**Shift Into Override**



[Overriding](javascript:void(0);) a method is something already very familiar to you in the real world. Before you object, let us explain. Take a child and her parents. Let us say the parents were born in Florida, but the child was raised in London. While the parents would speak with a Floridian accent, the child will most likely speak with an accent more similar to the environment in which she was raised. This is an example of overriding. The speak method of the child overrides the speak method of the parents.

Another example can be found in vehicles. A car is a vehicle. A bicycle is also a vehicle. Both have ride methods; however, the ride method for a car is very different from the ride method of a bicycle. If I told you that I had invented a new vehicle called a "widget vehicle," you would immediately assume there was a ride method. In fact, you would probably ask how you ride in it!

When dealing with Java classes, sometimes a subclass may need to [override](javascript:void(0);) a method instead of inheriting it from the superclass. What this means is that the subclass redefines the method using the same name, return type, and number and type of arguments. Any object of the subclass type will use the newer version of the method rather than inherit it. This process even works with methods that come from the Object class, which is a superclass of all classes.

An example is the toString() method. This method exists in the Object class and provides a string to be displayed when a print statement is executed. The string produced is just a reference to the particular object's memory location. Not terribly useful when attempting to communicate output to the user. Thanks to overriding, we can solve this by adding our own toString() method to the any implementation class.

Any inherited method can be overridden as needed.

[Earlier](https://www.eimacs.com/eimacs/mainpage?epid=E203650864&cid=162149#AutoCastToString) we mentioned the toString method that is called automatically by Java when an object is implicitly cast to a string. Since this method is an instance method of the Object class, and since all classes inherit from Object, the toString method is available to all objects.

Consider the following class, for example:

public class School   
{   
  private String myName;   
  
  public School( String name )   
  {   
    myName = name;   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
}

If s is a School object, then these two expressions:

"The object s is " + s.toString()  
"The object s is " + s

give rise to the same output. Run the following program to confirm this:

School s = new School( "Springfield HS" );

System.out.println( "The object s is " + s.toString() );

System.out.println( "The object s is " + s );

It should be clear from this example that, although the toString method does a fine job of unambiguously identifying a School object, the output it produces is not exactly user-friendly. As human beings, we are better able to recognize a school from its name rather than from an identifier that references the memory location in which the data regarding the school is stored. It is therefore of interest to know that Java allows us to customize the toString method specifically for School objects. To do this, all we have to do is include the customized definition within the class definition for the School class, like this:

public class School   
{   
  private String myName;   
  
  public School( String name )   
  {   
    myName = name;   
  }   
  
  public String toString()   
  {   
    return getName();   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
}

Note that, when we redefine the toString method in this way, it is the redefined version that is used, even in situations in which toString is called implicitly.

School s = new School( "Springfield HS" );

System.out.println( "The object s is " + s.toString() );

System.out.println( "The object s is " + s );

If we provide the School class with a customized version of the toString method, then it is that customized version that is inherited by subclasses of School. For example, suppose we use the version of toString from [Exercise 131](https://www.eimacs.com/eimacs/mainpage?epid=E201976356&cid=162149#Exe118)(a) and suppose we create a new class to represent elementary schools:

public class School   
{   
  private String myName;   
  
  public School( String name )   
  {   
    myName = name;   
  }   
  
  public String toString()   
  {   
    return "School: " + getName();   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
}   
  
public class ElementarySchool extends School   
{   
  public ElementarySchool( String name )   
  {   
    super( name );   
  }   
}

Since ElementarySchool inherits from School, when we call the toString method of an ElementarySchool object it will use the method provided by the School class rather than the one from the Object class.

  public static void main( String[] args )   
  {   
    ElementarySchool e = new ElementarySchool( "Eisenhower Elementary" );   
  
    System.out.println( e );   
  }

When we give the School class or the Employee class a customized version of the toString method, we are said to *override* the toString method of the Object class. The technique does not apply only to the toString method. To override an instance method of a class, we have only to include a new definition for that method in the definition of one of the subclasses of that class. Viewed from the other direction, one instance method (Method #2, say) overrides another (Method #1, say) if and only if

* Method #1 is defined in a *superclass* of the class in which Method #2 is defined; and
* Methods #1 and #2 have [matching signatures](https://www.eimacs.com/eimacs/mainpage?epid=E2124391658&cid=162149#SigMatch); and
* Methods #1 and #2 have the same return data type.

To illustrate, the following definitions introduce classes that represent customers of a home improvement store. The price method calculates the price to be charged to the customer for a particular total value of goods purchased:

public class Customer   
{   
  private String myName;   
  
  public Customer( String name )   
  {   
    myName = name;   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
  
  public double price( double total )   
  {   
    return total;   
  }   
}   
  
public class TradePerson extends Customer   
{   
  public TradePerson( String name )   
  {   
    super( name );   
  }   
  
  public double price( double total )   
  {   
    return 0.95 \* total;   
  }   
}

Notice that a (regular) customer pays the full price, whereas someone in the building trade gets a 5% discount.

Run the following program using a total of $200.00 and various other totals of your own choosing. Verify that the prices paid by a regular customer and a tradeperson are different.

  public static void main( String[] args )   
  {   
    Customer c1 = new Customer( "Fred" );   
    Customer c2 = new TradePerson( "Joe" );   
  
    double total = ;   
  
    System.out.println( "Fred pays $" + c1.price( total ) );   
    System.out.println( "Joe pays $" + c2.price( total ) );   
  }

Notice that the above code makes use of polymorphism. The variable c2 is declared to be a Customer object, but it is initialized to a TradePerson object. Since the TradePerson class has a price method that overrides the price method of the Customer class, this is an opportune moment to tell you officially how Java handles the situation when both polymorphism and overriding are involved. The rule is this:

Suppose that the variable v is declared to have data type D and is initialized to an instance of a subclass S of D. If D has an instance method m that is subsequently invoked on v, then Java checks whether or not S has its own customized version of m with the appropriate signature. If so, it uses that customized version. If not, Java looks back up the class hierarchy, starting at S and going toward D, checking at each stage for a suitable method m. It uses the first one it finds in this "bottom up" search.

In the case of Exercise 134, Java notices that the subclass TradePerson has it own customized version of the price method. So it uses that version when evaluating

c2.price( total ).

In our definition of the TradePerson class, we override the price method of its direct superclass (Customer) like this:

public class TradePerson extends Customer   
{   
  public TradePerson( String name )   
  {   
    super( name );   
  }   
  
  public double price( double total )   
  {   
    return 0.95 \* total;   
  }   
}

In effect, we *replace* the price method of the direct superclass with a method that simply ignores the one it is overriding. An alternative approach is to preserve the method being overridden for some purposes while modifying it for others. For example, suppose the home improvement store changes the rules so that, from now on, a trade person is given the 5% discount only when the total exceeds $50. If the total is $50 or less, then the trade person is treated just like a regular customer. This means that, for the lower totals, the price paid by a trade person is to be calculated using the price method of its direct superclass, Customer. In Java, we access a method of the direct superclass of a class by means of an expression formed by writing:

* the keyword super followed by
* a period, ., followed by
* the name of the method (in this case, price).

So, after the rule change, the new definition of the price method for the TradePerson class might start like this:

public class TradePerson extends Customer   
{   
  public TradePerson( String name )   
  {   
    super( name );   
  }   
  
  public double price( double total )   
  {   
    if ( total <= 50.0 )   
      return super.price( total );   
  
    // ...   
  }   
}

(Notice the two uses of the keyword super in this code.)

If the total is more than $50, then we apply the trade person discount:

public class TradePerson extends Customer   
{   
  public TradePerson( String name )   
  {   
    super( name );   
  }   
  
  public double price( double total )   
  {   
    if ( total <= 50.0 )   
      return super.price( total );   
  
    return 0.95 \* total;   
  }   
}

In the preceding pages, we have seen that the Object class provides a toString method that is inherited by and may be overridden in subclasses. In addition, the Object class provides a second instance method that is commonly overridden in subclasses, namely, the equals method.

The equals method of the Object class accepts one argument, which must itself be an instance of Object, and returns a boolean. The method returns true if and only if the object given as the argument is the same as the object whose instance method is being called. So, for example, the following code — which uses a form of the shorthand we introduced [earlier](https://www.eimacs.com/eimacs/mainpage?epid=E202813250&cid=162149) for declaring several variables of the same data type — outputs the boolean true:

public class Complex   
{   
  private double myReal,   
                 myImag;   
  
  public Complex( double real, double imag )   
  {   
    myReal = real;   
    myImag = imag;   
  }   
  
  public double getReal()   
  {   
    return myReal;   
  }   
  
  public double getImag()   
  {   
    return myImag;   
  }   
}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    Complex c = new Complex( -1.0, 2.0 );   
  
    System.out.println( c.equals( c ) );   
  }   
}

whereas, with the same definition of Complex, the following code outputs false. (Why?)

public class MainClass   
{   
  public static void main( String[] args )   
  {   
    Complex c = new Complex( -1.0, 2.0 );   
    Complex d = new Complex( -1.0, 2.0 );   
  
    System.out.println( c.equals( d ) );   
  }   
}

The reason is that, although c and d share the same real and imaginary parts, they are *not* the same instance of Complex.

In circumstances like this, it is common practice to override the equals method of Object, providing a customized version whose behavior is more useful. In the case of complex numbers, two such numbers are held to be equal if their real parts are equal and their imaginary parts are equal. So a suitable override for equals is as follows:

public class Complex   
{   
  private double myReal,   
                 myImag;   
  
  public Complex( double real, double imag )   
  {   
    myReal = real;   
    myImag = imag;   
  }   
  
  public double getReal()   
  {   
    return myReal;   
  }   
  
  public boolean equals( Object b )   
  {   
    // instanceof is not in the AP Java subset   
    if ( ! (b instanceof Complex) )   
      return false;   
  
    Complex t = (Complex)b;   
    return t.getReal() == getReal() && t.getImag() == getImag();   
  }   
  
  public double getImag()   
  {   
    return myImag;   
  }   
}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    Complex c = new Complex( -1.0, 2.0 );   
    Complex d = new Complex( -1.0, 2.0 );   
  
    System.out.println( c.equals( d ) );   
  }   
}

As noted in a comment in the new equals method, we have used something that does not belong to the Advanced Placement Java subset. We have done this because there is a convention that any equals method should always return false when its argument cannot be cast to the data type of the object whose equals method is being called. In this case, when applied to an object that is not derived from Complex, we want this equals method to return false. In the absence of the instanceof check, the new equals method would instead cause a ClassCastException to occur in such circumstances.

In Java, [instanceof](javascript:void(0);) is a relational operator that behaves in a similar fashion to the [numerical relational operators](https://www.eimacs.com/eimacs/mainpage?epid=E1961788358&cid=162149) <, >=, !=, and so on. It returns a boolean as follows: the expression a instanceof b has the value true if and only if:

* the value of a is of a [reference data type](https://www.eimacs.com/eimacs/mainpage?epid=E1942115087&cid=162149#DataTypes); and
* b names a reference data type; and
* a can be cast to the data type named by b.

Since instance of does not belong to the AP Java subset, there will be no questions on the Advanced Placement examination that use it. You will not be penalized for using it, however, in your answers to free response questions. We make free use of it in the remainder of this course, especially in circumstances like this where we are overriding an equals method.

We emphasize that, in the overridden equals method above, once an argument has passed the instance of check, it must still actually be cast to Complex before we may reference calls to any of the instance methods of the Complex class to it. Run the above program to verify that, using the overridden equals method, the expression c.equals( d ) now returns the boolean true.

In the above definition of the Complex class, the version of multiple variable declaration used for declaring the instance variables is an obvious extension of what we introduced [earlier](https://www.eimacs.com/eimacs/mainpage?epid=E203650600&cid=162149). The only difference is that the combined declaration statement has the accessor modifier keyword private inserted before the name of the data type.

Before moving on, we remark in passing that, when its operands are instances of the Object class or one of its subclasses, the identity operator == behaves exactly like the equals method of the Object class. If the variables a and b are initialized, each to an instance of Object or one of its subclasses, then a == b evaluates to true if and only if a and b are identically equal, that is, if and only if they are one and the same instance of one and the same class.

### Part 2

Let's modify our simple Rectangle and Box classes by adding a toString method to practice overriding methods.

* Create a project called 09.04 Overriding Methods in the Module 09 Lessons folder.
* Download the following Java files to the newly-created folder:
  + [Rectangle3.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson04/docs/13_04b/Rectangle3.java)
  + [Box3.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson04/docs/13_04b/Box3.java)
  + [ShapesTester3.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson04/docs/13_04b/ShapesTester3.java)
* Open the [09.04 Virtual Lecture Notes](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson04/pop/13_04b/13_04b_popa.htm).