**Polymorphic Methods**



Sometimes a class can have several subclasses. For a real-world example, consider the car. Everyone has in their mind a definition of a basic car. When a child sees a vehicle driving down the street that is small, compact, and has four wheels, he or she might shout, "Look, a car!" This demonstrates the **is-a** relationship. For example, a Toyota Corolla is-a car, a Dodge Viper is-a car, and a Mercedes Benz is-a car. Therefore, one could say that these three cars are subclasses of the original car superclass.

There are some options available when we want to retrieve the common information from objects we have created from subclasses.

One choice is to write a separate method for each type of object created. This involves more work than is necessary. Thanks to the way Java handles classes, a lot of time and effort can be saved.

For example, if we wanted to make a door that would allow a child, a teenager, and an adult to pass through, we can just create a single door that allows a human being to pass through it. Since a child, a teenager, and an adult are all derived from the human being class, they can all pass through the door made for a human being.

This concept of writing methods that work in this manner is called polymorphism. [Polymorphism](javascript:void(0);) involves our ability to write one method to handle objects that are related through inheritance, rather than multiple separate methods.

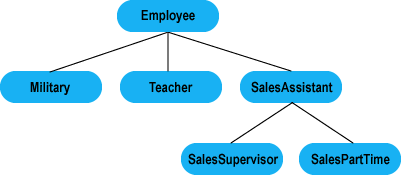
### Part 1

Understanding inheritance and polymorphism takes time, study, and practice. Pay close attention to the following eIMACS labs and these important OOP concepts will become clearer. Also, complete the practice test. It will test your knowledge and help make sure you understand what you have learned so far.

Recall that, when it processes a class definition, Java creates a new data type. This gives rise to a convenient way of declaring variables to hold instances of the classes we define, like this:

  SalesAssistant t = new SalesAssistant( "Eric Black", 123, 555 );

When working with a collection of classes that are related in a class hierarchy, however, the fact that each class in the hierarchy has its own data type can be a significant *in*convenience. Let us illustrate, using the Employee hierarchy:



Let's suppose that you have taken a census of some of the people in your neighborhood for the purpose of making a database of employee types, and have discovered the following information:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **ID** | **Other** |
| Fred Thompkins | Teacher | 55 | Room #523 |
| Eric Washington | SalesAssistant | 7 | Register key: 72 |
| Albert Costa | Military (Navy) | 236237 | Rank: Seaman |
| Jane Austin | Teacher | 724 | Room #92 |
| Jane Black | SalesAssistant | 91 | Register key: 295 |
| Scott Black | Employee | 23 | – |
| Janice Dell | SalesPartTime | 552 | Register key: 501 8 hours per week |

Your initial plan is to model this situation by creating an object for each of the data records, like this:

  Teacher e1 = new Teacher( "Fred Thompkins", 55, 525 );   
  SalesAssistant e2 = new SalesAssistant( "Eric Washington", 7, 72 );   
  Military e3 = new Military( "Albert Costa", 236237, "Navy", "Seaman" );   
  Teacher e4 = new Teacher( "Jane Austin", 724, 92 );   
  SalesAssistant e5 = new SalesAssistant( "Jane Black", 91, 295 );   
  Employee e6 = new Employee( "Scott Black", 23 );   
  SalesPartTime e7 = new SalesPartTime( "Janice Dell", 552, 501, 8.0 );

Next, you would like to write a static method that takes a single argument, namely, one of the objects above, and writes out the employee's name and ID — something like this:

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

But now you have a problem. You must specify a data type for the formal parameter of this method. What should it be? If you choose Teacher as the data type, then the method will only accept instances of Teacher as arguments. Similarly, if you specify a data type of SalesAssistant, then the method will only work on instances of SalesAssistant.

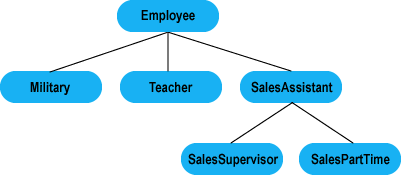
One solution to this problem is to use [method overloading](https://www.eimacs.com/eimacs/mainpage?epid=E2114554228&cid=162149) and define a method for each data type in our class hierarchy. Then Java can select which method to use according to the data type of the argument:

public class MainClass   
{   
  public static void printEmployeeDetails( SalesAssistant e )   
  {   
    System.out.println( e.getName() + " " + e.getEmployeeID() );   
  }   
  
  public static void printEmployeeDetails( Teacher e )   
  {   
    System.out.println( e.getName() + " " + e.getEmployeeID() );   
  }   
  
  public static void main( String[]  args )   
  {   
     SalesAssistant s = new SalesAssistant( "Fred Bloggs", 123, 6 );   
     Teacher t = new Teacher( "Jane Bloggs", 456, 7 );   
  
     printEmployeeDetails( s );   
     printEmployeeDetails( t );   
  }   
}

But that involves a lot of work — and the more classes there are in a hierarchy, the more work you would have to do — defining a different version of the same method for each class in the hierarchy. Fortunately, Java provides a better way to solve this problem.

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() );

}

}

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 using polymorphism we can overcome the [problem](https://www.eimacs.com/eimacs/mainpage?epid=E2212928924&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() );   
}

In the following program fragment, instances of Teacher and SalesAssistent are stored in variables of data type Employee. The method getName, which is an instance method of Employee, is then called on each instance to output the names of the two employees:

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.getName() );

}

}

Click the **Run** button to verify that the program produces the expected results.

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() );

}

}

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() );

}

}

Returning to the [database](https://www.eimacs.com/eimacs/mainpage?epid=E2203091642&cid=162149#NbhdDatabase) of people from your neighborhood, it should be clear that there are certain actions that you might be interested in performing on every data entry in the database. For example, it's likely that at some point you would need to retrieve everyone's name. With that kind of task in mind, it makes sense to store the data entries in an ArrayList. As you will discover from working on the next exercise, the ArrayList structure is admirably suited to the performance of iterative procedures where the same thing is done to each element of the ArrayList.

By storing the employee data entries in an ArrayList, we make it fairly easy to manipulate the data, at least when it comes to executing methods (such as getName) of the Employee class. In fact, [Exercise 129](https://www.eimacs.com/eimacs/mainpage?epid=E203650888&cid=162149#Exe116) was easy to solve precisely because getName is a method of the Employee class. It is not so simple, however, if we want to call the methods of classes that are lower down the Employee class hierarchy. Once the instances are stored in the ArrayList, they are treated solely as Employee objects. In a sense, their true nature is hidden. It would clearly be very useful if we had a programmatic way to identify the actual class of each element.

Imagine, for example, that we wish to obtain the secure codes of all the instances of SalesAssistant that are stored in the ArrayList. In the absence of a way to selectively skip over Employees that are not instances of SalesAssistant, as our first attempt we might simply try casting each Employee to a SalesAssistant and then calling the getSecureCode method. Run this code and see what happens.

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 )

{

SalesAssistant a = (SalesAssistant)e;

System.out.println( a.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=E2085044165&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() );

}

}

  }   
}

### Part 2

Use the Rectangle and Box classes. Observe Box is a subclass of Rectangle; therefore, an **is-a** relationship exists. That is to say, a Box is a Rectangle.

In Java, a method will accept as a parameter any object that is defined a subclass of the parameter's type.

For Rectangle and Box, if we write a method for Rectangle, it can be used for Box, as well. So we can write one method instead of two! If we wanted to print out the length and width of both a rectangle and a box we can with just one method.

The only cautionary note is that a polymorphic method cannot refer to any methods that the passed objects do not have in common. For example, we cannot reference the height of Box, since a rectangle does not have height!

* Create a project called 09.03 Polymorphism in the Module 09 Lessons folder.
* Download the following Java files to the newly-created folder.
  + [Rectangle2.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson03/docs/13_03b/Rectangle2.java)
  + [Box2.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson03/docs/13_03b/Box2.java)
  + [ShapesTester2.java](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson03/docs/13_03b/ShapesTester2.java)
* Open the [09.03 Virtual Lecture Notes](https://lti.flvsgl.com/flvs-cat-content/tou2esd2p8kdjf7ptn304rp3p7/flvs-cat-session/apcomputersciencea_v20/module09/lesson03/pop/13_03b/13_03b_popa.htm).
* The table below highlights the most important aspects of the work you have done since the last test. If you would like to review the text that relates to any of the listed items, click the corresponding green cell of the table. (It will turn pink as your mouse passes over it.)

|  |  |
| --- | --- |
| Classes | |
| Class  Review» | A class, each of whose instances represents the class of an object. The getClass method of the Object class returns a Class object. |
| Object  Review» | A class that is the *direct superclass* of every class that is not defined explicitly as the *direct subclass* of another. |
| Vocabulary | |
| Class hierarchy  Review» | The collection formed by a class and all its *subclasses*. |
| Derived  Review» | A class C is *derived* from a class D if D is the *direct superclass* of C. |
| Extend  Review» | A class C *extends* a class D if D is the *direct superclass* of C. |
| Inherit  Review» | A class *inherits* from all of its *superclasses* all their instance methods and other characteristics (but not their constructors) that are not private. The class has access to all these aspects of its superclasses without having to repeat their definitions or declarations in the class definition. |
| Polymorphism  Review» | The practice of declaring a variable to have a particular data type C and initializing it to an instance of a subclass of C. This has the effect of causing the variable to store an instance of the subclass while being treated within the program as if it stored an instance of the superclass. |
| Subclass  Review» | A class C is a *direct subclass* of a class D if and only if the definition of C includes the code "extends D". C is a *subclass* of D if and only if it is either a direct subclass of D or there is a chain of direct subclass relationships that begins with C and ends with D. |
| Superclass  Review» | A class C is the *direct superclass* of a class D if and only if the definition of D includes the code "extends C". C is a *superclass* of D if and only if it is either a direct superclass of D or there is a chain of direct superclass relationships that begins with C and ends with D. |
| Notation | |
| extends  Review» | The keyword that indicates that the class named to its left is a *direct subclass* of the class named to its right. |
| super  Review» | The keyword which, when applied to arguments, passes those arguments on to the constructor of the current class's *direct superclass*. When used in the definition of a class constructor, any super expression must be the first expression in the body of the constructor definition. |
| Methods | |
| getClass  Review» | An instance method of the Object class. When called on an Object object, returns an instance of the Class class that represents the object's class. |
| getName  Review» | An instance method of the Class class. When called on a Class object, returns a String that names the class represented by the object. |
| toString  Review» | An instance method of the Object class. When called on an Object object, returns a String that represents the object. |
| Errors | |
| Cannot resolve symbol  Review» | The error that results when, using *polymorphism*, a variable that is declared to have data type C and initialized to an instance of a *subclass* of C is subsequently referenced in the invocation of a method that is defined in the subclass but not in C. To avoid this error, cast the object to the subclass before invoking the method. |
| Class cast exception  Review» | The error that results when an attempt is made to cast an object to a class that is neither its actual class nor one of the superclasses of that class. |