

Code

public class Elevator {  
    public boolean doorOpen=false;  
    public int currentFloor = 1;  
    public final int TOP\_FLOOR = 10;  
    public final int MIN\_FLOOR = 1;  
  
    ... < code omitted > ...  
  
  public void goUp() {  
        if (currentFloor == TOP\_FLOOR) {  
            System.out.println("Cannot go up further!");  
        }  
        if (currentFloor < TOP\_FLOOR) {  
           currentFloor++;  
           System.out.println("Floor: " + currentFloor);  
        }  
    }  
}

In this example, the fields have all been made private. Now they cannot be accessed from a caller method that is outside this class. So any calling method that wants to control the floor that the elevator will go to must do so through its public methods.

Graphic

*The relevant code is:  
private boolean doorOpen=false;  
private int currentFloor = 1;  
private final int TOP\_FLOOR = 10;  
private final int MIN\_FLOOR = 1;*

Code

public class Elevator {  
    private boolean doorOpen=false;  
    private int currentFloor = 1;  
    private final int TOP\_FLOOR = 10;  
    private final int MIN\_FLOOR = 1;  
  
    ... < code omitted > ...  
  
  public void goUp() {  
        if (currentFloor == TOP\_FLOOR) {  
            System.out.println("Cannot go up further!");  
        }  
        if (currentFloor < TOP\_FLOOR) {  
           currentFloor++;  
           System.out.println("Floor: " + currentFloor);  
        }  
   }  
}

Methods are declared with a modifier just like fields. So some methods might be best declared with a private modifier.  
  
For example, in most elevators, the controls operated by the general public do not directly affect the elevator. Instead, a user presses a button, say, a request for an elevator to go to the fifth floor. The elevator does not respond immediately to the request, but puts the request in a queue and then eventually, perhaps after bringing users already on the elevator down to the first floor, goes to the fifth floor.  
  
It may be that the only public method needs to be requestFloor(), at least for the software that controls the buttons used by the general public.

Graphic

*The relevant code is private void setFloor().*

Code

public class Elevator {  
    ... < code omitted > ...  
  
    private void setFloor() {  
        int desiredFloor = 5;  
        while ( currentFloor != desiredFloor ){  
            if (currentFloor < desiredFloor) {  
                goUp();  
            } else {  
                goDown();  
        }  
    }  
  
  
    public void requestFloor(int desiredFloor) {  
        ... < contains code to add requested floor to a queue > ...  
    }  
}

When classes are encapsulated, other objects interact with only a few parts of every other class.  
  
In the example of the elevator, the control program that is triggered by the buttons can only call the requestFloor() method of Elevator. And, as long as Elevator implements this method, it doesn’t matter exactly how it is implemented. The method could store requests in a binary array where setting an element to true indicates that there is a request on the floor with that index. Or an ArrayList could be used to store the numbers of the floors requested.  
  
There might also be a moveElevator() method that is triggered by something, perhaps by the doors closing. Again, as long as this method moveElevator() is implemented, its implementation can be changed to change the way in which the elevator responds to requests coming in at the same time from different floors.

Graphic

*The control panel in an elevator should have public access.*

If you make attributes private, one object can access the private attributes of a second object if the second object provides public methods for each of the operations that are to be performed on the value of an attribute.  
  
For example, it is recommended that all fields of a class should be private, and those that need to be accessed should have public methods for setting and getting their values.   
  
This ensures that, at some future time, the actual field type itself could be changed, if that were advantageous. Or the getter or setter methods could be modified to control how the value could be changed, in the same way you wrote code to ensure that the currentFloor field of the elevator could not be set to an invalid value.

Code

public class Shirt {  
    private int shirtID = 0; // Default ID for the shirt  
    private String description = "-description required-"; // default  
    // The color codes are R=Red, B=Blue, G=Green, U=Unset  
    private char colorCode = 'U';  
    private double price = 0.0; // Default price for all items  
  
    public char getColorCode() {  
    return colorCode;  
    }  
    public void setColorCode(char newCode) {  
        colorCode = newCode;  
    }  
    // Additional get and set methods for shirtID, description,  
    // and price would follow  
  
} // end of class

In this example, even though the code for the Shirt class is syntactically correct, the setcolorCode method does not contain any logic to ensure that the correct values are set. So the code successfully sets an invalid color code in the Shirt object.  
  
However, because ShirtTest accesses a private field on Shirt via a setter method, Shirt can now be recorded without modifying any of the classes that depend on it.

Graphic

*The relevant code is theShirt.setColorCode('Z'); not a valid color code.*

Code

public class ShirtTest {  
    public static void main (String args[]) {  
    Shirt theShirt = new Shirt();  
    char colorCode;  
    // Set a valid colorCode  
    theShirt.setColorCode('R');  
    colorCode = theShirt.getColorCode();  
    // The ShirtTest class can set and get a valid colorCode  
    System.out.println("Color Code: " + colorCode);  
    // Set an invalid color code  
    theShirt.setColorCode('Z'); not a valid color code  
    colorCode = theShirt.getColorCode();  
    // The ShirtTest class can set and get an invalid colorCode  
    System.out.println("Color Code: " + colorCode);  
}

Here is another version of the Shirt class. However, in this class, before setting the value, the setter method ensures that the value is valid.

Code

public void setColorCode(char newCode) {  
    switch (newCode) {  
        case 'R':  
        case 'G':  
        case 'B':  
        colorCode = newCode;  
        break;  
        default:  
        System.out.println("Invalid colorCode. Use R, G, or B");  
    }  
}

If it is not valid, the colorCode field remains unchanged.

Graphic

*The relevant code is Shirt1.setColorCode('Z');*

Code

public class ShirtTest {  
    public static void main (String args[]) {  
    Shirt theShirt = new Shirt();  
    System.out.println("Color Code: " + theShirt.getColorCode());  
  
    // Try to set an invalid color code  
    Shirt1.setColorCode('Z');  
    System.out.println("Color Code: " + theShirt.getColorCode());  
}

An error message is also printed.

Code

Color Code: U Before call to setColorCode()   
Invalid colorCode.  
Color Code: U

Question

How does encapsulation protect data?

**Options:**

1. By making all fields private
2. By mandating programming to the interface
3. By encouraging good OO design
4. By allowing classes to inherit behavior from other classes
5. By making a class abstract

Answer

***Option 1:****Correct. Encapsulation means hiding object fields by making all fields private. This includes using getting and setter methods, and coding setter methods to check that values are valid.*

***Option 2:****Correct. Encapsulation mandates programming to the interface. The data type of the field is irrelevant to the caller method, and a class can be changed as long as the interface remains the same.*

***Option 3:****Correct. Encapsulation encourages good OO design and is one of the fundamental OOP concepts along with inheritance, polymorphism, and abstraction.*

***Option 4:****Incorrect. Inheritance, and not encapsulation, allows classes to inherit behavior and state that are frequently used from other classes.*

***Option 5:****Incorrect. The concept of abstraction, and not encapsulation, is used to make a class abstract. Abstract classes cannot be instantiated.*

**Correct answer(s):**

1. By making all fields private  
2. By mandating programming to the interface  
3. By encouraging good OO design

You want to create a class containing private attributes and access them in another class.   
  
To begin, you create a new project and, within this project, you create a new Java class called DateOne. You declare three public member fields of type int named day, month, and year.

Code

public class DateOne {  
    public int day, month, year;  
}

In the DateOneTest class which you will use to test the DateOne class, you add code to the main method, following these steps:

* create and initialize an object of type DateOne   
  *The relevant code is:  
  DateOne date = new DateOne();*  
    
  **Code**  
  public static void main(String args[]){   
      DateOne date = new DateOne();   
  }
* assign different numeric values to the member fields of the DateOne object   
  *The code for this is:  
  date.day = 28;  
  date.month = 07;  
  date.year = 2011;*   
    
  **Code**  
  public static void main(String args[]){   
      DateOne date = new DateOne();   
      date.day = 28;  
      date.month = 11;  
      date.year = 2011;  
  }
* display the value of the member fields of the DateOne object, and concatenate them into a single string with your choice of date formatting, and   
  *The code that does this is:  
  System.out.println(“DateOne: “+date.day+ ”/” +date.month+”/” +date.year);*   
    
  **Code**  
  public static void main(String args[]){   
      DateOne date = new DateOne();   
      date.day = 28;  
      date.month = 11;  
      date.year = 2011;  
      System.out.println(“DateOne: “+ date.day + ”/”   
          + date.month + ”/” + date.year);  
  }
* save and compile your program and run the DateOneTest class to test the program   
    
  **Code**  
  run:  
  DateOne: 28/11/2011  
  BUILD SUCCESSFUL (total time: 1 second)

Note

*The back slash character is a special character in the Java language called an escape character. If you wish to use it as part of your date format, you use two back slashes together in order to have the back slash appear in the String. For example, you might type day + "\\" + month. There are no restrictions for using a forward slash.*

Next you want to create another new Java class called DateTwo similar to DateOne. These are the steps:

* create a new Java class called DateTwo and declare three member fields of type int named day, month, and year, with the access modifier set to private and   
    
  **Code**  
  public class DateTwo {  
      private int day, month, year;  
  }
* in the DateTwoTest class, add code to the main method as you did with DateOneTest   
    
  **Code**  
  public static void main(String args[]){   
      DateTwo date = new DateTwo();   
      date.day = 28;  
      date.month = 07;  
      date.year = 2011;  
  }

When you complete the code, warning messages occur for each member field. You examine a warning message by putting your cursor over any of the red icons.  
  
While NetBeans will let you click **Save** without issuing a compiler error, it only saved the file. It does not compile the code or create the DateTwoTest.class file.

Graphic

*The error message reads 'day has private access in DayTwo'.*

Now you want to create a class with private attributes but enable them to be manipulated from another class. These are the steps for this procedure:

* create a new Java class called DateThree and add the same three private fields as the DateTwo class   
    
  **Code**  
  public class DateThree {  
      private int day, month, year;  
  }
* add a public get method for the day field, and   
  *The relevant code is:  
  public int getDay(){  
     return day;*   
    
  **Code**  
  public class DateThree {  
      private int day, month, year;  
    
      public int getDay(){  
          return day;  
      }  
    
  }
* add a public set method for the day field   
  *The code for this is:  
  public void setDay(int d){  
     this.day = day;*   
    
  **Code**  
  public class DateThree {  
      private int day, month, year;  
    
      public int getDay(){  
          return day;  
      }  
    
      public void setDay(int day){  
          this.day = day;  
      }

You then also add a similar get and set method for both the month and the year fields.

Graphic

*The relevant code is:  
public int getMonth(){  
   return month;  
}  
  
public void setMonth(int month){  
   this.month = month;  
}  
  
public int getYear(){  
   return year;  
}  
  
public void setYear(int year){  
   this.year = year;  
}*

Code

    public int getDay(){  
        return day;  
    }  
  
    public void setDay(int day){  
        this.day = day;  
    }  
  
    public int getMonth(){  
        return month;  
    }  
  
    public void setMonth(int month){  
        this.month = month;  
    }  
  
    public int getYear(){  
        return year;  
    }  
  
    public void setYear(int year){  
        this.year = year;  
   }

Note

*Most IDEs will automatically create the get and set methods for private fields in a class. This is part of a feature called "Refactoring". In NetBeans, you can take advantage of this feature by selecting a private field and right-clicking it. You then select****Refactor - Encapsulate****fields from the context menu to open the Encapsulate Fields window where you can opt to create getter and setter methods.*

Next you work with the DateThreeTest class, using these steps:

* in the main method, declare an object of type DateThree called dateObj, and create an instance of the DateThree class   
  *The code for this is:  
  DateThree dateobj = new DateThree();*   
    
  **Code**  
  public class DateThreeTest {  
    
      public static void main(String args[]){   
          DateThree dateobj = new DateThree();
* using the DateThree object reference, invoke the setMonth, setDay, and setYear methods of the DateThree object to set the three values of a date   
  *The code for this is:  
  dateObj.setMonth(6);   
  dateObj.setDay(12);   
  dateObj.setYear(2011);*   
    
  **Code**  
  public class DateThreeTest {  
    
      public static void main(String args[]){   
          DateThree dateobj = new DateThree();   
    
          dateObj.setMonth(6);   
          dateObj.setDay(12);   
          dateObj.setYear(2011);
* complete the main method by displaying the entire date in the format of your choice, and   
  *The code that does this is:  
  System.out.println(The date is :”+ dateObj.getMonth() + “/” + dateObj.getDay() + “/” + dateObj.getYear());*   
    
  **Code**  
  public class DateThreeTest {  
    
      public static void main(String args[]){   
          DateThree dateobj = new DateThree();   
    
          dateObj.setMonth(6);   
          dateObj.setDay(12);   
          dateObj.setYear(2011);  
    
          System.out.println(The date is :” +  
                              dateObj.getMonth() +  
          “/” + dateObj.getDay() + “/” +  
                              dateObj.getYear());
* save and compile your program, and run the DateThreeTest class to test it.   
    
  **Code**  
  run:  
  The date is: 6/12/11  
  BUILD SUCCESSFUL (total time: 0 seconds)

You now want to add a setDate method to the DateThree class you already created, which performs validation on the date part values that are passed into the method.  
  
To begin, in the DateThree class, you add a public setDate method that takes three arguments of type int. These values will be assigned to the day, month, and year fields, respectively.

Code

public void setDate(int d, int m, int y){  
}

You now need to perform the validation indicated before assigning the argument values to the fields by implementing these steps:

* In the setDate method, add the if/else statement to check the validity of the year argument.   
    
  **Code**  
  public void setDate(int d, int m, int y){  
      if(y > 1000 && y < 10000){  
          this.year = y;  
      } else {  
          System.out.println(y +   
              " is not a valid year.");  
          this.year = 0;  
      }
* Then you create a switch statement that evaluates the month argument. Months 1, 3, 5, 7, 8, 10, and 12 have 31 days. You check for these values first in the switch statement. If the month argument equals any of these cases, assign the month argument to the month field, include an if/else statement to test the value of the day argument. It should be between 1 and 31. If it is not, display an error message and set the day field to 0.   
    
  **Code**  
  switch (m) {   
  case 1:   
  case 3:   
  case 5:   
  case 7:   
  case 8:   
  case 10:   
  case 12:  
      this.month = m;  
      if (d > 0 && < 32){  
          this.day = d;  
          } else {  
              System.out.println(d +   
              “ is an invalid day for “  
   + m);  
              this.day = 0;  
          }  
          break;  
  //(switch statement continues in  next step)
* Finally, you use this logic to complete the switch statement. In the case block for the month of February, you also test for a leap year if the day argument is 29. The logic for the rest of the months is similar to what you wrote for months containing 31 days.   
    
  **Code**  
      this.month = m;  
      if(d > 0 && d < 29){  
          this.day = d;  
          //check if year is a leap year if d==29 and m==2  
      }else if (d == 29){  
          if(((y%4 == 0) && !(y%100 == 0)) || (y%400 == 0)){  
                          day = d;  
          }else{  
              System.out.println(  
              "Invalid day. Day cannot be 29 unless" +    
              "the year is a leap year.");  
              this.day = 0;  
          }  
     }

Next you need to add one more method called displayDate. In this method, you first check for values of zero in day, month, or year. If any of these has a 0 value, print an Invalid date message. Otherwise, display the date using a date format of your choice.

Code

public void displayDate(){  
    if(day == 0 || month == 0 || year == 0){  
        System.out.println(“Invalid date.”);  
    }  
    else {  
    System.out.println(“Date is: “ + month +”/”+  
    day +”/”+ year);  
    }  
}

Supplement

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

**Code Window**

Access the [DateThree Class](javascript:doWindow('../t24/misc/cw_jsef_a04_t24_001.html')) to view the full source code for the DateThree Class.

Now you open the DateThreeTest class and – using the setDate and displayDate methods – write code to perform the following tests:

* test with valid values for month, day and year   
    
  **Code**  
  dateObj.setDate(06, 12, 2011);  
  dateObj.displayDate();
* test with invalid value for month 14   
    
  **Code**  
  dateObj.setDate(05, 14, 2011);  
  dateObj.displayDate();
* test with invalid value for day 35, and   
    
  **Code**  
  dateObj.setDate(35, 11, 2011);  
  dateObj.displayDate();
* test with invalid year 200   
    
  **Code**  
  dateObj.setDate(05, 09, 200);  
  dateObj.displayDate();

Finally, you save and compile your program and run the DateThreeTest class. You can compare the output to your code in order to match up the messages with the particular test that was run.

Code

run:  
The date is: 30/12/2011  
14 is an invalid month.  
Invalid date.  
35 is an invalid day for month 5.  
Invalid date.  
200 is not a valid year.  
Invald date.  
BUILD SUCCESSFUL (total time: o seconds)

Question

Suppose you have created an Employee class that retrieves and saves employee data from a database. You use a private fetch method to populate the fields in the Employee class with data, including the employee name. And you use a public getEmpName method to allow other objects to retrieve an employee name. Say you change the code in the fetch method.   
  
What does the change affect?

**Options:**

1. Employee class methods that call getEmpName
2. External objects that call the fetch method
3. External objects that call the getEmpName method
4. The way fields in the Employee class's are populated

Answer

***Option 1:****Incorrect. The public getEmpName method is not affected by the change to the private fetch method code, so any calls to it are not affected.*

***Option 2:****Incorrect. External objects cannot call the fetch method because it is private.*

***Option 3:****Incorrect. The getEmpName method is part of the Employee class’s public interface, so the change to the fetch method does not affect external objects that interact with the Employee object using the getEmpName method.*

***Option 4:****Correct. The code for saving or retrieving employee data from a database in the fetch method is part of the Employee class’s private code, so it will be affected.*

**Correct answer(s):**

4. The way fields in the Employee class's are populated

**2. Summary**

In this topic, you've learned how to use encapsulation in Java.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#top)

Creating Constructors to Initialize Objects

Learning Objective

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

* *create constructors in Java*

**1. Initializing objects**

In this example, assuming you have setters for all the private fields of Shirt, you could instantiate and initialize a Shirt object by instantiating it and then setting the various fields through the setter methods.  
  
However, Java provides a more convenient way to instantiate and initialize an object by using a special method called a constructor.

Code

public class ShirtTest {  
    public static void main (String args[]) {  
    Shirt theShirt = new Shirt();  
    // Set values for the Shirt  
    theShirt.setColorCode('R');  
    theShirt.setDescription("Outdoors shirt");  
    theShirt.price(39.99);  
}

Constructors are method-like structures in a class, which

* have the same name as the class
* are usually used to initialize fields in an object, and
* can be overloaded

You can create a constructor using this syntax.

Syntax

*[modifiers]*class*ClassName {  
    [modifiers] ClassName([arguments]) {  
        code\_block  
    }  
}*

*[modifiers] represent several unique Java technology keywords that can modify the way constructors are accessed. Modifiers are optional, as indicated by the square brackets.*

*ClassName is the name of the class and the name of the constructor method. The name of the constructor must be the same as the ClassName in the class declaration.*

*[arguments] represent one or more optional arguments passed to the constructor.*

*The code\_block represents one or more optional lines of code for the constructor.*

This example has a constructor that accepts a char value to initialize the color code for this object. Because setColorCode() ensures that an invalid code cannot be set, the constructor can just call this method.

Code

public class Shirt {  
    public int shirtID = 0; // Default ID for the shirt  
    public String description = "-description required-"; // default  
    // The color codes are R=Red, B=Blue, G=Green, U=Unset  
    private char colorCode = 'U';  
    public double price = 0.0; // Default price all items  
    // This constructor takes one argument  
    public Shirt(char colorCode ) {  
        setColorCode(colorCode);  
}

Passing a valid color code to the Shirt constructor creates a new Shirt object, and calling display() results in this output.  
  
However, you will receive an error in NetBeans if you try to call the Shirt constructor with no arguments, the way you have been doing up until now.  
  
The reason for the problem is that if there is no explicit constructor in a class, Java assumes you want to be able to instantiate the class and gives you a default no-argument constructor. But if you create one constructor, Java assumes that you might want that to be the only constructor, and no longer provides a default no-argument implementation.

Code

Item ID: 0  
Item description:-description required-  
Color Code: G  
Item price: 0.0

This code shows three overloaded constructors:

Code

public class Shirt {  
    ... < declarations for field omitted > ...  
    // Default constructor  
    public Shirt() {  
        // You could add some default processing here  
    }  
    // This constructor takes one argument  
    public Shirt(char colorCode ) {  
        setColorCode(colorCode);  
    }  
    public Shirt(char colorCode, double price) {  
        this(colorCode);  
        setPrice(price);  
   }

* a default constructor with no parameters  
    
  *The code for this is:  
  public Shirt()*   
    
  **Code**  
  public class Shirt {  
      ... < declarations for field omitted > ...  
      // Default constructor  
      public Shirt() {  
          // You could add some default processing here  
      }  
      // This constructor takes one argument  
      public Shirt(char colorCode ) {  
          setColorCode(colorCode);  
      }  
      public Shirt(char colorCode, double price) {  
          this(colorCode);  
          setPrice(price);  
     }
* a constructor with one parameter - a char – and  
    
  *The code for this is:  
  public Shirt(char colorCode)*   
    
  **Code**  
  public class Shirt {  
      ... < declarations for field omitted > ...  
      // Default constructor  
      public Shirt() {  
          // You could add some default processing here  
      }  
      // This constructor takes one argument  
      public Shirt(char colorCode ) {  
          setColorCode(colorCode);  
      }  
      public Shirt(char colorCode, double price) {  
          this(colorCode);  
          setPrice(price);  
     }
* a constructor with two parameters - a char and a double   
  *The code for this is:  
  public Shirt(char colorCode, double price)*   
    
  **Code**  
  public class Shirt {  
      ... < declarations for field omitted > ...  
      // Default constructor  
      public Shirt() {  
          // You could add some default processing here  
      }  
      // This constructor takes one argument  
      public Shirt(char colorCode ) {  
          setColorCode(colorCode);  
      }  
      public Shirt(char colorCode, double price) {  
          this(colorCode);  
          setPrice(price);  
     }

The third constructor in this example sets both the colorCode field and the price field. However, the syntax where it sets the colorCode field is one you haven’t seen yet. It would be quite possible to set colorCode with a simple call to setColorCode() just as the previous constructor does, but this is another option.

Graphic

*The code referred to is:  
public Shirt(char colorCode, double price) {  
this(colorCode);*

Code

public class Shirt {  
    ... < declarations for field omitted > ...  
    // Default constructor  
    public Shirt() {  
        // You could add some default processing here  
    }  
    // This constructor takes one argument  
    public Shirt(char colorCode ) {  
        setColorCode(colorCode);  
    }  
    public Shirt(char colorCode, double price) {  
        this(colorCode);  
        setPrice(price);  
   }

Imagine that the second constructor has a lot of quite complex code associated with setting the colorCode field.   
  
If that were the case, you would also not want to duplicate the code in the third constructor, so you would chain the constructors by calling the second constructor in the first line of the third constructor, using this syntax.  
  
The keyword this is a reference to the current object.

Syntax

this(*argument*);

Question

Which part of the syntax for creating constructors always has to be included?

**Options:**

1. *ClassName*
2. *[modifiers]*
3. *[arguments]*
4. *code\_block*

Answer

***Option 1:****Correct. ClassName is the name of the class and the name of the constructor method. It is mandatory in the syntax.*

***Option 2:****Incorrect. The [modifiers] part of the syntax represents several unique Java technology keywords that can modify the way constructors are accessed. Modifiers are optional, as indicated by the square brackets.*

***Option 3:****Incorrect. The [arguments] part of the syntax does not have to be included. It represents one or more optional arguments passed to the constructor.*

***Option 4:****Incorrect. The code\_block part of the syntax is not mandatory. It represents one or more optional lines of code for the constructor.*

**Correct answer(s):**

1. *ClassName*

You now want to create a class and use constructors to initialize objects.  
  
To begin, you create a new Java class called Rectangle and add two private fields of type int – named width and height.

Code

public class Rectangle{  
    private int width, height;  
}

Next you add a constructor with no arguments by following these steps:

* declare a public no args constructor in the Rectangle class   
    
  **Code**  
  public Rectangle(){  
  }
* use System.out.println to display a message, and   
    
  **Code**  
  public Rectangle(){  
      System.out.println("Rectangle created: width = 25" +   
      " and height = 10");  
  }
* assign the width field to the value 25 and the height field to the value 10   
    
  **Code**  
  public Rectangle(){  
      System.out.println("Rectangle created: width = 25" +   
      " and height = 10");  
      width = 25;  
      height = 10;  
  }

Next you add a second constructor that accepts two int arguments - w and h.

Graphic

*The relevant code is :  
public Rectangle(int w, int h)*

Code

public Rectangle(int w, int h){  
}

Then, in the constructor, you begin adding an if/else statement to ensure that the values passed into the constructor are within the acceptable range of 1 through 29. If both arguments are valid, the argument is assigned to its respective member field.

Code

public Rectangle(int w, int h){  
    if(w > 0 && w < 30 && h > 0 &&     
       h < 30){  
    width = w;  
    height = h;  
    }  
}

Next you add code to print a message that indicates that a rectangle has been created with the designated values. You include the width and height values in the message, and if the argument values are not valid, you ensure that an error message is displayed.

Code

public Rectangle(int w, int h){  
    if(w > 0 && w < 30 && h > 0 && h < 30){  
        width = w;  
        height = h;  
        System.out.println("Rectangle created: width = "+width+   
        " and height = "+height);  
    }  
    else {  
        System.out.println("Invalid width and/or height. " +  
                    "Each must be positive and less than 30.");  
     }  
}

Next you create a getArea method that calculates and returns the area of the rectangle.

Code

public int getArea(){  
        return width \* height;  
}

Then you create a draw method that prints the rectangle shape, as determined by its width and height, in a series of rows containing asterisks. These are the steps:

* Create a nested for loop to draw the rectangle using asterisks.   
    
  **Code**  
  public void draw(){        
      System.out.println();   
      // create a new line  
      } // end of all rows  
  } // end of draw method
* Add code so that the outer for loop iterates through the rows of the rectangle. The number of rows corresponds to the value of the height field.   
  *The relevant code is:  
  for(int rowCounter=0;rowCounter<height;rowCounter++)*   
    
  **Code**  
  public void draw(){  
      for(int rowCounter=0;rowCounter<height;rowCounter++){  
         System.out.println();   
      // create a new line  
      } // end of all rows  
  } // end of draw method
* Add code so that the inner for loop iterates through the columns of each row. The number of columns corresponds to the value of the width field.   
  *The code for this is:  
  for(int colCounter=0;colCounter<width;colCounter++)*   
    
  **Code**  
  public void draw(){  
      for(int rowCounter=0;rowCounter<height;rowCounter++){  
          for(int colCounter=0;colCounter<width;colCounter++)  
          {   
              System.out.print(“\*”);  
          // end of each row  
      System.out.println();   
      // create a new line  
      } // end of all rows  
  } // end of draw method

You save and compile the program.  
  
Then, in the main method of the RetangleTest class, you declare and create two Rectangle objects, r1 and r2. You need to ensure the following:

Code

public static void main(String args[]){  
    // Rectangle with default values (no args)   
    Rectangle r1 = new Rectangle();  
    r1.draw();  
    //Rectangle from args constructor   
    Rectangle r2 = new Rectangle(15,5);   
    System.out.println(“Area of r2 is: “+r2.getArea());  
    r2.draw();  
}

* r1 is created with the no args constructor   
  *The relevant code is:  
  Rectangle r1 = new Rectangle();*   
    
  **Code**  
  public static void main(String args[]){  
      // Rectangle with default values (no args)   
      Rectangle r1 = new Rectangle();  
      r1.draw();  
      //Rectangle from args constructor   
      Rectangle r2 = new Rectangle(15,5);   
      System.out.println(“Area of r2 is: “+r2.getArea());  
      r2.draw();  
  }
* r1 is drawn immediately after it is created   
  *The relevant code is:  
  Rectangle r1 = new Rectangle();  
  r1.draw();*  
    
  **Code**  
  public static void main(String args[]){  
      // Rectangle with default values (no args)   
      Rectangle r1 = new Rectangle();  
      r1.draw();  
      //Rectangle from args constructor   
      Rectangle r2 = new Rectangle(15,5);   
      System.out.println(“Area of r2 is: “+r2.getArea());  
      r2.draw();  
  }
* r2 is created using the constructor with arguments, and   
  *The relevant code is:  
  Rectangle r2 = new Rectangle(15,5)*   
    
  **Code**  
  public static void main(String args[]){  
      // Rectangle with default values (no args)   
      Rectangle r1 = new Rectangle();  
      r1.draw();  
      //Rectangle from args constructor   
      Rectangle r2 = new Rectangle(15,5);   
      System.out.println(“Area of r2 is: “+r2.getArea());  
      r2.draw();  
  }
* r2 is drawn and the area is printed   
  *The relevant code is:  
  System.out.println(“Area of r2 is: “+r2.getArea());  
  r2.draw();*   
    
  **Code**  
  public static void main(String args[]){  
      // Rectangle with default values (no args)   
      Rectangle r1 = new Rectangle();  
      r1.draw();  
      //Rectangle from args constructor   
      Rectangle r2 = new Rectangle(15,5);   
      System.out.println(“Area of r2 is: “+r2.getArea());  
      r2.draw();  
  }

You save and compile your program and run the RectangleTest class to test it.

Code

run:  
Default rectangle created: width = 25, height = 10  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
Default rectangle created: width = 15, height = 5  
Area of r2 is: 75  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
BUILD SUCCESSFUL (total time: 0 seconds)

Question

What is the default constructor for this class?

**Code**  
public class Square {  
     String name = “box”;  
}

**Options:**

1. public Square(String name)
2. public Square()
3. class()
4. String()

Answer

***Option 1:****Incorrect. The code, public Square(String name), is not an example of a default constructor. The default constructor for this class is public Square().*

***Option 2:****Correct. public Square() is the default constructor for this class.*

***Option 3:****Incorrect. The default constructor is not class(), it is public Square().*

***Option 4:****Incorrect. The default constructor is public Square().*

**Correct answer(s):**

2. public Square()

**2. Summary**

In this topic, you've learned how to create constructors in Java.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#top)

Using Methods, Overloading, and Encapsulation in Java

Learning Objectives

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

* *create and call methods with arguments and return values*
* *implement encapsulation*
* *use constructors to initialize objects*

**1. Exercise overview**

In this exercise you will create methods with arguments and experiment with field access and encapsulation, as well as create and use overloaded constructors.

This involves the following tasks:

* using methods with arguments and return values
* implementing encapsulation, and
* using constructors to initialize objects

**2. Using arguments and return values**

You want to work with methods that have arguments and return values so that you can create code that will calculate the total order of many books.

Question

You have created a class named BookOrder that allows you to place an order for more than one Book. You want to add a method to this class to calculate the total price for the amount of Books that have been ordered.   
  
Which code segment correctly completes the method declaration so you can pass an argument representing a Book object into the method?

**Code**  
    public double addBookINSERT THE MISSING CODE {  
        totalPrice = totalPrice + b.price;  
        …  
    }

**Options:**

1. (Book b)
2. (Book b;)
3. (totalPrice)
4. (b)

Answer

***Option 1:****Correct. This segment correctly indicates that this method receives an argument of type Book naming it b.*

***Option 2:****Incorrect. This segment uses a semicolon, which is used to indicate the end of a statement.*

***Option 3:****Incorrect. This segment refers to the totalPrice variable, which is used to hold the total cost of the books purchased.*

***Option 4:****Incorrect. This segment indicates only the name of the parameter and not the type.*

**Correct answer(s):**

1. (Book b)

Question

You want to complete the method so that it calculates a value for the total price for the amount of books that can be assigned to a variable when called.   
  
Which code segment correctly completes the method?

**Code**  
    public double addBook(Book b) {  
        totalPrice = totalPrice + b.price;  
        INSERT THE MISSING CODE  
    }

**Options:**

1. return totalPrice;
2. System.out.println(totalPrice);
3. return b;
4. return b.price;

Answer

***Option 1:****Correct. This statement correctly returns the totalPrice variable.*

***Option 2:****Incorrect. This statement would print out the value but not return it. Since a return type is indicated in the method declaration, you must return a value of this type.*

***Option 3:****Incorrect. This statement would return a reference to the Book object that was passed to this method.*

***Option 4:****Incorrect. This statement would return the value for an individual book and not the running total.*

**Correct answer(s):**

1. return totalPrice;

Question

You now want to invoke the addBook(Book b) method from your test class.   
  
Which statement correctly completes the code sample and calls the method?

**Code**  
BookOrder order = new BookOrder();  
Book myBook = new Book();  
double totalCost = 0.0;  
myBook.price = 29.99;  
INSERT THE MISSING CODE

**Options:**

1. totalCost = order.addBook(myBook);
2. totalCost = order.addBook();
3. totalCost = myBook.addBook(myBook);
4. order.addBook(myBook) = totalCost;

Answer

***Option 1:****Correct. This statement correctly calls the method and assigns its return value to the totalCost variable.*

***Option 2:****Incorrect. Recall that the addBook method requires you to pass a Book object reference to it.*

***Option 3:****Incorrect. You need to use a BookOrder object, in this case order, to invoke the method.*

***Option 4:****Incorrect. This statement assigns the return value incorrectly.*

**Correct answer(s):**

1. totalCost = order.addBook(myBook);

**3. Implementing encapsulation**

You now want to use encapsulation in your code.

Question

You decide to implement encapsulation in your Book class. You have already made the field variables author, title, and price private.   
  
Which code segment is the correct way to implement a setter method for the String title field?

**Options:**

1. public void setTitle(title) {  
      this.title = title;  
    }
2. public void setTitle(String title) {  
      title = this.title;  
    }
3. public String setTitle(String title) {  
      this.title = title;  
    }
4. public void setTitle(String title) {  
      this.title = title;  
    }

Answer

***Option 1:****Incorrect. This method fails to properly indicate the type of the parameter required.*

***Option 2:****Incorrect. This method assigns the value of the field to the parameter. It should be the other way around.*

***Option 3:****Incorrect. This method has a return type. Typically a setter method sets a value and does not return one.*

***Option 4:****Correct. This method correctly accepts a parameter and sets the private field to that value.*

**Correct answer(s):**

4. public void setTitle(String title) {  
   this.title = title;  
 }

Question

Assuming you have a Book object named myBook in your test class, which code segment correctly completes the sample to use the get method for the price field?

**Code**  
myBook.setPrice(25.99);   
System.out.println("The book price is: " + myBook.INSERT THE MISSING CODE );

**Options:**

1. getPrice()
2. getPrice(25.99)
3. setPrice()

Answer

***Option 1:****Correct. This segment correctly calls the getPrice method.*

***Option 2:****Incorrect. Since this method is supposed to retrieve a value, you would not need to pass it a parameter.*

***Option 3:****Incorrect. You use the getPrice() method to retrieve the value.*

**Correct answer(s):**

1. getPrice()

**4. Using constructors to initialize objects**

You now want to use constructors to set the author, title, and price of the books.

Question

You decide to add a no args constructor to the Book class to set the basic values for a Book object.   
  
Which code segment correctly completes the sample?

**Code**  
INSERT THE MISSING CODE{  
    author = “John Doe”;  
    title = "Book One"  
    price = 9.99;  
    }

**Options:**

1. public Book()
2. public Book{}
3. public Book(String author, String title, double price)
4. private Book()

Answer

***Option 1:****Correct. This segment correctly completes the constructor method.*

***Option 2:****Incorrect. This segment incorrectly uses curly braces.*

***Option 3:****Incorrect. You require a no args constructor to complete the sample – not one that accepts parameters for the field values.*

***Option 4:****Incorrect. You need to make sure that constructors are declared as public so they can be accessed outside of the class.*

**Correct answer(s):**

1. public Book()

Question

You have added a constructor to the Book class that allows you to set author, title, and price.   
  
Which statement correctly calls this constructor?

**Options:**

1. Book myBook = Book("S. Queen", "The Golden Spire", 14.99);
2. Book myBook = new Book("S. Queen"; "The Golden Spire"; 14.99);
3. Book myBook = new Book("S. Queen", "The Golden Spire", 14.99);
4. Book myBook = new Book();

Answer

***Option 1:****Incorrect. This statement neglects to use the new keyword.*

***Option 2:****Incorrect. This statement uses semi-colons to separate the parameters. These should be commas.*

***Option 3:****Correct. This statement correctly calls the constructor to create a new Book object with the default values.*

***Option 4:****Incorrect. This statement calls the no args constructor.*

**Correct answer(s):**

3. Book myBook = new Book("S. Queen", "The Golden Spire", 14.99);

Methods with arguments and return values have been used, encapsulation has been implemented, and constructors have been used.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#top)

Using Advanced Object-Oriented Concepts

Learning Objective

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

* *create and use superclasses and subclasses*

**1. Introduction to inheritance**

Inheritance results in a hierarchy of Java technology classes similar to the taxonomies found in biology. For example, Blue Whale is a subclass of Whale.  
  
In a hierarchy for whales, warm blooded is an attribute of the Mammal superclass. The phrase "breathes air" represents some operation that is also a part of the Mammal superclass. Flukes and flippers are attributes that are specific to Whale, which is a subclass of mammal.

Graphic

*A class hierarchy diagram is displayed with the superclass Mammal at the top. Attributes of this class include breathes air and warm blooded. Extending from the superclass Mammal, is the subclass Whale. This subclass includes the attributes flukes and flippers. Finally, two further subclasses, Blue Whale and Humpback Whale, extend from the Whale subclass.*

Suppose Duke's Choice defines a set of behaviors for the Shirt class and for a new class, the Trousers class.   
  
The classes are fully encapsulated so that all field values are only accessible through setter and getter methods. For example, a getter method for the Shirt class is getID(). Both classes use many of the same methods, which may result in code duplication. This makes maintenance and further expansion more difficult and error prone.

Graphic

*The classes Shirt and Trousers are displayed. Both include the methods getID(), display(), getPrice(), getSize(), getColor(), and getFit(). Trousers has the additional method of getGender().  
  
The diagram shows only the getter methods for accessing field values. Equivalent setter methods are also included in each of the classes.*

Supplement

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

**Learning aid**

Access the learning aid, [Common Behaviors](javascript:doWindow('./la_jl_jsef_a04_it_enus_T1501_frame.html')), to access a set of behaviors for the Shirt class and for the Trousers class.

If Duke's Choice decides to add a third item, socks, as well as trousers and shirts, you may find even greater code duplication.

Graphic

*The Socks class is added to the existing Shirt and Trousers classes. It has the methods getID(), display(), getPrice(), getSize(), getColor().*

You can eliminate the need to duplicate this information in every class by implementing inheritance. Inheritance enables programmers to put common members, such as fields and methods, in one class - the superclass. Other classes, or subclasses, inherit these common members from this new class.  
  
An object instantiated from a subclass behaves as if the fields and methods of the subclass were in the object. For example, the Trousers class can be instantiated and have the display() method called even though the Trousers class doesn't contain a display() method; it is inherited from the Clothing class.

Graphic

*The Clothing superclass contains the methods common to the Shirt, Trousers and Socks classes: getID(), display(), getPrice(), getSize and getColor. The subclass for Shirt now only has the method getFit(), while the Trousers subclass has getFit() and getGender(). There are no additional methods for Socks.*

Methods that exist in the superclass can be implemented in

* the superclass only, or
* the subclass and superclass

Subclasses may implement methods that already have implementations in the superclass. In this case, the method implementations in the subclass are said to override the method implementation from the superclass. The method declared in the subclass will then be used at runtime. For example, although the colorCode field and its accessor methods is in the superclass, the color choices may be different in each subclass. You may need to override the get and set methods for this field in the individual subclasses.

You should use inheritance only when it is completely valid or unavoidable.  
  
To check the appropriateness of a superclass and subclass relationship, you can use the "is a" phrase. For example, the phrase "a Shirt is a piece of Clothing" expresses a valid inheritance link. However, the phrase "a Hat is a Sock" expresses an invalid inheritance link.

**2. Declaring a subclass**

You want to create a Clothing superclass. First you define the fields and constructor for the class.

Graphic

*The relevant code is  
  
// Fields  
  
and  
  
// Constructor*

Code

public class **Clothing** {  
    // Fields  
    private int itemID = 0; // Default ID for all clothing items  
    private String description = "-description required-"; // default  
    private char colorCode = 'U'; //'U' is Unset  
    private double price = 0.0; // Default price for all items  
  
    // Constructor  
    public Clothing(int itemID, String description, char colorCode, double price) {  
        this.itemID = itemID;  
        this.description = description;  
        this.colorCode = colorCode;  
        this.price = price; }

You code the getters for the superclass.

Code

public void display() {  
    System.out.println("Item ID: " + getItemID());  
    System.out.println("Item description: " + description);  
    System.out.println("Item price: " + getPrice());  
    System.out.println("Color code: " + getColorCode());  
} // end of display method  
public String getDescription(){  
    return description;  
}  
public double getPrice() {  
    return price;  
}  
public int getItemID() {  
    return itemID;  
}  
public char getColorCode() {  
    return colorCode;  
}

You code the setter methods for the superclass.

Code

public void setItemID(int itemID) {  
    this.itemID = itemID;  
}  
public void setDescription(String description) {  
    this.description = description;  
}  
public void setColorCode(char colorCode) {  
    this.colorCode = colorCode;  
}  
public void setPrice(double price) {  
    this.price = price;  
}

When declaring a subclass, you use the extends keyword to indicate inheritance. This way you can ensure that Shirt inherits members of Clothing.  
  
The Shirt subclass declares attributes and methods that are unique to this class. Attributes and methods that are common with the Clothing class are inherited and do not need to be declared.

Graphic

*The relevant code and syntax is  
  
extends*

Code

public class Shirt extends Clothing {  
    private char fit = 'U'; //'U' is Unset, other codes 'S', 'M', or 'L'  
  
    public Shirt(int itemID, String description, char colorCode, double price, char fit) {  
        super(itemID, description, colorCode, price);  
        this.fit = fit;  
    }  
  
    public char getFit() {  
        return fit;  
    }  
  
    public void setFit(char fit) {  
        this.fit = fit;  
   }

Syntax

[*class\_modifier*] class *class\_identifier* extends *superclass\_identifier*

When working with classes, two useful keywords are

Code

public class Shirt extends Clothing {  
    private char fit = 'U'; //'U' is Unset, other codes 'S', 'M', or 'L'  
  
    public Shirt(int itemID, String description, char colorCode, double price, char fit) {  
        super(itemID, description, colorCode, price);  
        this.fit = fit;  
    }  
  
    public char getFit() {  
        return fit;  
    }  
  
    public void setFit(char fit) {  
        this.fit = fit;  
   }

**super, and**

You use super to refer to the methods and attributes of the superclass. Even if a method of the superclass has been overridden in the subclass, using the super keyword allows you to invoke the method of the superclass.  
  
In the Shirt subclass, it's used to invoke the constructor on the superclass. By using this technique, the constructor on the superclass can be invoked to set all the common attributes of the object being constructed. Then additional attributes can be set in the statements that follow.

**this**

In contrast to the super keyword, this is a reference to the object of this class. It's not necessary to use it, but it's common to do so in constructors to help make the code more readable.  
  
The only additional attribute that Shirt has is the fit attribute, and it's set after the invocation of the superclass constructor. You use this to indicate the attribute of the object rather than a variable passed to the class.  It is not necessary to use it in this example, but it is common to do so in constructors to help make the code more readable

The display() method overrides the method of the superclass and is more specific to the Shirt class.  
  
Likewise, the method setColorCode() overrides the setColorCode() method of the superclass to check that a valid value is being used for this class. This is one of the advantages of encapsulating fields.

Code

    //This method overrides display in the Clothing superclass  
    public void display() {  
        System.out.println("Shirt ID: " + getItemID());  
        System.out.println("Shirt description: " + description);  
        System.out.println("Shirt price: " + getPrice());  
        System.out.println("Color code: " + getColorCode());  
        System.out.println("Fit: " + getFit());  
    } // end of display method  
  
    // This method overrides the methods in the superclass  
    public void setColorCode(char colorCode) {  
        ... include code here to check that correct codes used ...  
        this.colorCode = colorCode;  
    }  
}

Question

Identify the best examples of an inheritance hierarchy.

**Options:**

1. Animal extends Dog extends Cat
2. Cappuccino extends Coffee extends Beverage
3. Director extends Manager extends Employee
4. Vehicle extends Car

Answer

***Option 1:****Incorrect. Dogs and cats are types of animals. This hierarchy says that a dog is a type of cat and that an animal is a type of dog, which is not the case.*

***Option 2:****Correct. The "is a" test reveals that a cappuccino is a coffee, which is a beverage.*

***Option 3:****Correct. The "is a" test reveals that a director is a manager, who is an employee.*

***Option 4:****Incorrect. The "is a" test reveals that a vehicle is a car. This does not make sense because not all vehicles are cars. For example, a motorcycle is a type of vehicle.*

**Correct answer(s):**

2. Cappuccino extends Coffee extends Beverage  
3. Director extends Manager extends Employee

**3. Abstract classes**

Sometimes a superclass makes sense as an object, and sometimes it doesn't. For example, Duke's Choice carries shirts, socks, and trousers, but it does not have an individual item called clothing.   
  
Also, in the application, the superclass Clothing may declare some methods that may be required in each subclass, and thus can be in the superclass, but cannot really be implemented in the superclass.

Graphic

*The Clothing superclass is represented by a question mark. It contains the methods common to the Shirt, Trousers and Socks classes: getID(), display(), getPrice(), getSize and getColor. The subclass for Shirt now only has the method getFit(), while the Trousers subclass has getFit() and getGender(). There are no additional methods for Socks.*

Abstraction refers to creating classes that are general and may contain methods without particular implementation or method body code.   
  
An example of an abstract class is the Clothing class. Clothing is an abstract concept. You usually don't go to a store and say, "I want to buy a clothing item" because that can refer to anything. However, all clothing items have some similar characteristics in the context of an order entry system, such as an ID or a method to display information about the item.  
  
Classes that are generic and cannot be fully defined, such as an Item class, are referred to as abstract classes. The abstract keyword ensures that the class cannot be instantiated.

Graphic

*The relevant code is  
  
abstract class Clothing*

Code

public abstract class Clothing {  
    // Fields  
    private int itemID = 0; // Default ID for all clothing items  
    private String description = "-description required-"; // default  
    private char colorCode = 'U'; //'U' is Unset  
    private double price = 0.0; // Default price for all items  
  
    // Constructor  
    public Clothing(int itemID, String description, char colorCode,  
        double price) {  
        this.itemID = itemID;  
        this.description = description;  
        this.colorCode = colorCode;  
        this.price = price;  
   }

Classes that extend an abstract class must implement all the empty methods of the abstract class with code specific to the subclass. You should spend time on your analysis and design to make sure your solution has enough abstraction to ensure flexibility.  
  
For example, the get and set methods for the colorCode field are abstract to ensure they are implemented appropriately in each subclass. The Shirt subclass will compile correctly as a subclass of this abstract class, as it already has implementations of these two methods.   
  
But if the implementations of getColorCode() and setColorCode() are removed from the Shirt subclass, the compile will fail, because abstract methods in the superclass must be implemented in the subclass.

Graphic

*The relevant code is  
  
abstract*

Code

    public abstract char getColorCode() ;  
  
    public abstract void setColorCode(char colorCode);  
  
    ... other methods not listed ...  
}

You can have superclass and subclass relationships of more than two levels.   
  
For example, the base superclass Employee currently has two subclasses. An advantage of inheritance is that it is easy at any time to create a new class that extends Employee. This new class then inherits all the functionality that Employee has.  
  
One of the Employee subclasses is SkilledEmployee, which has three subclasses of its own – Editor, GraphicIllustrator, and TechnicalWriter. None of these classes are abstract because there is such a thing as an employee and some processes in an application using these classes may work with the Employee class.

Graphic

*A diagram displays the superclass Employee, which contains the methods getName() and getEmpId(). There are two subclasses extending from the superclass: SkilledEmployee, which has the method getSkills(), and Manager, which has the method getEmployees(). Extending from the SkilledEmployee subclass, are the subclasses Editor, GraphicIllustrator, and TechnicalWriter. The Editor subclass has a method of getEditType(), and GraphicIllustrator, has the method of getToolsUsed().*

Question

You need to design an employee database for your company. You decide to create classes for Employee, Assistant, and Manager.   
  
Select the most effective inheritance hierarchy for the three classes.

**Options:**

1. A Manager class and an Assistant subclass
2. An Employee superclass and Manager and Assistant subclasses
3. Assistant and Manager classes that override an Employee class
4. Separate Assistant and Manager classes

Answer

***Option 1:****Incorrect. This option fails the "is a" test. An assistant is not a manager, although both assistants and managers are employees.*

***Option 2:****Correct. Both assistants and managers share the characteristics of employees. In this case, creating a generic Employee class containing the common aspects of an employee enables object orientation.*

***Option 3:****Incorrect. A class does not override another class. Methods of a subclass can override their counterparts in a superclass.*

***Option 4:****Incorrect. Creating separate classes for Assistant and Manager will lead to duplicate coding because assistants and employees share characteristics. Code and data replication is inefficient and such repetition can be minimized by deploying an appropriate hierarchical model.*

**Correct answer(s):**

2. An Employee superclass and Manager and Assistant subclasses

**4. Using superclasses and subclasses**

You want to design and then create a class hierarchy that will form the basis for an Employee Tracking System of the Marketing department in the Duke's Choice Company. First you'll create a simple design model for the class hierarchy. Then you'll create the actual classes and test them.

The Marketing department of Duke's Choice has employees in several positions, including Technical Writers, Graphic Illustrators, Managers, and Editors. Marketing wants you to create a program for tracking their information.  
  
You'll need to include each worker’s name, job title, employee ID, and level, such as 1, 2, or 3, and there must be a means by which to display all the information for a given employee type.  
  
Managers must have a list of employees that they manage, while Technical Writers, Graphic Illustrators, and Editors must have a list of skills that they possess. You must also include a value for Editors indicating whether they prefer to do electronic editing or paper-based editing.

You need to create a class hierarchy of superclass and subclass relationships for the employees of the Marketing department. You can model the class hierarchy using class diagrams drawn on paper, or use the UMLet tool. Follow these steps:

Graphic

*The diagram displays the superclass Employee.*

**use the "is a" phrase**

Ask yourself if all or many of the job types have some of the same attributes, or fields, and operations, or methods. For example, all of the different job types can also be called Employees in the general sense. They share certain fields and operations. Therefore, a Manager *is an* Employee. An Editor also *is an* Employee.  
*The Manager and Editor subclasses are added to the superclass Employee.*

**consider an interim superclass**

If you find that certain employee types share common fields or operations that are not shared by other employee types, for instance a list of skills, yet they are all employees, consider creating a common superclass for these employees – inherited from the top-level superclass, Employee.  
*The Skilled Employee subclass is added to the Employee superclass. It has the subclasses GraphicIllustrator, TechnicalWriter, and Editor added to it.*

**determine common fields, and**

Common information fields shared by the different employees can be displayed from the top-level superclass. In the subclass, simply add the fields unique to this particular employee type. You override the method from the superclass and call it from within the subclass method of the same name. This adds code to display additional fields.  
*The Employee superclass has the fields: name, jobTitle, employeeID, and level. The Manager subclass has the field: Employee [] employeeList, and the SkilledEmployee subclass has the field String [] skillList. No new fields are added to GraphicIllustrator or TechnicalWriter, but Editor has a prefersPaperEditing field.*

**use encapsulation**

You demonstrate encapsulation for each of the classes in your design by including get and set methods for each private field, according to the type of access required.  
*The Employee superclass has a number of methods added to it, including getName(), setName(), getJobTitle(), setJobTitle(), and getEmployeeID(). The Manager subclass has methods including setEmployee(), and getEmployees() added to it. There are no new methods added to the subclasses GraphicIllustrator or TechnicalWriter, but the Editor subclass has the setPrefersPaperEditing(), getEditingPreference(), and displayInformation() : override, methods added.*

Before you begin creating the classes, you need to change a property of the NetBeans IDE. The Add @Override Annotation property of the editor is useful when you are creating javadocs for your application. This property is applied when you override a method in the superclass. Because you are not creating javadocs, you want to turn off this property. To make this change, you perform the following steps:

Graphic

*The NetBeans IDE 7.0.1 toolbar has menu options for File, Edit, View, Navigate, Source, Refactor, Run, Debug, Profile, Team, Tools, Window, and Help. The Projects window has three tabs: Projects, Files, and Services. The window is empty as no project is currently open.*

* Select **Tools** - **Options** from the main menu. In the Options window, click the **Editor** toolbar button and then click the **Hints** tab.   
  *The toolbar contains these buttons: General, Editor, Fonts and Colors, Keymap, and Miscellaneous. Tabs accessible from the Editor button include General, Formatting, Code Completion, Code Templates, Hints, Mark Occurrences, and Macros. The General tab displays options to customize Code Folding and Camel Case Behavior.*
* Change the language to Java - the hints in the left column will change accordingly.   
  *On the Hints tab, the Language drop-down list is set to Java. There are several expandable node options, which can be selected or cleared with a checkbox, including APIs, Assignment Issues, Bitwise Operations, and Braces.*
* Expand the **JDK 1.5 and later** node and clear **Add @Override Annotation**.   
  *Other options under the JDK 1.5 and later node include Can Use Diamond, Convert to try-with-resources, and Don't use Annotation as super interface.*
* Click **OK** to save the change and close the Options window.   
  *The Options window includes buttons for Export, Import, OK, Cancel, and Help.*

To create the Employee class, you

Code

public class Employee {  
    private int employeeID;   
    private String name;   
    private String jobTitle;  
    private int level;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    .....  
  
    public int getEmployeeID() {  
        return employeeID;  
    }  
  
     
    private void setEmployeeID(int employeeID) {  
        this.employeeID = employeeID;  
    }  
   ......  
}

**declare the fields private, and**

All of the fields should be private. You follow the same naming pattern that you have been using, such as camelCase.  
*The relevant code is  
  
private int employeeID;   
private String name;   
private String jobTitle;  
private int level;*

**encapsulate the fields**

You use the Refactor feature of NetBeans to encapsulate these fields, that is, to create get methods for each field and set methods for each field. And you change the access modifier for the setEmployeeID method to private.  
  
Employee IDs will be calculated to ensure uniqueness and you must restrict public write access to this field so that the IDs will always be unique. ID values are only set by the calculateEmployeeID method.  
*The relevant code is  
  
getName()  
getJobTitle()  
getEmployeeID()  
  
and  
  
getLevel()*

You add another field called employeeIDCounter. Make it a protected static int field and initialize it to zero.  
  
A static field is a "class" field. There is only one value for this field that is shared by all instances of this class. The static field will be used here to store an integer value that is incremented from within the calculateEmployeeID method to generate the next ID value. The employeeIDCounter is accessed and incremented by all instances of Employee and its subclasses, thus ensuring that no duplicate employee IDs are generated.  
  
In a real business application, this technique would not be robust enough to guarantee unique IDs. Instead, a database would probably generate the IDs. However, this technique suffices for this simple application.

Graphic

*The relevant code is  
  
protected static int employeeIDCounter = 0;*

Code

public class Employee {  
    protected static int employeeIDCounter = 0;  
    private int employeeID;   
    private String name;   
    private String jobTitle;   
    private int level;

Finally, you add the methods

Code

public class Employee {  
    protected static int employeeIDCounter = 0;  
    private int employeeID;   
    private String name;   
    private String jobTitle;   
    private int level;  
  
  
    public void calculateEmployeeID() {  
        employeeIDCounter++; // inc so employeeID's unique  
        setEmployeeID(employeeIDCounter);  
    }  
  
  
    public void displayInformation() {  
        System.out.println("Name: " + getName());   
        System.out.println("Job Title:" + getJobTitle());  
        System.out.println("Employee ID: " + getEmployeeID());  
        System.out.println("Level: " + getLevel());  
    }  
    // The set and get methods are not shown here   
}

**calculateEmployeeID, and**

You create the calculateEmployeeID method. It takes no arguments and does not return a value. In the body of this method, increment the employeeIDCounter and then set the new value in the employeeID field. You use the set method of the field.

**displayInformation**

You create the displayInformation method. It takes no arguments and does not return a value. In this method, print out the value of each field of the class with a suitable label. You then click **Save** to compile the class.

You create the Manager class using this code.

Code

public class Manager extends Employee{  
    private ArrayList employeeList = new ArrayList();  
      
    public void setEmployee(Employee emp){  
        employeeList.add(emp);  
    }  
      
    public ArrayList getEmployees(){  
        return employeeList;  
   }

**public class Manager extends Employee{**

*After creating the new Java Class file, you add a phrase to the class declaration to indicate that it is a subclass of Employee.*

**private ArrayList employeeList = new ArrayList();**

*You declare and instantiate the employeeList field as a private ArrayList instead of the array of type Employee. This will be simpler to work with than an array. Be sure to add the necessary import statement to import the java.util.ArrayList class. You can do this by clicking the****error****icon in the left margin and letting NetBeans add the import statement for you.*

**setEmployee(Employee emp)**

*You add a public setEmployee method to add a single employee to the employeeList. The method takes an argument of type Employee. You use the add method of the ArrayList to add the Employee object to the employeeList object.*

**getEmployees()**

*You then add a public getEmployees method that simply returns the employeeList field.*

You need to add a displayInformation method to override the method in the Employee class. In this method, you invoke the displayInformation method in the superclass, and then display additional information specific to the Manager class.  
  
You must declare the method with the exact same signature as in the superclass method by having it return void and accept no arguments.

Code

public class Manager extends Employee{  
    private ArrayList employeeList = new ArrayList();  
      
    public void setEmployee(Employee emp){  
        employeeList.add(emp);  
    }  
      
    public ArrayList getEmployees(){  
        return employeeList;  
    }  
      
    public void displayInformation(){  
        Employee emp;

Note

*NetBeans will display a green circle icon in the margin as you have finished typing the method declaration. This indicates that this method overrides the superclass method. When you click the green circle, it opens the Employee class in the editor to show you the ancestor method.*

In the method block, you need to invoke the superclass method using the super keyword as a reference to the Employee class.  
  
You have the method display the message "Manager has the following employees".

Code

public class Manager extends Employee{  
    private ArrayList employeeList = new ArrayList();  
      
    public void setEmployee(Employee emp){  
        employeeList.add(emp);  
    }  
      
    public ArrayList getEmployees(){  
        return employeeList;  
    }  
      
    public void displayInformation(){  
       Employee emp;  
       super.displayInformation();  
       System.out.println(  
           "Manager has the following employees: ");

Next you iterate through the employeeList using an enhanced for loop. The employeeList is an ArrayList that holds objects. The compiler does not know that these objects happen to be Employee objects. Therefore, in order to get the name field from each object to display it, you have to cast the Object to an Employee. An employee is an object. You declare a local variable at the top of this method of type Employee to hold the cast value.

Graphic

*The relevant code is  
  
for(Object obj : employeeList)*

Code

public class Manager extends Employee{  
    private ArrayList employeeList = new ArrayList();  
      
    public void setEmployee(Employee emp){  
        employeeList.add(emp);  
    }  
      
    public ArrayList getEmployees(){  
        return employeeList;  
    }  
      
    public void displayInformation(){  
       Employee emp;  
       super.displayInformation();  
       System.out.println(  
           "Manager has the following employees: ");  
       for (Object obj : employeeList){  
           emp = (Employee)obj;  
           System.out.println("\t"+ emp.getName());  
       }  
   }  
}

You create the SkilledEmployee class, which should also extend Employee. To do this, you

Code

public class SkilledEmployee extends Employee{  
    private ArrayList skillList = new ArrayList();  
      
    public void setSkill(String skill){  
        skillList.add(skill);  
    }  
      
    public ArrayList getSkills(){  
        return skillList;  
   }

**use an ArrayList instead of a String array and import the ArrayList class**

You use an ArrayList instead of a String array when you declare the skillList field. You instantiate the field to an empty ArrayList. You must also add the necessary import statement for the ArrayList class.

**add a public setSkill method, and**

You add a public setSkill method to add a single skill to the skillList. The method takes an argument of type String. You use the add method of the ArrayList to add the String to the skillList object.

**add a public getSkills method**

You add a public getSkills method that returns the skillList.

You override the displayInformation method as you did in the Manager class. After invoking the superclass method, you display the message "Employee has the following skills". Then iterate through the skillList using an enhanced for loop, displaying each skill, indented by a tab as you did in the Manager class.  
  
The skillList object contains String objects. In this case, you can directly print the Object reference from the ArrayList without casting it to a String. The reason for this is that every Object has a toString method. The println method will invoke this for you, resulting in the display of the String value, such as the skill.  
  
You click **Save** to compile the program.

Code

public class SkilledEmployee extends Employee{  
    private ArrayList skillList = new ArrayList();  
      
    public void setSkill(String skill){  
        skillList.add(skill);  
    }  
      
    public ArrayList getSkills(){  
        return skillList;  
    }  
      
    public void displayInformation(){  
        super.displayInformation();  
        System.out.println("Employee has the following skills: ");  
        for(Object obj : skillList){  
            System.out.println("\t"+ obj);  
        }  
   }  
}

You create the Editor class as a subclass of SkilledEmployee. These are the steps:

Code

public class Editor extends SkilledEmployee {

* Declare the prefersPaperEditing field as a private boolean. It will be initialized to a default value of false.   
    
  **Code**  
      private boolean prefersPaperEditing;
* Add a setPrefersPaperEditing method that takes a boolean argument and returns void. Assign the argument to the private field.   
  *The relevant code is  
    
  public void setPrefersPaperEditing(boolean pref)*   
    
  **Code**  
          public void setPrefersPaperEditing(boolean pref){  
          prefersPaperEditing = pref;  
      }
* Add a getEditingPreference method that returns a String value. Use an if/else construct to check the value of prefersPaperEditing and set the return value to either "Paper" or "Electronic".   
  *The relevant code is  
    
  public String getEditingPreference()*   
    
  **Code**  
          public String getEditingPreference(){  
          if(prefersPaperEditing){  
              return "Paper";  
          }  
          else {  
              return "Electronic";  
          }  
     }
* Override the displayInformation method as you did in the Manager class, invoking the superclass method first and then displaying the return value of this.getEditingPreference() with a suitable label. Then click **Save** to compile the program.   
  *The relevant code is  
    
  super.displayInformation();  
    
  and  
    
  getEditingPreference());*   
    
  **Code**  
      public void displayInformation(){  
          super.displayInformation();  
          System.out.println("Editing preference: " + getEditingPreference());  
      }  
  }

You can now create the remaining two classes, GraphicIllustrator and TechnicalWriter. Both of these classes extend the SkilledEmployee class. It isn't necessary to add any additional fields or methods, nor is it necessary to override the displayInformation method.  
  
You then save and compile the program.

Code

public class GraphicIllustrator extends SkilledEmployee{  
    // nothing to add  
}  
  
public class TechnicalWriter extends SkilledEmployee{  
    // nothing to add  
}

You then code and compile the EmployeeTest class. If there are any error indicators, check to make sure that you have spelled all of your method names the same way they are spelled in this class.

Code

public class EmployeeTest {  
    public static void main(String[] args) {  
        Printer myPrinter = new Printer();  
        Manager myManager = new Manager();  
        Editor myEditor = new Editor();  
        GraphicIllustrator myGI = new GraphicIllustrator();  
        TechnicalWriter myTW = new TechnicalWriter();  
  
        myEditor.setName("Fred Hanson");  
        myEditor.setJobTitle("Editor");  
        myEditor.calculateEmployeeID();  
        myEditor.setLevel(1);  
        myEditor.setSkill("technical editing");  
        myEditor.setSkill("typing");  
        myEditor.setPrefersPaperEditing(true);  
        //myEditor.displayInformation();  
        myPrinter.printToScreen(myEditor);  
        System.out.println("\*\*\*\* \*\*\*\*\*");  
  
        myGI.setJobTitle("Graphic Illustrator");  
        myGI.setName("Frank Moses");  
        myGI.calculateEmployeeID();  
        myGI.setLevel(3);  
        myGI.setSkill("technical illustration");  
        myGI.setSkill("video production");  
        myGI.setSkill("media authoring");  
        //myGI.displayInformation();  
       ...

If there are still error indicators after making any changes, try clicking the **Save** button again or try clicking a line in EmployeeTest that indicates an error. This will remind the syntax checker in NetBeans to try resolving the references once more.  
  
You can then run the EmployeeTest class to test your program.

Graphic

*The output displays as  
  
run:  
Name: Fred Hansen  
Job Title: Editor  
Employee ID: 1  
Level: 1  
Employee has the following skills:  
technical editing  
typing  
Editing preference: Paper  
\*\*\*\* \*\*\*\*\*  
Name: Frank Moses  
Job Title: Graphic Illustrator  
Employee ID: 2  
Level: 3  
Employee has the following skills:  
technical illustration  
video production  
media authoring  
\*\*\*\* \*\*\*\*\*  
Name: James Ralph*

**5. Summary**

In this topic, you've learned how to create and use superclasses and subclasses.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#top)

Polymorphism, Interfaces, and the Object class

Learning Objectives

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

* *use polymorphism and override methods*
* *use Java interfaces*

**1. Using polymorphism**

The ability to use not just the class itself, but any superclass of the class as its reference type is known as polymorphism – a very important feature of Java.  
  
For example, you can refer to both a Shirt object and a Trousers object with a Clothing reference. This means that a reference to a Shirt or Trousers object can be passed in to a method that requires a Clothing reference. Or a Clothing array can contain references to Shirt, Trousers, or Socks objects.

Graphic

*The relevant code is  
  
Clothing*

Code

Shirt myShirt = new Shirt();  
  
Clothing clothingItem1 = new Shirt();  
Clothing clothingItem2 = new Trousers();

Accessing the methods of a class using a superclass reference is a little like accessing the controls of an electronic device using a remote control instead of the controls on the device itself.  
  
Often a device such as a video camera has a comprehensive set of controls for recording, playing, editing, and otherwise accessing every available function of the camera. This is a lot like using the class of the object as the reference type.

Graphic

*A video camera is displayed being operated via a push button remote control. The camera is labeled as having a full set of controls, while the remote is labeled as having a subset of controls.*

For some combinations of video camera and remote, the remote may give you exactly the same controls. This can also be the case when using a superclass as reference for an object. The superclass gives you access to all the methods of the object, while the object's class doesn't add any new methods.

Graphic

*A video camera is displayed being operated via a push button remote control. Both the camera, and the remote are labeled as having a full set of controls.*

It's often the case, however, that the remote control doesn't have the full set of controls available on the camera itself, and again, this is common when using the superclass as reference.  
  
The superclass only has access to the methods of the class that are declared on the superclass even if the class has a number of other methods.

For example, using a reference type Clothing will not allow access to the getFit() or getGender() methods of the Trouser object. Usually this isn't a problem as you're most likely to be passing Clothing references to methods that don't require access to these methods. A purchase() method could receive a Clothing argument as it only needs access to the getPrice() method.

Graphic

*A diagram displays the superclass Clothing, which contains a number of getter methods, such as getID() and getPrice(). The same methods are displayed under the Trousers object which has inherited them from the superclass. The Trousers object also has a couple of methods that are unique to it: getFit(), and getGender().*

Given that a superclass may not have access to all the methods of the object it's referring to, you can make the methods accessible by replacing the superclass reference with

Graphic

*A diagram displays the superclass Clothing, which contains a variety of getter methods, such as getSize(), and getColor(). A class reference for Trousers contains all of the same getter methods, as well as unique methods for Trousers: getFit(), and getGender(). Both the superclass methods and the unique methods are displayed in the Trousers object. The process indicates that the reference types has been casted by replacing the superclass with a reference that is the same type as the object.*

* a reference that is the same type as the object, as shown here, and
* an interface that declares the method or methods in question, and is implemented by the class of the object

For example, a Clothing reference can be cast to a Trousers reference to access the getFit() method, which isn't accessible via the Clothing reference. The inner parentheses around Trousers are part of the cast syntax, and the outer parentheses around (Trousers)cl are there to apply the cast to the Clothing type.

Graphic

*The relevant code is  
  
((Trousers)cl)*

Code

Clothing cl = new Trousers(123, "Dress Trousers", 'B', 17.00, 4, 'S');  
cl.display();  
  
//char fitCode = cl.getFit(); // This won't compile  
  
char fitCode = ((Trousers)cl).getFit(); // This will compile

Question

You have an abstract Vehicle class. The Brand class extends the Car class, which extends the Vehicle class, and overrides its getSpec method. You assign a Brand reference to a variable, vehicle, of type Vehicle.  
  
Which getSpec method is called through vehicle?

**Code**  
Vehicle vehicle = new Brand();  
vehicle.getSpec();

**Options:**

1. The getSpec method of the Car class
2. The getSpec method of the Brand class
3. The getSpec method of the Vehicle class

Answer

***Option 1:****Incorrect. You're using a base class reference variable to access an object of a subclass – a Brand object, in this case.*

***Option 2:****Correct. The Brand version of the getSpec method is executed, even though this method is different from the Vehicle getSpec method. The Vehicle reference can call the overriding getSpec method of any object of a class that extends the Vehicle class, without any change in its code.*

***Option 3:****Incorrect. Polymorphism ensures that the getSpec subclass version will be called over any superclass version.*

**Correct answer(s):**

2. The getSpec method of the Brand class

**2. Using Java interfaces**

A more complex set of classes may have items in two different hierarchies. If Duke's Choice were to start selling outdoors gear, they may have a different superclass called Outdoors, with its own set of subclasses – for example, getWeight() as an Outdoors method.  
  
There may be some classes from each hierarchy that have something in common. For example, the custom shirt item in Clothing is not returnable since it's made by hand for a particular person. Neither is the Stovefuel item in the Outdoors hierarchy. All other items are returnable.

Graphic

*A diagram displays the Clothing and Outdoors superclasses, each with related getter methods. Each superclass is linked to related subclasses. Clothing is linked to Shirt, Trousers, and Custom Shirt. Outdoors is linked to Tent, Camp stove, and Stove fuel.*

To model this, you should consider the following:

* A new superclass won't work because a class can only extend one superclass, and all items are currently extending either Outdoors or Clothing.
* A new field named returnable, added to every class, could be used to determine if an item can be returned. This is possible, but then there's no single reference type to pass to a method that initiates or processes a return.
* A special type called an Interface can be implemented by any class. This Interface type can then be used to pass a reference of any class that implements it.

All returnable items can implement the Returnable interface with its single method, doReturn(). Methods can only be declared in an interface, so each class that implements Returnable must implement doReturn() for itself. All returnable items could be passed to a processReturns() method of a Returns class and then have their doReturn() method called.

Graphic

*The diagram displays the Returnable interface linked to the subclasses under the superclasses Clothing and Outdoors that can implement the doReturn() method: Shirt, Trousers, Tent, and Camp stove. The subclasses Custom shirt and Stove fuel are excluded.*

This example shows the Returnable interface and the Shirt class. Only the constructor and the doReturn() methods are shown.

Code

public interface Returnable {  
    public String doReturn();  
}  
  
public class Shirt extends Clothing implements Returnable {  
    public Shirt(int itemID, String description, char colorCode,  
            double price, char fit) {  
        super(itemID, description, colorCode, price);  
        this.fit = fit;  
    }  
public String doReturn() {  
    // See notes below  
    return "Suit returns must be within 3 days";  
}  
...< other methods not shown > ... } // end of class

**public String doReturn();**

*The Returnable interface, like an abstract method, only has the method stub.*

**implements Returnable**

*Returnable provides a marker to indicate that the item can be returned, and ensures that the developer of Shirt must implement all methods of Returnable, such as the doReturn() method.*

**public String doReturn() {**

*The doReturn() method is declared in the Returnable interface, and returns a String describing the conditions for returning the item.*

The reference used to access an object determines the methods that can be called on it. For example, the doReturn() method can be called for the reference Returnable. If a method receives a Returnable reference, however, and needs access to methods on Clothing or methods on Trousers, the reference can be cast to the appropriate reference type.

Graphic

*The diagram displays the superclass reference for Clothing with getter methods such as getID() and getPrice(). These methods are also shown as inherited by the Object subclass of Trousers, which also has the two unique methods getFit(), and getGender(). The interface reference Returnable is displayed with the doReturn() method. This method is also implemented under the Trousers object.*

Some of the best examples of inheritance and the utility of Interface and Abstract types can be  
found in the Java API. For example the ArrayList class extends AbstractList class, which itself extends AbstractCollection.  
  
AbstractCollection implements the List interface, which means that ArrayList also implements the List interface. To use the ArrayList as a List, use the List interface as the reference type.

The List interface is implemented by many classes. This means that any method that requires a List may actually be passed a List reference to any objects of these types, but not the abstract classes because they cannot be instantiated.

All classes have at the very top of their hierarchy the Object class. It's so central to how Java works that all classes that don't explicitly extend another class, automatically extend Object.  
  
This means that all classes have access to the methods of Object. Being the root of the object hierarchy, Object doesn't have many methods, only very basic ones that all objects must have.

Code

public class Object

An interesting method is the toString() method. The Object toString() method gives very basic information about the object. Generally, classes will override the toString() method to provide more useful output. The System.out.println() uses the toString() method on an object passed to it to output a string representation.  
  
All objects have a toString() method because it exists in the Object class. But the toString() method may return quite different results depending on whether that method has been overridden.

Code

public String toString() {

In this example, toString() is called, via the println() method of System.out, on four objects.

Code

public class TestToString {  
      
    public static void main(String[] args) {  
  
  
        // Output an Object to the console  
        System.out.println(new Object());  
  
        // Output this StringBuilder object to the console  
        System.out.println(new StringBuilder("Some text for StringBuilder"));  
  
        //Output a class that does not override the toString() method  
        System.out.println(new First());  
  
        //Output a class that \*does\* override the toString() method  
        System.out.println(new Second());  
  
    }  
}

**System.out.println(new Object());**

*An Object object calls the toString() method of the base class. It returns the name of the class (java.lang.Object), an ampersand symbol, and a hash of the object, which is a unique number associated with the object.*

**System.out.println(new StringBuilder("Some text for StringBuilder"));**

*A StringBuilder object calls the toString() method on the StringBuilder object. StringBuilder overrides the toString() method that it inherits from Object to return a String object of the set of characters it's representing.*

**System.out.println(new First());**

*An object of type First, a test class, is a class with no code, so the toString() method called is the one that is inherited from the Object class.*

**System.out.println(new Second());**

*An object of type Second, a test class, is a class with one method named toString(), so this overridden method will be the one that is called. There is a case for re-implementing the getDescription() method used by the Clothing classes to instead use an overridden toString() method.*

Question

Which code segments, if included in an interface, would cause a compiler error?

**Options:**

1. protected void selectShipmentType(int type);
2. public final void printShippingDetails();
3. static final int MAX = 100;
4. void printShippingDetails();

Answer

***Option 1:****Correct. Declaring a method or variable as protected would generate a compile error, because the method or variable would be hidden from an implementing class. The purpose of an interface is to make its methods available to all classes that implement that interface.*

***Option 2:****Correct. Declaring an interface method as final implies that the method cannot be modified in any class that implements the interface. The final keyword would generate a compile error, because all methods are meant to be implemented by one or more classes.*

***Option 3:****Incorrect. In this case, using the static and final keywords is acceptable, because you are creating and initializing a variable. The variable MAX will retain the value 100, and cannot be re-assigned.*

***Option 4:****Incorrect. Methods declared in an interface are implicitly public, whereas variables are implicitly static and final. You are not required to specify these modifiers when declaring interface methods and variables.*

**Correct answer(s):**

1. protected void selectShipmentType(int type);  
2. public final void printShippingDetails();

**3. Creating a Java interface**

You want to create an interface called Printable and implement it within an existing class hierarchy. You also want to examine and run another small application that uses the same Printable interface in order to better understand the benefits of using interfaces.

To create a new Java Interface using the NetBeans New File wizard, follow these steps:

Graphic

*The NetBeans IDE 7.0.1 interface consists of a toolbar and the Projects window with three tabs: Projects, Files, and Services. The Project window displays the file tree for Practice12. The Source Packages folder is expanded and contains a number of files, including Company.java, CompanyTest.java, Country.java, and EmployeeTest.java.*

* click **File** - **New File**, and   
  *The File menu is selected. It contains options such as New Project, New File, Open Project, Open Recent Project, Open File, Open Recent File, and Import Project.*
* on the Choose File Type page, select **Java Interface** in the File Types column, name the interface Printable, and click **Finish**   
  *The Choose File Type section of the New File wizard consists of the Project text box, the Categories list box, and the File Types list box. Java is selected in the Categories list and a variety of Java file types display in the File Types list box, including Java Class, Java Interface, Java Enum, and Java Annotation Type.*

In the Printable interface, you declare a public abstract method called print. It should return void and accept zero arguments. You can then click **Save**.

Graphic

*The relevant code is  
  
public abstract void print();*

Code

/\*  
 \* To change this template, choose Tools | Templates  
 \* and open the template in the editor.  
 \*/  
  
/\*\*  
 \*  
 \* @author Administrator  
 \*/  
public interface **Printable** {  
    public abstract void print();      
}

Then you implement the Printable interface in the Employee class. All of the other classes in this hierarchy are subclasses of Employee, therefore, they also now implement Printable through inheritance.

Graphic

*The relevant code is  
  
public class Employee implements Printable*

Code

/\*  
 \* To change this template, choose Tools | Templates  
 \* and open the template in the editor.  
 \*/  
  
/\*\*  
 \*  
 \* @author Administrator  
 \*/  
public class **Employee** implements Printable {  
    private String name;  
    private String jobTitle;  
    private int employeeID;  
    private int level;  
    protected static int *employeeIDCounter* = 0;  
  
    public void **displayInformation**(){  
        System.*out*.println("Name: "+ getName());  
        System.*out*.println("Job Title: "+ getJobTitle());  
        System.*out*.println("Employee ID: "+ getEmployeeID());  
        System.*out*.println("Level: "+ getLevel());  
   }

The syntax checker shows an error icon in the margin of line 10. You can move your cursor over the error icon to see the potential compilation error that it recognizes.  
  
Any non-abstract classes that implement an interface must also implement all of the abstract methods of the interface. In this case, the only abstract method in Printable is print.

You change the name of the displayInformation method to print. You make this same change in the Manager, SkilledEmployee, and Editor classes to ensure that they also implement the print method.   
  
You also need to change the name of the superclass method called in the first line of the new print method, because it's no longer called displayInformation.

Graphic

*The relevant code is  
  
public void print()*

Code

/\*  
 \* To change this template, choose Tools | Templates  
 \* and open the template in the editor.  
 \*/  
  
/\*\*  
 \*  
 \* @author Administrator  
 \*/  
public class **Employee** implements Printable {  
    private String name;  
    private String jobTitle;  
    private int employeeID;  
    private int level;  
    protected static int *employeeIDCounter* = 0;  
  
    public void **print**(){  
        System.*out*.println("Name: "+ getName());  
        System.*out*.println("Job Title: "+ getJobTitle());  
        System.*out*.println("Employee ID: "+ getEmployeeID());  
        System.*out*.println("Level: "+ getLevel());  
   }

Then you open the Printer class in the editor and examine its only method – printToScreen. This method takes an argument of type Printable. Any class that implements Printable would be accepted as an argument. This method invokes the print method of the Printable object.

Graphic

*The relevant code is  
  
public void printToScreen(Printable p){  
p.print();  
}*

Code

/\*  
 \* To change this template, choose Tools | Templates  
 \* and open the template in the editor.  
 \*/  
  
/\*\*  
 \*  
 \* @author Administrator  
 \*/  
public class **Printer** {  
    public void printToScreen(Printable p){  
        p.print();  
    }  
      
    // This is a utility class that could potentially contain    
    //   many otherprint-related methods and fields  
}

In the EmployeeTest class, you make several changes.

Code

    /\*\*  
     \* @param args the command line arguments  
     \*/  
    public static void ***main***(String[] args) {  
        Printer myPrinter = **new** Printer();  
        Manager myManager = **new** Manager();  
        Editor myEditor = **new** Editor();  
        GraphicIllustrator myGI = **new** GraphicIllustrator();  
        TechnicalWriter myTW = **new** TechnicalWriter();  
  
        myEditor.setName("Fred Hanson");  
        myEditor.setJobTitle("Editor");  
        myEditor.calculateEmployeeID();  
        myEditor.setLevel(1);  
        myEditor.setSkill("technical editing");  
        myEditor.setSkill("typing");  
        myEditor.setPrefersPaperEditing(true);  
        //myEditor.displayInformation();  
        myPrinter.printToScreen(myEditor);  
        System.*out*.println("\*\*\*\* \*\*\*\*\*");

**Printer myPrinter = new Printer();**

*You use this code to declare and create an instance of the Printer class.*

**myPrinter .printToScreen**

*For every invocation of the displayInformation method, comment out the line and invoke the printToScreen method of the Printer object.*

**myEditor**

*You use this code to pass in a reference to the Printable object.*

You run the EmployeeTest class to examine the output. It should be identical to the output you saw before implementing the interface.  
  
One of the benefits of using interfaces is that you can abstract functionality that is used in different applications and different class hierarchies. This functionality is moved into the interface and can then be used anywhere that the functionality is required. For example, the ability to display class fields with labels and formatting is moved into the Printable interface.  
  
You can test the cross-application benefit by running a different application that also implements Printable. You could use the Company class hierarchy to display information about Duke’s Choice top-level management, as well as that of its regional and divisional management. The code would be similar to that in the Employee hierarchy.

Code

run:  
Name: Fred Hansen  
Job Title: Editor  
Employee ID: 1  
Level: 1  
Employee has the following skills:  
technical editing  
typing  
Editing preference: paper  
\*\*\*\* \*\*\*\*\*  
Name: Frank Moses  
Job Title: Graphic Illustrator  
Employee ID: 2  
Level: 3  
Employee has the following skills:  
technical illustration  
video production  
media authoring  
\*\*\*\* \*\*\*\*\*

To do this, you close all of the classes you've been working on and, in the editor, open the following classes that have already been created:

* Company   
  *The highlighted node is company.java.*
* Region   
  *The highlighted node is region.java.*
* Country, and   
  *The highlighted node is country.java.*
* CompanyTest   
  *The highlighted node is CompanyTest.java.*

You examine the Company class first. This is the superclass of Region and Country. It implements the same Printable interface that you used in the Employee hierarchy. You can then examine the Region, Country, and CompanyTest classes in turn to ensure the same interface is referenced. You can run the CompanyTest class if you want to view the output of this application.

Graphic

*The relevant code is  
  
public class Company implements Printable{*

Code

/\*\*  
 \*  
 \* @author Administrator  
 \*/  
public class **Company** implements Printable{  
    private String companyName;  
    private int taxID;  
    private String CEO;  
      
    // Implement the method of the Printable interface  
    public void print(){  
        System.*out*.println("Company: "+ getCompanyName());  
        System.*out*.println("TaxID: "+ getTaxID());  
        System.*out*.println("The CEO is "+ getCEO());  
        System.*out*.println("\*\*\*\*\*\*\*  \*\*\*\*\*\*");  
    }  
  
    public String getCompanyName() {  
        return companyName;  
   }

**4. Summary**

In this topic, you've learned how to use polymorphism and override methods. You've also learned how to use Java interfaces.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jsef_a04_it_enus/output/html/course_transcript.html#top)

© 2016 Skillsoft Ireland Limited