Having defined a class containing all the methods and instance variables shared by all our employee classes, we next define each of the more specialized classes by adding more methods and/or instance variables to the common core definition. That is, we *extend* the common core Employee class, and we indicate this relationship by employing a new keyword: extends. For example,

public class SalesAssistant extends Employee   
{   
  private int mySecureCode;   
  
  public SalesAssistant( String name, int employeeID, int secureCode )   
  {   
    super( name, employeeID );   
    mySecureCode = secureCode;   
  }   
  
  public int getSecureCode()   
  {   
    return mySecureCode;   
  }   
}

This definition informs Java that the SalesAssistant class is based on the Employee class. Note that the constructor for the SalesAssistant class takes three arguments. The first two arguments correspond to the two arguments of the Employee constructor, and the expression super( name, employeeID ) passes these arguments along to the Employee constructor.

Important note: When "extending" class definitions like this, it is important that the super statement in the extended class constructor should be the first statement in the constructor's body.

We have been careful to specify that subclasses inherit public methods from their superclasses. Conversely, we should also make it clear that constructors are not inherited (each class, whether or not it is a subclass of another, must provide its own constructor) and nothing private is inherited. In particular, private instance variables are not inherited. For example, the following code — which is a modification of our earlier definition of the SalesAssistant class — generates an error. Run the code and read the error message:

public class SalesAssistant extends Employee

{

  private int mySecureCode;

  public SalesAssistant( String name, int employeeID, int secureCode )

  {

    super( name, employeeID );

    mySecureCode = secureCode;

  }

  public boolean isTemporary()

  {

    // an assistant is a temp if the employeeID

    // is less than 100

    return (**myEmployeeID** < 100);

  }

  public int getSecureCode()

  {

    return mySecureCode;

  }

}

public static void main( String[] args )

{

  SalesAssistant s = new SalesAssistant( "Fred Brown", 18, 123 );

  System.out.println( "Name: " + s.getName()  );

  System.out.println( "ID: " + s.getEmployeeID()  );

  System.out.println( "Code: " + s.getSecureCode()  );

  System.out.println( "Temp?: " + s.isTemporary()  );

}

As the message indicates, the problem lies in the fact that the modified definition attempts to access one of the private instance variables of the Employee superclass. (Which one?) But the private marking means what it says! Those variables are not available to subclasses of the Employee class — nor indeed to anything outside the Employee class definition.

There are two ways to solve this problem. We could mark the Employee class's instance variables as public; they would then become available to any subclasses of the Employee class. Alternatively we could use the public getEmployeeID method that the SalesAssistant class inherits from its direct superclass.

There are good reasons for preferring the second alternative. In particular, by insisting that instance variables are only accessed through the corresponding accessor methods we make it possible to track all uses of the variables with the help of a debugger (by placing a breakpoint in the accessor method, for example). This is good programming practice. Secondly, as we mentioned earlier, in the Advanced Placement Computer Science Java subset all instance variables must be marked as private.

Change

public class SalesAssistant extends Employee

{

  private int mySecureCode;

  public SalesAssistant( String name, int employeeID, int secureCode )

  {

    super( name, employeeID );

    mySecureCode = secureCode;

  }

  public boolean isTemporary()

  {

    // an assistant is a temp if the employeeID

    // is less than 100

    return (getEmployeeID() < 100);

  }

  public int getSecureCode()

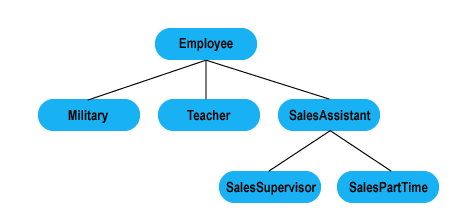
  {

    return mySecureCode;

  }

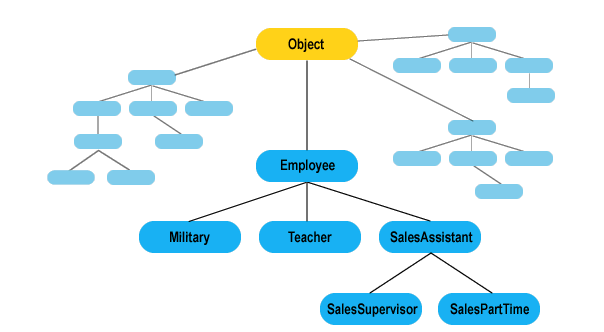
}

Recall that the class hierarchy we have been discussing for the past few pages may be pictured like this:



If you examine the definitions of all these classes, you will notice that the only one whose definition does *not* involve the extends keyword is the Employee class, at the top of the hierarchy. Despite the absence of this keyword, however, the Employee class *does* have a direct superclass. Java provides a special class, named Object, that it uses as the direct superclass for classes such as Employee that do not have an explicit direct superclass. It follows that all classes inherit from Object, either directly (as Employee does) or indirectly (as SalesAssistant does).

The following illustration pictures this relationship between the Object class and the Employee hierarchy:



We show Object in a different color to emphasize that it is a class that Java provides. The illustration also includes ghostly suggestions of other class hierarchies, unrelated to the Employee hierarchy, of which Object is a superclass. There is no limit to the number of such class hierarchies — every class that we, you, or anybody else defines without an explicit direct superclass automatically has Object as its direct superclass.

There are numerous useful consequences of the fact that the Object class is a common superclass for all classes. In particular, Object has a number of public instance methods that all its subclasses inherit. We meet several of them as this course proceeds — the first, on the next page.

One of the effects of Object's status as a universal superclass is that it leads to a method for discovering the name of any class. This involves a two-stage process. One of the Object class's instance methods is getClass. When this method is invoked on any object, it returns an instance of a special class, called Class. The Class object returned by getClass represents the class of the object on which the method is invoked.

The Class class in turn provides a number of useful instance methods, among them the getName method. In the case of a Class object returned by Object's getClass method, the getName method returns a String that names the object's class. So the two-stage process of discovering the name of the class of an object is this:

* Invoke the getClass method on the object.
* Invoke the getName method on the result of the first stage.

For example, if e is a variable storing an instance of the SalesAssistant class, then the expression

e.getClass().getName()

evaluates to the String "SalesAssistant".

  public static void main( String[] args )

   {

     SalesAssistant a =

           new SalesAssistant( "Ken Blank", 18, 123 );

     SalesSupervisor b =

           new SalesSupervisor( "Ken Blank", 18, 123, "aR62B" );

     Employee c = new Employee( "Ken Blank", 18 );

     System.out.println( a.getClass().getName() );

     System.out.println( b.getClass().getName() );

     System.out.println( c.getClass().getName() );

   }

 SalesAssistant   
SalesSupervisor   
Employee

The Class class is unusual. It has no public constructor; instances of it are created automatically by Java as the need arises while a program is running. Furthermore, each Class object *represents* the class of an object in much the same way that your student record in your school's computer system represents you. Your student record "knows" your name, your parents' names, your address, the courses you have taken, and so on. Similarly, a Class object that represents the class of an object A "knows" the name of A's class, the name of the direct superclass of that class, and so on.

We remark in passing that, even though the getName method of the Class class and the getName method of the Employee class have the same signature, namely,

String getName()

they come from different classes and are therefore different. Which one is applied in any particular situation depends upon the data type of the object on which it is called — a Class object will use the Class method and an Employee object will use the Employee method.

In addition to getClass, the Object class provides another instance method: toString. This takes no arguments and returns a String that represents the object. Run the following program to view the string that is returned for an actual SalesAssistant object:

public class Employee   
{   
  private String myName;   
  private int myEmployeeID;   
  
  public Employee( String name, int employeeID )   
  {   
    myName = name;   
    myEmployeeID = employeeID;   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
  
  public int getEmployeeID()   
  {   
    return myEmployeeID;   
  }   
}   
  
public class SalesAssistant extends Employee   
{   
  private int mySecureCode;   
  
  public SalesAssistant( String name, int employeeID, int secureCode )   
  {   
    super( name, employeeID );   
    mySecureCode = secureCode;   
  }   
  
  public int getSecureCode()   
  {   
    return mySecureCode;   
  }   
}   
  
public class SalesSupervisor extends SalesAssistant    
{   
  private String mySafeKey;   
  
  public SalesSupervisor( String name, int employeeID,    
                  int secureCode, String safeKey )   
  {   
    super( name, employeeID, secureCode );   
    mySafeKey = safeKey;   
  }   
  
  public String getSafeKey()   
  {   
    return mySafeKey;   
  }   
}   
  
    
public class MainClass   
{   
  public static void main( String[] args )  
  {  
    SalesAssistant a = new SalesAssistant( "Ken Blank", 18, 123 );  
    
    System.out.println( a.toString() );  
  }    
}

Output

SalesAssistant@659e0bfd

The characters that follow the "@" sign in this string give the internal name of the memory location where the SalesAssistant instance is stored. This information is often very helpful in debugging situations when we are trying to determine whether or not two instances of a class are in fact the same.

You may recall that we met something very like this [earlier](https://www.eimacs.com/eimacs/mainpage?epid=E2163739179&cid=162149#StrDoubleConv) when we were discussing converting numbers to strings and *vice versa*. There, we saw that applying Double.toString to a double or Integer.toString to an int has the effect of converting the numerical argument to the corresponding String. Clearly, each of these procedures involves a toString method. But in these cases the method takes an argument, so it cannot be the same as the Object class's toString method, which does not.

Even so, there are certain parallels in behavior. Recall that it is possible for a number to be converted *automatically* to a string, without invoking the Double.toString method or the Integer.toString method. All we have to do is concatenate the number with a string and Java takes care of the number-to-string conversion. In fact, as we pointed out at the time, the string in question could even be the empty string "". So,

"" + 5.625

evaluates to the String "5.625".

In a similar way, Java automatically converts to a String any instance of a subclass of the Object class that is involved in an expression that evaluates to a String. It does this by implicitly invoking the toString method on the object. For example, if a is a variable that stores an instance of SalesAssistant, then the expression "a is " + a behaves exactly as if it were "a is " + a.toString(), as the following code confirms:

public class Employee   
{   
  private String myName;   
  private int myEmployeeID;   
  
  public Employee( String name, int employeeID )   
  {   
    myName = name;   
    myEmployeeID = employeeID;   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
  
  public int getEmployeeID()   
  {   
    return myEmployeeID;   
  }   
}   
  
public class SalesAssistant extends Employee   
{   
  private int mySecureCode;   
  
  public SalesAssistant( String name, int employeeID, int secureCode )   
  {   
    super( name, employeeID );   
    mySecureCode = secureCode;   
  }   
  
  public int getSecureCode()   
  {   
    return mySecureCode;   
  }   
}   
  
public class SalesSupervisor extends SalesAssistant    
{   
  private String mySafeKey;   
  
  public SalesSupervisor( String name, int employeeID,    
                  int secureCode, String safeKey )   
  {   
    super( name, employeeID, secureCode );   
    mySafeKey = safeKey;   
  }   
  
  public String getSafeKey()   
  {   
    return mySafeKey;   
  }   
}   
  
    
public class MainClass   
{   
  public static void main( String[] args )  
  {  
    SalesAssistant a = new SalesAssistant( "Ken Blank", 18, 123 );  
    
    System.out.println( "Using toString, a is " + a.toString() );  
    System.out.println( "Without toString, a is " + a );  
  }    
}

Using toString, a is SalesAssistant@659e0bfd   
Without toString, a is SalesAssistant@659e0bfd

To avoid the inconvenience of working with specific data types when dealing with class hierarchies, Java allows us to store an instance of a class in a variable whose data type is that of a superclass of the stored object's class.

For example, the Employee class is a superclass — the direct superclass, in fact — of the SalesAssistant class:

So Java allows us to store a SalesAssistant object in an Employee variable, like this:

public class MainClass

{

  public static void main( String[]  args )

  {

    Employee e = new SalesAssistant( "Fred Bloggs", 123, 6 );

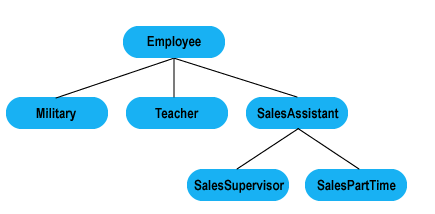
    System.out.println( e.getName() );

  }

}

Fred Bloggs

The technique of treating an object as though it were an instance of a superclass of its true class is called [polymorphism](javascript:void(0);). Note that the data type of the variable can be that of any superclass, not just the direct superclass, of the class of the object being stored.



By experimenting with the above program, answer the following questions:

1. Can a Teacher object can be stored in a variable of data type Employee?
2. Can a SalesPartTime object be stored in a variable of data type SalesAssistant?
3. Can a SalesAssistant object be stored in a variable of data type Teacher?
4. Can a SalesPartTime object can be stored in a variable of data type Employee?
5. Can a SalesAssistant object can be stored in a variable of data type SalesPartTime?
6. Yes. Employee is a superclass — in fact, the direct superclass — of Teacher.
7. Yes. SalesAssistant is a superclass — in fact, the direct superclass — of SalesPartTime.
8. No. An incompatible types error is generated because Teacher is not a superclass of SalesAssistant.
9. Yes. Employee is a superclass of SalesPartTime.
10. No. An incompatible types error is generated because SalesPartTime is not a superclass of SalesAssistant.

By using polymorphism we can overcome the [problem](https://www.eimacs.com/eimacs/mainpage?epid=E2193254360&cid=162149#WhatDataType) we were having as we tried to define printEmployeeDetails. Since the arguments to this method are all going to be instances of classes in the Employee class hierarchy, they will all either be Employee objects or instances of some subclass of Employee. We may therefore give the formal parameter the data type Employee:

static void printEmployeeDetails( Employee e )   
{   
  System.out.println( e.getName() + " " + e.getEmployeeID() );   
}

Experiment with the program below and confirm that the above version of printEmployeeDetails works in the expected manner when the argument is

1. a Teacher object;
2. a SalesAssistant object.

public class MainClass

{

  static void printEmployeeDetails( Employee e )

  {

    System.out.println( e.getName() + " " + e.getEmployeeID() );

  }

  public static void main( String[] args )

   {

     Teacher t = new Teacher( "Sue Cassidy", 1024, 215 );

     SalesAssistant a =

           new SalesAssistant( "Ken Dodd", 44311, 2477 );

     printEmployeeDetails( t );

     printEmployeeDetails( a );

   }

}

Sue Cassidy 1024   
Ken Dodd 44311

Now, the variable f, as defined in the code above, has data type Employee. In fact, though, it actually contains an instance of SalesAssistant. This means that f has a mySecureCode instance variable. So the question arises: Can we directly access the value of that variable? The following code involves an attempt to do so. Click the **Run** button to find out what happens:

public class MainClass

{

  public static void main( String[]  args )

  {

    Employee e = new Teacher( "Fred Bloggs", 123, 5 );

    Employee f = new SalesAssistant( "Erica Black", 456, 95 );

    System.out.println( e.getName() );

    System.out.println( f.getSecureCode() );

  }

}

MainClass.java:12: error: cannot find symbol   
  
    System.out.println( f.getSecureCode() );   
  
                         ^

When this code is executed, a cannot resolve symbol error is generated and the error message indicates that the error is provoked by the symbol getSecureCode, referenced to f. This is Java's way of complaining that f does not have a getSecureCode method. Evidently, Java has not noticed that in actual fact f contains a SalesAssistant object. It follows that, if we want to access the secure code of that object, we must first draw to the Java compiler's attention the fact that f contains an instance of SalesAssistant. We do this using a cast.

The following code fragment illustrates two different strategies for obtaining the secure code. In the first, the instance stored in f is cast to an instance of SalesAssistant and the result is stored in a new variable g. At this point, the Java compiler knows that g has a getSecureCode method that we may use to obtain the secure code. The second strategy combines these two steps, casting the variable f to SalesAssistant and then immediately calling the getSecureCode method on the result, without the intermediate step of initializing a new variable:

public class MainClass

{

  public static void main( String[]  args )

  {

    Employee f = new SalesAssistant( "Erica Black", 456, 95 );

    // cast, store in a new variable, then call method

    SalesAssistant g = (SalesAssistant)f;

    System.out.println( g.getSecureCode() );

    // cast and call the method directly

    System.out.println( ((SalesAssistant)f).getSecureCode() );

  }

}

The second strategy above uses the expression

((SalesAssistant)f).getSecureCode()

to obtain the secure code. What happens if the expression

(SalesAssistant)f.getSecureCode()

is used instead, omitting the outer pair of parentheses from ((SalesAssistant)f)?

When Java attempts to evaluate the expression

(SalesAssistant)f.getSecureCode()

it tries to do so by first evaluating f.getSecureCode() and then casting the result to SalesAssistant. But this fails at the first stage (with a cannot resolve symbol error) because the compiler does not yet know that f contains an instance of SalesAssistant. In order to force Java to do things in the right order — casting to SalesAssistant first, and then invoking the getSecureCode method — we must enclose the cast in parentheses, like this:

((SalesAssistant)f).getSecureCode()

Not surprisingly, an error occurs — specifically, a ClassCastException is thrown. This happens whenever Java attempts to cast an instance to a class other than the actual class of the instance or one of its superclasses.

This is where the getClass method of the Object class comes into its own. We introduced you to this [method](https://www.eimacs.com/eimacs/mainpage?epid=E2104718735&cid=162149) before polymorphism had entered the picture, so the full significance of the following description could not have been apparent to you:

The Class object returned by getClass represents the class of the object on which the method is invoked.

In this statement, "the class of the object" means the actual class of the object, regardless of how it may have been declared or stored. So if we make the following declaration:

Employee e = new Teacher( "Fred Thompkins", 55, 525 ) );

or if, as in the code above, we add Mr. Thompkins as the first element of the employees ArrayList of Employee objects, then the expressions

e.getClass().getName()

or

employees.get( 0 ).getClass().getName()

will both return the String "Teacher" because, deep down inside, Mr. Thompkins is a Teacher, despite his appearance on the surface as an Employee.

This means that the getClass method provides us with a way to select from the employees ArrayList just those elements that have a getSecureCode method. It enables us to check the name of the actual class of each element before performing the cast:

public class MainClass   
{   
  public static void main( String[] args )   
  {   
    ArrayList<Employee> employees = new ArrayList<Employee>();   
  
    employees.add( new Teacher( "Fred Thompkins", 55, 525 ) );   
    employees.add( new SalesAssistant( "Eric Washington", 7, 72 ) );   
    employees.add( new Military( "Albert Costa", 236237, "Navy", "Seaman" ) );   
    employees.add( new Teacher( "Jane Austin", 724, 92 ) );   
    employees.add( new SalesAssistant( "Jane Black", 91, 295 ) );   
    employees.add( new Employee( "Scott Black", 23 ) );   
    employees.add( new SalesPartTime( "Janice Dell", 552, 501, 8.0 ) );

for ( Employee e : employees )

{

  String className = e.getClass().getName();

  if ( className.equals( "SalesAssistant" )

        || className.equals( "SalesPartTime" ) )

  {

    SalesAssistant a = (SalesAssistant)e;

    System.out.println( a.getSecureCode() );

  }

}

}   
}

72   
295   
501

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 );

The object s is Springfield HS   
The object s is Springfield HS

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=E2222766342&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 );   
  }

[Show program details »](https://www.eimacs.com/eimacs/mainpage?cid=162149&epid=E2355858172)

School: Eisenhower Elementary

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=E1922439810&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.

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.

#### Exercise 134

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 ) );   
  }

[Show program details »](https://www.eimacs.com/eimacs/mainpage?cid=162149&epid=E2104719141)

Fred pays $500.0   
Joe pays $475.0

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=E2144066250&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 ) );   
  }   
}

true

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 ) );   
  }   
}

false

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:

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 ) );   
  }   
}

true

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=E2163739294&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=E2193253980&cid=162149#DataTypes); and
* b names a reference data type; and
* a can be cast to the data type named by b.

Since instanceof 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 instanceof 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=E2385367450&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.