**Chapter IX**

**Focus on OOP,**

**Class Interaction with Inheritance**

**Chapter IX Topics**

9.1 Introduction

9.2 Is-A and Has-A Class Interactions

9.3 GridWorld Inheritance Observations

9.4 Accessing Class Attributes

9.5 Inheritance Constructor Issues

9.6 super Calling a SuperClass Method

9.7 Umbrella Classes

9.8 Summary

**9.1 Introduction**

Chapter IV introduced Object Oriented Programming. We took small steps and started by explaining how to use class methods and object methods. In some later chapters you learned how to declare your own classes and how to create your own class methods and object methods. These early chapters set the stage so that you would be ready to understand Object Oriented Programming more formally. Encapsulation was explained in the last chapter and started the first OOP focus. Encapsulation is the OOP feature whereby the data of an object and all the methods, which access the data, are placed in the same container. Understanding encapsulation is very important in learning OOP, but it is only the first step.

The previous chapter taught you how to create a class and how to create the attributes and methods that are members of the class. You also observed that a second testing class was used to see if your new class works properly. Besides learning how to test a class, nothing was shown about how multiple classes are used together. How do they connect? How do they interact? This is the second OOP focus that will be discussed in this chapter. *Class interaction* is further divided into *inheritance* and *composition*. We will investigate *inheritance* first.

You have already seen and used inheritance in many previous program examples. In chapter IV you were introduced to using graphics in Java and creating applets for web pages. In each one of your applet programs the class heading was followed with the words **extends Applet**. Little was mentioned about those words, but those two little words indicate that the current class will be using features that have already been declared in the **Applet** class.

A programming language is initially so complex that it is necessary to use various language features that are not immediately explained. Explaining everything may be very thorough, but such an approach will be very intimidating and certainly overwhelming. Now you are ready to explore inheritance.

**9.2 Is-A and Has-A Class Interactions**

*Object Oriented Programming* is very popular for a good reason. There are so many features that make a program better designed, more reliable, easier to test and simpler to update than previous, non-OOP programs used to be. You have already seen a fair amount of encapsulation and learned the benefits of placing methods and data in the same module, along with constructors. Now let us see what class interaction with inheritance can do for the programmer. Do not be alarmed if you do not instantly see the great benefits of all this OOP stuff. OOP is very powerful, but it takes time to reach a comfort level with this programming approach.

Imagine that you are very creative in designing custom vans. You know just how to replace regular seats with comfortable “captain” seats, add attractive paneling, provide lots of lights, install an entertainment system, add a small kitchen and include many other goodies to make a simple van become a terrific vehicle for long road trips. Now are you interested in putting together the basic chassis, the frame, the doors, the engine, the transmission, the drive shaft, the air-conditioning and all the other details that a van requires? No, you do not care. Engines, transmissions and air-conditioning are simply not your concerns. Your only concern is designing a comfortable, custom van. The solution is to go out and get a fully functional basic van. Now with that van you can get to work. This means that any one of your custom vans first and foremost *is-a-van*. This is a very important concept and it is called the *is-a*relationship.

Geometry does a nice job explaining this concept. Geometry starts with very elementary concepts like points and lines. Then theorems and definitions are created that continually build upon previous knowledge. Everywhere the assumption is that previous definitions may be used. For example, the definition of a rectangle does not need to start from scratch. You can state that *a rectangle is a parallelogram with four right angles*. The assumption is that the definition of a parallelogram is known. This process can continue by stating that *a square is a rectangle with four equal sides*. All through Geometry you will see the *is-a* statements used. This style of definition is very efficient. You establish a logical sequence of information and provide definitions that are based on a clear understanding of previous elements in the sequence. Basically, you stop reinventing the wheel on a regular basis.

Let us switch to computer science and see how this inheritance business might apply. You are a number-one-awesome programmer and you have just finished creating a **Window** class. Now this **Window** class is a beauty, because you can now display a window anywhere on the monitor. The window can minimize, maximize and you can drag the window to any location. You are rightfully very proud of your new **Window** class.

Now you want to expand the capability of your **Window** class and use it to enter text. This adds a whole new dimension of capabilities to your humble **Window** class. Now, your new class - let us call it **TextWindow** - needs to still do everything that the plain old vanilla **Window** class did and more. It is possible to start from scratch, but that would be as efficient as our van converter person who first builds his own vans from scratch before customizing them for special needs. The secret is to start with the **Window** class and use all its capabilities for a new class called **TextWindow**. In computer science we say that **TextWindow** inherits from **Window**. Such an approach saves time and it adds tons of reliability. Just imagine that you are using an existing class that withstood the test of time. The class has been used over and over again, and all the bugs are corrected. Everybody is happy with your nifty **Window** class. Now do not touch or alter **Window**. Start a new class, but do it in such a way that it is capable of using all the nicely tested features that are available with the existing **Window** class. In a nutshell that is inheritance.

|  |
| --- |
| **Inheritance Definition** |
| **Inheritance** is the process of using features (both attributes and methods) from an existing class. The existing class is called the **superclass** and the new class, which inherits the superclass features, is called the **subclass**. |

Students frequently make a common mistake with inheritance. They confuse the *"is-a"* relationship with the *"has-a"* relationship. Both relationships involve class interaction and both are very useful in computer science, but they are very different. Go back to the van example. Our van custom converter started with a fully functional van and customized it. The finished product “*is-a”* van.

Now when the van was first assembled it was a different story. Many different assembly lines converged to create the van. The van “*has-an”* engine, and it “*has-a”* transmission, and it “*has”* doors. The van is composed of many parts, but it is incorrect to state *a van “is-a”* door.

The confusion starts because in both cases something new is created with existing components. However there is a big difference between the two creations. With composition a new item is created that is composed of many existing parts. This is the case of assembling a van with a frame, wheels, doors, seats, engine, and all the other van parts.

Now inheritance also uses something that already exists, but the whole item is used and then enhanced. A square is a special type of rectangle. A tiger is one particular type of cat. An off-road truck is first a truck. Take the example of the off-road truck. You can buy a regular truck. Now you put in special shocks that provide greater clearance. You also add four-wheel drive and a locking differential feature. You add special tires that will do well in mud and snow and your original truck is now an off-road truck. However, *it is still a truck.*

|  |
| --- |
| “Is-A” and “Has-A” |
| The creation of new classes with the help of existing classes makes an important distinction between two approaches.  An *"is-a"*relationship declares a new class as a special  “new-and-improved” case of an existing class. In Geometry, a parallelogram *"is-a"*quadrilateral with special properties.  A *“has-a”*relationship declares a new class composed of an existing class or classes. A line has points, a square has lines, and a cube has squares.  A truck *"is-a"*vehicle, but it *"has-an"*engine.  In computer science an *"is-a"*relationship involves *class interaction* that is called *inheritance* and a *"has-a"*relationship involves *class interaction* that is called *composition*. |

**9.3 GridWorld Inheritance Observations**

The College Board provided *GridWorld Case Study*, created by *Cay Horstmann*, is an excellent example of program design. It is an especially good model for demonstrating class interaction with inheritance. Shortly, there will be many program examples that will explain the precise syntax and logic that is required to use inheritance properly. First, there will be four short projects using GridWorld to demonstrate inheritance concepts visually.

**GridWorld Lab Experiment Java0901**

Inside the APCS-09 Learning Unit is a **Programs09** folder. This folder contains the four folders for the GridWorld projects. Start by creating a project for the **Java0901** folder. This project contains a new **Spider** class, which is shown in figure 9.1, and presents a class heading with an empty class body. Each project will show a different stage of the **Spider** class and it also requires another file to test each stage. **TestJava0901.java**, shown in figure 9.2, displays the program that is meant to test the **Spider** class.

**Figure 9.1**

|  |
| --- |
| /\*  \* AP(r) Computer Science GridWorld Case Study:  \* Copyright(c) 2005-2006 Cay S. Horstmann (http://horstmann.com)  \*  \* 04-12-12 by Leon Schram  \* Used for Java0901 project  \*/  public class Spider  {  } |

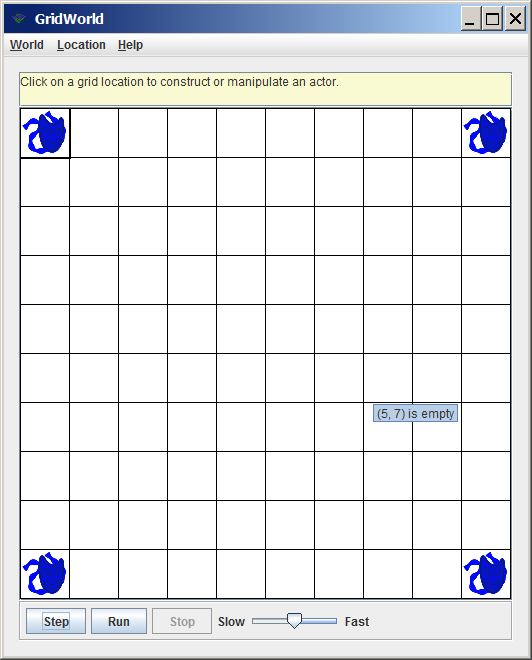
**Figure 9.2**

|  |
| --- |
| /\*  \* AP(r) Computer Science GridWorld Case Study:  \* Copyright(c) 2005-2006 Cay S. Horstmann (http://horstmann.com)  \*  \* 04-12-12 by Leon Schram  \* Used for Java0901 project  \*/  import info.gridworld.actor.ActorWorld;  import info.gridworld.actor.Actor;  import info.gridworld.grid.Location;  public class TestJava0901  {  public static void main(String[] args)  {  ActorWorld world = new ActorWorld();  Actor actor1 = new Actor();  Actor actor2 = new Actor();  Actor actor3 = new Actor();  Actor actor4 = new Actor();  world.add(new Location(0,0),actor1);  world.add(new Location(0,9),actor2);  world.add(new Location(9,0),actor3);  world.add(new Location(9,9),actor4);  world.show();  }  } |

Figure 9.3 shows the project output and you will see four **Actor** objects. There is no evidence of the new **Spider** class. This may seem quite logical, because there are no program statements inside the **Spider** class.

The empty class body is not as much of a problem as the fact that there is no object of the **Spider** class that is added to the grid. Look at figure 9.2 and you will see that four **Actor** objects are constructed, and added to the grid, but there is nothing connecting a **Spider** object anywhere.

**Figure 9.3**



The first project provided no information about inheritance. It did show two separate classes. The project contained a **Spider** class and a **TestJava0901** class. At least those are the only two classes mentioned and directly visible in your project. If you look in figure 9.2 you will see three import statements, which indicate that the **ActorWorld**, **Actor** and **Location** classes are used for this program. Additionally, the creation of any GridWorld project involves the inclusion of the **gridworld.jar** file, which contains many classes that handle lots of the *hidden, behind the scene* GridWorld operations, such as graphics display interaction, that are not meant to be seen by students working with GridWorld files. Consider the previous **Spider** class. The class body was empty, but the heading did not provide any interaction with any other class. It is the class heading that states very clearly that a new class exists and depends upon resources provided from an existing class by using the **extends** Java keyword.

**GridWorld Lab Experiment Java0902**

Create the next **Java0902** GridWorld project and look at the next stage of the **Spider** class in figure 9.4. There are two changes. The **Spider** class body is still empty, but the heading has added **extends Actor**. These two little words imply that the new **Spider** class is a *subclass* of the **Actor** class. The second change is an import statement that includes the **Actor** class.

The second project also brings a change in the testing class. Look at Figure 9.5 and you will see that two objects of the **Spider** class are constructed and added to the grid. Now do not get confused. The **Spider** objects are not named like the four **Actor** objects. They do exist and this style of construction is known as constructing *anonymous objects*.

**Figure 9.4**

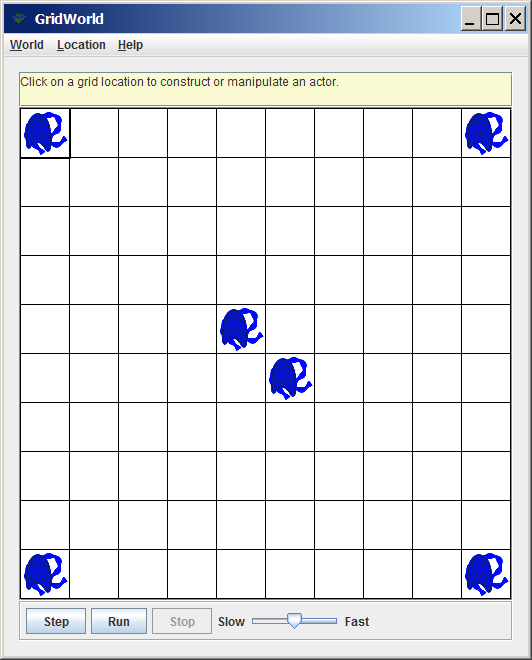
|  |
| --- |
| Wha/\*  \* AP(r) Computer Science GridWorld Case Study:  \* Copyright(c) 2005-2006 Cay S. Horstmann (http://horstmann.com)  \*  \* 04-12-12 by Leon Schram  \* Used for Java0902 project  \*/  import info.gridworld.actor.Actor;  public class Spider extends Actor  {  } |

**Figure 9.5**

|  |
| --- |
| /\*  \* AP(r) Computer Science GridWorld Case Study:  \* Copyright(c) 2005-2006 Cay S. Horstmann (http://horstmann.com)  \*  \* 04-12-12 by Leon Schram  \* Used for Java0902 project  \*/  import info.gridworld.actor.ActorWorld;  import info.gridworld.actor.Actor;  import info.gridworld.grid.Location;  public class TestJava0902  {  public static void main(String[] args)  {  ActorWorld world = new ActorWorld();  Actor actor1 = new Actor();  Actor actor2 = new Actor();  Actor actor3 = new Actor();  Actor actor4 = new Actor();  world.add(new Location(0,0),actor1);  world.add(new Location(0,9),actor2);  world.add(new Location(9,0),actor3);  world.add(new Location(9,9),actor4);  world.add(new Location(4,4),new Spider());  world.add(new Location(5,5),new Spider());  world.show();  }  } |

Look at the execution of the second project, in figure 9.6. There used to be four objects and now there are six objects. The objects are intentionally placed at specified locations. The four **Actor** objects are placed at the four corners and the two new **Spider** objects are located in the center of the grid.

**Figure 9.6**



The truth is that you do not really see two **Spider** objects. You see six **Actor** objects. What you see is very logical. It is a consequence of the empty **Spider** bodies. The purpose of inheritance is to reuse existing classes so that you can benefit from methods that already have been tested. The **new** class can then be written to *re-define* one or more existing methods or possibly *newly-define* one or more methods.

|  |
| --- |
| The Reason for Using Inheritance |
| Inheritance adds reliability to a program by using existing classes with methods that have been created and tested thoroughly.  A new class is created, and at a minimum, needs to re-define or newly-define one method for the new class to have practical value. |
| Any subclass that **extends** a super class with an empty class body, such as the second **Spider** class stage, will behave 100 percent like its super class. |

**GridWorld Lab Experiment Java0903**

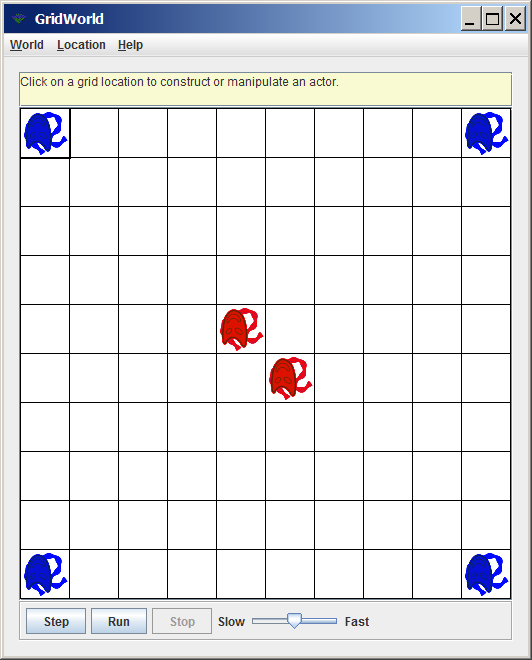
The third project will finally create a **Spider** class with more than a heading and an empty body. Figure 9.7 shows that the constructor is re-defined. It is not much, but it is something and it means that every **Spider** object will be constructed as a red object. The testing program is not altered and will not be redisplayed. You can go back to figure 9.5, if you want to check the code.

**Figure 9.7**

|  |
| --- |
| /\*  \* AP(r) Computer Science GridWorld Case Study:  \* Copyright(c) 2005-2006 Cay S. Horstmann (http://horstmann.com)  \*  \* 04-12-12 by Leon Schram  \* Used in Java0903 project  \*/  import info.gridworld.actor.Actor;  import java.awt.Color;  public class Spider extends Actor  {  public Spider()  {  setColor(Color.red);  }  } |

The output, shown in figure 9.8, was stopped intentionally when the **Actor** objects were upside down. You will note that both the blue and red objects are upside down. All six objects act in the same way, even though some objects are a different color. This should makes sense since only the constructor is re-defined. When all of the objects are called to **act**, every object follows the instructions of its **act** method. In the case of **Spider** objects, there are no **act** methods, which results in calling the **act** method of the **Actor** superclass.

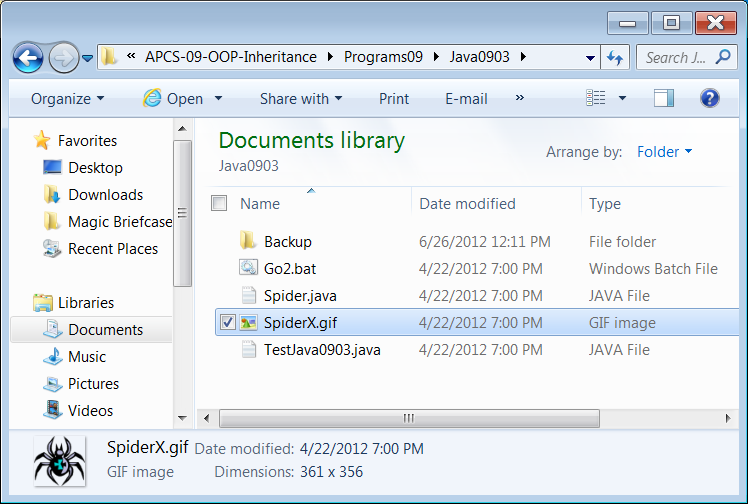
**Figure 9.8**



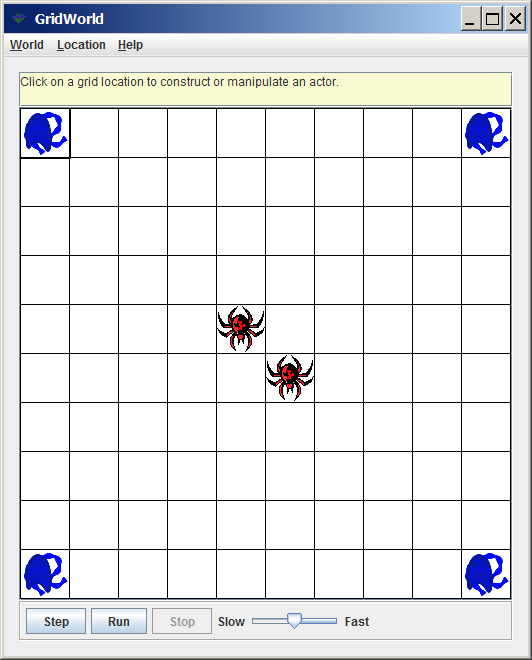
The mask of the **Actor** objects is appropriate for an **Actor** object, but it is quite strange for a **Spider** object. Once again, nothing was done except create a new constructor. Some instructions, hidden in the **gridworld.jar** file associate an object with a graphics file of the same name. The mask comes from the **Actor.gif** file. If a new class is created without an available graphics file by the same name, the icon displayed is the one of its superclass.

Look in figure 9.9. It shows the files of the **Java0903** project. It includes a graphics file, called **SpiderX.gif**. Rename that file **Spider.gif**. Run the project again and observe in figure 9.10 that this time you will get two **Spider** objects that look like spiders. Once again, the execution is stopped when the **Actor** objects are upside down**.** The two **Spider** objects are red, and they look like spiders, but they still act exactly as their **Actor** superclass parents, because method **act** is not re-defined in the **Spider** subclass.

**Figure 9.9**



**Figure 9.10**



**GridWorld Lab Experiment Java0904**

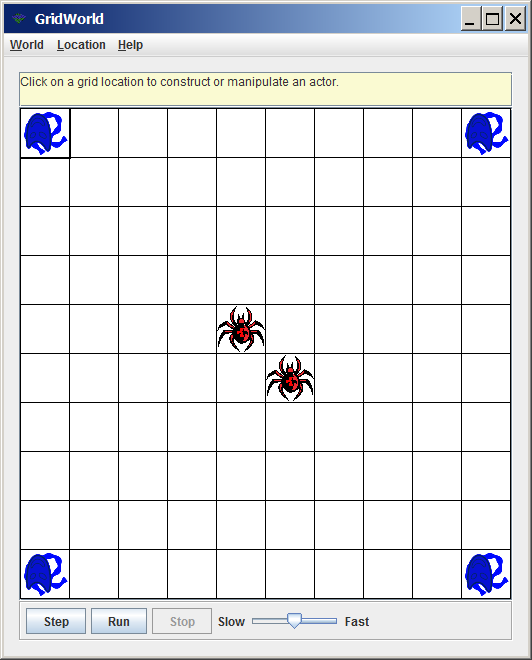
You must be tired of spiders that only act like actors. The fourth project introduces the **LazySpider**, which is a subclass of the **Spider** class and re-defines the **act** method. Do not get overly excited. Figure 9.11 shows that the **act** method is quite empty, which results in the same behavior as a **Rock** object.

**Figure 9.11**

|  |
| --- |
| /\*  \* AP(r) Computer Science GridWorld Case Study:  \* Copyright(c) 2005-2006 Cay S. Horstmann (http://horstmann.com)  \*  \* 04-12-12 by Leon Schram  \* Used in Java0904 project  \*/  public class LazySpider extends Spider  {  public void act()  {  }  } |

The execution shown in figure 9.12 shows that the **Actor** objects are upside down, which happens in their behavior cycle. At the same time the **LazySpider** objects are still facing north. You will note that **LazySpider** objects are lazy and sit still, but they are red. There is no constructor defined in the **LazySpider** class. This means that Java goes up one level to the superclass and uses the constructor at that level, which makes any **Spider** or **LazySpider** object red.

**Figure 9.12**



There is a lot more to be said about inheritance in the remaining sections. This GridWorld introduction is meant to show some fundamental concepts. You must realize that inheritance is used for reliability, and yes also for efficiency by using existing code. You must realize that any existing code is not touched. There are too many opportunities to make unwanted changes in perfectly good code.

The reliable OOP approach to use existing classes is to leave them complete alone and then create a new class. This new class becomes a subclass of the existing class by using **extends** in the class heading of the subclass.

The subclass then is written so that the only methods in the subclass are re-definitions of superclass methods or they are newly-defined methods.

|  |
| --- |
| Subclass Methods |
| Never alter a well-designed, and tested, existing class.  Write a new subclass class to use the methods of the existing class and create new methods in your new class. |
| Write methods in the subclass that are *re*-*definitions* of the existing superclass methods or write totally *new*-*definitions.* |

**9.4 Accessing Class Attributes**

The GridWorld inheritance programs were meant to introduce inheritance visually using the existing **Actor** class. We continue now with a series of simple programs to clarify the syntax of inheritance and also to demonstrate how a subclass has access to members of a superclass. Program **Java0905.java**, in figure 9.13 shows a program that starts with a **Person** class. The **Person** class is not very impressive since it has only one attribute, **age**, and only one method, **getAge**. A second class, **Student**, inherits the **Person** capabilities. This means that **Student** *is-a* **Person**. The **Student** class is equally skimpy and has one attribute, **grade**, and one method, **getGrade**.

**Figure 9.13**

|  |
| --- |
| // Java0905.java  // This program demonstrates fundamental inheritance with <extends>.  // There are no constructors yet, which results in Java handling the  // construction and assigning default values to the attributes.  public class Java0905  {  public static void main(String args[])  {  System.out.println("\nJAVA0905\n");  Student tom = new Student();  System.out.println("tom's age is " + tom.getAge());  System.out.println("tom's grade is " + tom.getGrade());  System.out.println();  }  }  class Person  {  private int age;  public int getAge()  {  return age;  }  }  class Student extends Person  {  private int grade;  public int getGrade()  {  return grade;  }  } |

**Figure 9.13 Continued**

|  |
| --- |
| Java0905.java Output JAVA0905  Person's age is 0  Student's grade is 0 |

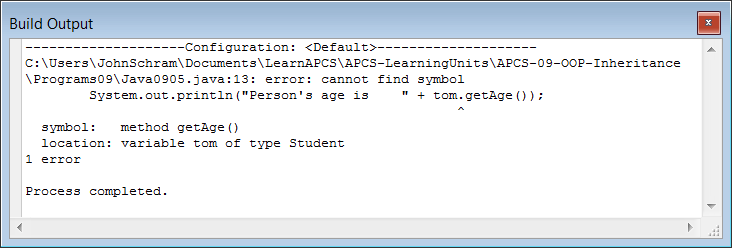
Program **Java0905.java** shows the required syntax to implement inheritance. **Person** is the **superclass**, and **Student** is the **subclass**, which inherits from **Person**. The declaration of the **Student** class starts with the customary statement

**class Student**, and then it is followed with **extends Person**. The keyword **extends** is the secret of inheritance in Java syntax. With **extends** the Java compiler is told that the new subclass has access to the features of the superclass, which is mentioned after the **extends** statement.

Do you see proof of inheritance? Take a look at the main method. Only one object is constructed. It is **tom**, which is a **Student** object. Now observe that the **tom** object calls both the **getAge** and **getGrade** methods. This is great but **getAge** is not a **Student** method. It is possible to access this method precisely because **Student** is a subclass of superclass **Person**.

Try a quick experiment. Delete the two words **extends Person** and recompile the program. Figure 9.14 shows Java's response. The **Person** class and the **Student** class no longer have any class interaction. It is now impossible to call a **Person** method by using the **Student** object **tom**.

**Figure 9.14**



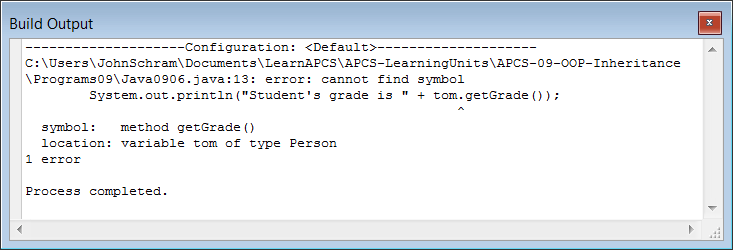
We have seen proof that **extends Person** gives access to a method in the **Person** with a **Student** object. Does this work in both directions? Can a superclass object access a method in a subclass? Program **Java0906.java**, in figure 9.15 checks that possibility.

**Figure 9.15**

|  |
| --- |
| // Java0906.java  // This program reverses the access between the classes.  // A person object now tries to access a subclass method.  public class Java0906  {  public static void main(String args[])  {  System.out.println("\nJAVA0906\n");  Person tom = new Person();  System.out.println("Person's age is " + tom.getAge());  System.out.println("Student's grade is " + tom.getGrade());  System.out.println();  }  }  class Person  {  private int age;  public int getAge()  {  return age;  }  }  class Student extends Person  {  private int grade;  public int getGrade()  {  return grade;  }  } |

This experiment falls into the category of nice try. Figure 9.16 shows that Java cannot digest this type of interaction. Now think about this. Class interaction is meant to take advantage of using established classes. We say *class interaction*, but this is not meant to be in both directions. One class already exists and supposedly is well written and thoroughly tested. This existing class now becomes the superclass for a new class that takes advantage of the available members of the superclass and defines some of its own methods.

**Figure 9.16**



|  |
| --- |
| Access with Inheritance |
| When two classes have an inheritance interaction then access is only possible from the subclass to the superclass members.  It is not possible to access subclass members from the superclass. |

Class access has been limited to methods so far. What happens when a subclass tries to access the data attributes of a super class? Program **Java0907.java**, in figure 9.17, answers that question. In this program is a **Student** method, called **showData**. It is written to display the data attribute values of the **Person** superclass, as well as the **Student** subclass.

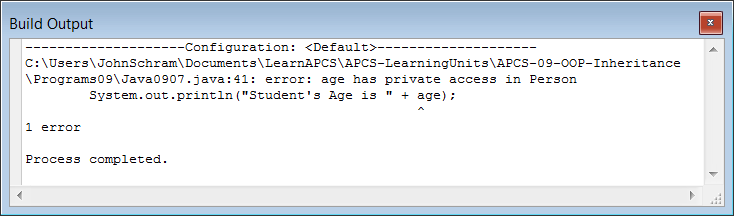
You have seen how *get* methods are designed to return the values of data attributes. This is not the intention here. The **showData** method goes straight to the data fields and intents to display them.

The experiment is quickly stopped. Java makes it quite clear that access to **private** data is not allowed, even from a proper subclass. **Private** means **private** and only local members have direct access. Figure 9.18 displays the stern warning by Java that you are very confused. Notice that the error applies to **age** only. Access to the **grade** data field is permitted. **grade** is a local attribute of the **Student** class.

**Figure 9.17**

|  |
| --- |
| // Java0907.java  // This program shows that the subclass does not have access to the private data  // of the superclass. This program will not compile.  public class Java0907  {  public static void main(String args[])  {  System.out.println("\nJAVA0907\n");  Student tom = new Student();  tom.showData();  System.out.println();  }  }  class Person  {  private int age;  public Person()  {  age = 17;  }  }  class Student extends Person  {  private int grade;  public Student()  {  grade = 12;  }  public void showData()  {  System.out.println("Student's Grade is " + grade);  System.out.println("Student's Age is " + age);  }  } |

**Figure 9.18**



Now we have made a big deal in past chapters that data should only be accessed by special methods that access the data correctly. Basically, this may mean that you are not very disturbed about an inability to access data by a subclass. On the other hand, you could argue that a subclass is supposed to inherit all the features of the superclass, and does that not include the data of the superclass? This is a good point and Java's inheritance allows you to have your cake and eat it too.

It is possible to access data of the superclass, but that is not possible if the superclass data is declared **private**. The declaration needs to change to **protected**. Program **Java0908.java**, in figure 9.19, is almost identical to the previous program. The only change is that the data is now declared **protected**, not **private**. This time the program compiles just fine, and the subclass shows access to the data of the superclass.

**Figure 9.19**

|  |
| --- |
| // Java0908.java  // This program changes private member data to "protected" data.  // The <Student> class can now access data from the <Person> class.  public class Java0908  {  public static void main(String args[])  {  System.out.println("\nJAVA0908\n");  Student tom = new Student();  tom.showData();  System.out.println();  }  }  class Person  {  **protected** int age;  public Person()  {  age = 17;  }  }  class Student extends Person  {  protected int grade;  public Student()  {  grade = 12;  }  public void showData()  {  System.out.println("Student's Grade is " + grade);  System.out.println("Student's Age is " + age);  }  } |

**Figure 9.19 Continued**

|  |
| --- |
| Java0908.java Output JAVA0908  Student's Grade is 12  Student's Age is 17 |

Life used to be simple. **Public** items in a class can be accessed outside the class and **private** items can only be accessed inside the class. Now **protected** adds confusion to the simple rules. What does **protected** mean? Protected is an interesting hybrid. With **protected** you can access from outside the class, but only if you are a subclass. Any other class has no access. This means that **protected**, to a subclass behaves like **public**, but to other classes it behaves like **private**.

|  |
| --- |
| Public, Private & Protected |
| Attributes & methods declared public can be accessed by methods declared both outside and inside the class.Attributes & methods declared private can only be accessed by methods declared inside the class. Attributes & methods declared **protected** can be accessed by methods declared inside the class or subclass. |

**9.5 Inheritance Constructor Issues**

Constructors play an important role in any class and you have learned that it is common that a class has multiple constructors. When classes interact you will find that the constructors also interact. Understanding the special requirements of constructors when inheritance is used insures that all classes operate as expected. Program **Java0909.java**, in figure 9.20, defines the **Person** and **Student** constructors with output statements.

**Figure 9.20**

|  |
| --- |
| // Java0909.java  // This program adds output in the constructors to the <Person> and <Student> classes.  // Note how the <Person> constructor is called, even though there does  // not appear to be a <Person> object instantiated.  public class Java0909  {  public static void main(String args[])  {  System.out.println("\nJAVA0909\n");  Student tom = new Student();  System.out.println("tom's age is " + tom.getAge());  System.out.println("tom's grade is " + tom.getGrade());  System.out.println();  }  }  class Person  {  private int age;  public Person()  {  System.out.println("Person Constructor");  age = 17;  }  public int getAge()  {  return age;  }  }  class Student extends Person  {  private int grade;  public Student()  {  System.out.println("Student Constructor");  grade = 12;  }  public int getGrade()  {  return grade;  }  } |

|  |
| --- |
| Java0909.java Output JAVA0909  Person Constructor  Student Constructor  tom's age is 17  tom's grade is 12 |

There is a **Student** object, called **tom**, constructed. The single **new** operator calls the **Student()** constructor and it is reasonable in the output, that you see Student Constructor. What about the Person Constructor output?

There is not a **new Person()** statement in the program and yet the evidence displays that somehow the **Person** constructor is called.

Go back to the van analogy. If a custom van *is-a* van, does it not start with a plain-old-vanilla van first? In other words, is it not required first to construct a van before you can construct a custom van? This is precisely what happened in program **Java0909.java**. One object is instantiated, but the **tom** object starts its life by first becoming a **Person** and then becoming a **Student**.

Program **Java0910.java** helps to explain exactly how Java manages to handle this business of constructing a super class object first. The program is almost identical to the previous program and generates the exact same output. The only difference is shown in figure 9.21 where you can compare both of the two **Student** constructors. The second program uses a call to a **super** method, which calls the superclass constructor. A programmer can do this, as it was done in **Java0910.java**, but Java is happy to do this for you automatically. This is done, even if you do not provide the **super** statement in the subclass.

**Figure 9.21**

|  |
| --- |
| **Java0909 Student constructor**  **public Student()**  **{**  **System.out.println("Student Constructor");**  **grade = 12;**  **}** |
| **Java0910 Student constructor**  **public Student()**  **{**  **super();** // must be first statement in the constructor  **System.out.println("Student Constructor");**  **grade = 12;**  **}** |

|  |
| --- |
| Inheritance and Constructors Calls |
| When an object of a *subclass* is instantiated, the constructor of the *superclass*is executed first, followed by completing the execution of the *subclass*constructor. |
| An invisible - to the programmer - call is made by Java to the **super** method, which generates a call to the superclass constructor. This statement can be written in the subclass constructor with the same results, *but it is not required*. |

#### Passing Information to SuperClass Constructors

You have seen various program examples involving both inheritance and constructors. In each case, evidence showed that the constructor of the superclass is executed first followed by executing the constructor of the subclass. In each example, the superclass constructor used was a *no-parameter, default* constructor. This begs the question what might happen if *parameter constructors* are used, and how is information passed in such a case on to the superclass?

Program **Java0911.java**, in figure 9.22, shows the data information path. The process starts by creating a new **Student** object **tom**. The **Student** constructor is called first with parameter information **(12,17)**.

The **12** is meant to be the value for the **grade** dataattribute of the **Student** class and the **17** is meant to be the value for the **age** dataattribute of the **Person** class. The statement **new Student(12,17)** passes parameter values to the constructor method heading of **Student**. You have seen that a superclass object must be constructed first and that means that the **17** value needs to travel to the superclass constructor before anything can happen in the **Student** subclass constructor.

Note, that the very **first** statement in the subclass constructor is **super(a)**. Variable **a** (for **age**) has received a copy of the actual parameter **17** and the superclass **Person** constructor is now called with the **17** value for **age**.

After the superclass is satisfied, the **g** parameter provides the **12** value to the **grade** attribute of the **Student** class. Java is happy and all the attributes have received the proper value in the proper sequence.

**Figure 9.22**

|  |
| --- |
| // Java0911.java  // This program demonstrates how a subclass constructor passes  // parameter information to a superclass constructor.  public class Java0911  {  public static void main(String args[])  {  System.out.println("\nJAVA0911\n");  Student tom = new Student(12,17);  tom.showData();  System.out.println();  }  }  class Person  {  protected int age;  public Person(int a)  {  System.out.println("Person Parameter Constructor");  age = a;  }  public int getAge()  {  return age;  }  }  class Student extends Person  {  protected int grade;  public Student(int g, int a)  {  **super(a);** // this must be the first call  grade = g;  System.out.println("Student Parameter Constructor");  }  public int getGrade()  {  return grade;  }  public void showData()  {  System.out.println("Student's Grade is " + getGrade());  System.out.println("Student's Age is " + getAge());  }  } |

**Figure 9.22 Continued**

|  |
| --- |
| Java0911.java Output JAVA0911  Person Parameter Constructor  Student Parameter Constructor  Student's Grade is 12  Student's Age is 17 |

This parameter passing business to superclass constructors can happen at multiple levels. Program **Java0912.java**, in figure 9.23, has three classes and information needs to be provided for attributes at three different levels.

The highest superclass is **Animal**. This is followed by **Mammal**, which **extends** **Animal**. In this situation **Mammal** is the subclass of **Animal** and simultaneously takes on the role of being the superclass of the **Cat** class. This type of interaction is actually quite natural when you view your own world. Every person is a child and then many children become parents. For many years a person will be both a child and a parent.

Parameter passing in this three-level inheritance example starts at the bottom. The testing class creates a **Cat** object **tiger** and calls the **Cat** constructor with three data values.

The **Cat** constructor receives the three data values and first passes two values to the superclass **Mammal** with the statement **super(w,a)**. At the second level another **super(a)** statement is used to pass **age** to the highest super class. This process can be used a large number of inheritance levels.

**Figure 9.23**

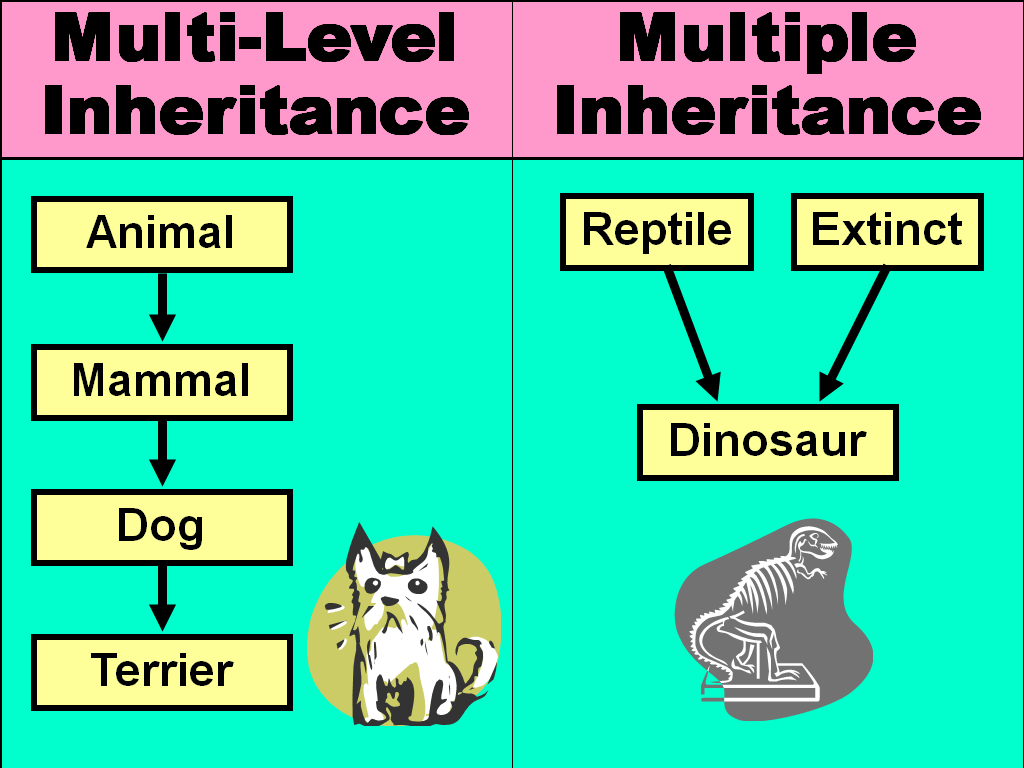
|  |
| --- |
| // Java0912.java  // This program demonstrates inheritance at three levels.  public class Java0912  {  public static void main(String args[])  {  System.out.println("\nJAVA0912\n");  Cat tiger = new Cat("Tiger",500,5);  System.out.println();  System.out.println("Animal type: " + tiger.getType());  System.out.println("Animal weight: " + tiger.getWeight());  System.out.println("Animal age: " + tiger.getAge());  System.out.println();  }  }  class Animal  {  protected int age;  public Animal(int a)  {  System.out.println("Animal Constructor Called");  age = a;  }  public int getAge()  {  return age;  }  }  class Mammal extends Animal  {  protected int weight;  public Mammal(int w, int a)  {  **super(a);**  weight = w;  System.out.println("Mammal Constructor Called");  }  public int getWeight()  {  return weight;  }  }  class Cat extends Mammal  {  protected String type;  public Cat(String t, int w, int a)  {  **super(w,a);**  type = t;  System.out.println("Cat Constructor Called");  }  public String getType()  {  return type;  }  } |

**Figure 9.23 Continued**

|  |
| --- |
| Java0912.java Output JAVA0912  Animal Constructor Called  Mammal Constructor Called  Cat Constructor Called  Animal type: Tiger  Animal weight: 500  Animal age: 5 |

When you talk about *Multi-Level Inheritance*, be careful that you say it properly. There is something else called *Multiple Inheritance* which sounds similar, but is completely different. In *Multiple Inheritance*, one subclass can have multiple super classes. The best way to explain the difference is to look at figure 9.24, below. A dinosaur is-a reptile, and a dinosaur is also extinct. Something else to keep in mind, Java allows *Multi-Level Inheritance*, but it does NOT allow *Multiple Inheritance*. Some languages, like C++, allow *multiple inheritance*.

**Figure 9.24**



**9.6 super Calling a SuperClass Method**

All the program examples have conveniently used method identifiers in the **Person** and **Student** class that are not identical. Considering the fact that a major reason for using inheritance is the *re-definition* of existing method, it should be quite common to use identical identifiers in multiple classes. How does Java handle this confusion? These questions are answered by this section starting with program **Java0913.java**, shown in figure 9.25. In that program example both the **Person** class and the **Student** class use method **getData** to return the data information of the class.

**Figure 9.25**

|  |
| --- |
| // Java0913.java  // In this program both the <Person> class and the <Student> class each have a <getData> method.  public class Java0913  {  public static void main(String args[])  {  System.out.println("\nJAVA0913\n");  Person ann = new Person();  Student tom = new Student();  System.out.println("Person getData: " + ann.getData());  System.out.println("Student getData: " + tom.getData());  System.out.println();  }  }  class Person  {  protected int age;  public Person()  {  age = 21;  }  public int getData()  {  return age;  }  }  class Student extends Person  {  protected int grade;  public Student()  {  grade = 12;  }  public int getData()  {  return grade;  }  } |

|  |
| --- |
| Java0913.java Output JAVA0913  Person getData: 21  Student getData: 12 |

I am sure that you are not surprised by the program output in figure 9.25. There are two different methods with the same name, but the values returned are the values we want. This is not too tricky for Java. You call an **ann Person** object and you get **ann**'s **getData** method. Then you continue and call a **tom getData** object and you get **tom**'s **getData** method.

There was no confusion in **Java0913.java**. Java executes the method that is defined by the object that you are using. Fair enough. Now what happens if you have a **Student** object and you do not want to call the **Student getData**, you want the **Person getData** method. Program **Java0914.java**, in figure 9.26 actually calls **getData** twice with the intention to get two different values. This requirement is handled by using **super** in front of **getData** to indicate the superclass method.

**Figure 9.26**

|  |
| --- |
| // Java0914.java  // This program demonstrates that it is possible to distinguish between  // two methods with the same identifier using <super>.  public class Java0914  {  public static void main(String args[])  {  System.out.println("\nJAVA0914\n");  Student tom = new Student(12,17);  tom.showData();  System.out.println();  }  }  class Person  {  protected int age;  public Person(int a)  {  System.out.println("Person Parameter Constructor");  age = a;  }  public int getData()  {  return age;  }  }  class Student extends Person  {  protected int grade;  public Student(int g, int a)  {  super(a);  grade = g;  System.out.println("Student Parameter Constructor");  }  public int getData()  {  return grade;  }  public void showData()  {  System.out.println("Student's Grade is " + getData());  System.out.println("Student's Age is " + **super**.getData());  }  } |

**Figure 9.26 Continued**

|  |
| --- |
| Java0914.java Output JAVA0914  Person Parameter Constructor  Student Parameter Constructor  Student's Grade is 17  Student's Age is 12 |

|  |
| --- |
| Using super |
| The keyword **super** used as the first statement in a constructor passes information to the super class constructor, like **super(a);**  The same keyword **super** used in front of a method indicates that a method of the superclass needs to be called, like **super.getData();**  Information can be passed up to multiple inheritance levels, but it can only be passed one level at one time. |

**9.7 Umbrella Classes**

Start by looking at program Java0915.java, in figure 9.28. Specifically, look at the **main** method and observe how the objects are constructed. You are used to seeing the same identifier at the start and end of an object definition, like

**Bug barry = new Bug();**

What you see in this program example is quite different. There are three definitions and all three use the same class identifier, **Animal**, but then each object uses a different constructor, **Cat**, **Bird** and **Fish**. This type of Java code is quite common and you will see it used in the *GridWorld Case Study*. Right now in this program there is little benefit. The program executes in the exact same manner if the declarations in figure 9.27 were used.

**Figure 9.27**

|  |
| --- |
| **Cat tiger = new Cat("Tiger");**  **Bird eagle = new Bird("Eagle");**  **Fish shark = new Fish("Shark");** |

**Figure 9.28**

|  |
| --- |
| // Java0915.java  // This program demonstrates that it is possible to use the super class identifier  // <Animal> to declare each subclass object.  public class Java0915  {  public static void main(String args[])  {  System.out.println("\nJAVA0915\n");  **Animal** tiger = new **Cat**("Tiger");  **Animal** eagle = new **Bird**("Eagle");  **Animal** shark = new **Fish**("Shark");  }  }  class Animal  {  public Animal()  {  System.out.println("Animal constructor called");  }  }  class Cat extends Animal  {  protected String catType;  public Cat(String ct)  {  System.out.println("Cat constructor called");  catType = ct;  }  }  class Bird extends Animal  {  protected String birdType;  public Bird(String bt)  {  System.out.println("Bird constructor called");  birdType = bt;  }  }  class Fish extends Animal  {  protected String fishType;  public Fish(String ft)  {  System.out.println("Fish constructor called");  fishType = ft;  }  } |

**Figure 9.28 Continued**

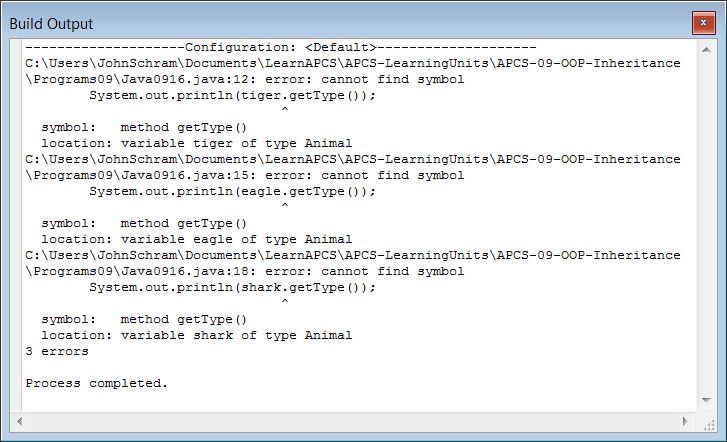
|  |
| --- |
| Java0915.java Output **JAVA0915**  Animal constructor called  Cat constructor called  Animal constructor called  Bird constructor called  Animal constructor called  Fish constructor called |

The next program **Java0916.java**, in figure 9.29, changes classes **Cat**, **Bird** and **Fish**, whichnowhave a new **getType** method. The **getType** method has the same name and the same purpose, but the methods are specialized to return the value of the object's data attribute. Using the same class name is the culprit. **Cat**, **Bird** and **Fish** may extend **Animal**, but **getType** is not a *re-definition*, it is *new-definition* and Java cannot find this method anywhere in the **Animal** class.

**Figure 9.29**

|  |
| --- |
| // Java0916.java  // This program adds <getType> methods for each one of the three <Animal> sub classes.  // The program does not compile, because <getType> is not an <Animal> method.  public class Java0916  {  public static void main(String args[])  {  System.out.println("\nJAVA0916\n");  Animal tiger = new Cat("Tiger");  System.out.println(tiger.getType());  Animal eagle = new Bird("Eagle");  System.out.println(eagle.getType());  Animal shark = new Fish("Shark");  System.out.println(shark.getType());  }  }  class Animal  {  public Animal()  {  System.out.println("Animal constructor called");  }  }  class Cat extends Animal  {  protected String catType;  public Cat(String ct)  {  catType = ct;  }  public String getType()  {  return catType;  }  }  class Bird extends Animal  {  protected String birdType;  public Bird(String bt)  {  birdType = bt;  }  public String getType()  {  return birdType;  }  }  class Fish extends Animal  {  protected String fishType;  public Fish(String ft)  {  fishType = ft;  }  public String getType()  {  return fishType;  }  } |

**Figure 9.29 Continued**



Program **Java0917.java**, in figure 9.30, eliminates the error by creating three objects of three separate classes and not extending the **Animal** class. This is all very nice, but it makes you wonder why we started on this road of using the same class identifier with three different constructors. There also is the issue that the program in figure 9.30 does not use inheritance, which is the whole purpose of this chapter.

**Figure 9.30**

|  |
| --- |
| // Java0917.java  // This program does compile, because Each object is a <Cat>,  // <Bird> or <Fish>, which does have a <getType> method.  public class Java0917  {  public static void main(String args[])  {  System.out.println("\nJAVA0917\n");  Cat tiger = new Cat("Tiger");  System.out.println(tiger.getType());  Bird eagle = new Bird("Eagle");  System.out.println(eagle.getType());  Fish shark = new Fish("Shark");  System.out.println(shark.getType());  }  }  class Cat  {  protected String catType;  public Cat(String ct)  {  System.out.println("Cat constructor called");  catType = ct;  }  public String getType()  {  return catType;  }  }  class Bird  {  protected String birdType;  public Bird(String bt)  {  System.out.println("Bird constructor called");  birdType = bt;  }  public String getType()  {  return birdType;  }  }  class Fish  {  protected String fishType;  public Fish(String ft)  {  System.out.println("Fish constructor called");  fishType = ft;  }  public String getType()  {  return fishType;  }  } |

**Figure 9.30 Continued**

|  |
| --- |
| Java0917.java Output JAVA0917  Cat constructor called  Tiger  Bird constructor called  Eagle  Fish constructor called  Shark |

There is a reason for the inheritance and there is a reason to make every object an **Animal** object. Program, **Java0918.java**, in figure 9.31, goes back to the earlier program. Once again **Cat**, **Bird** and **Fish** extend **Animal**, but now the **Animal** class has a peculiar **getType** method that returns an empty string. The **Animal** class does not appear to be that practical either since it stores no values in any attributes.

We call **Animal** an *umbrella* class, since it covers multiple classes. In your school there are boys and girls; there are 9th, 10th, 11th and 12 graders; there are football players, cheerleaders and band members; there are many, many categories in your school and they all fall under the single umbrella of *students*.

This is a very significant concept that will be investigated in great detail in a future chapter titled *Focus on OOP, Polymorphism*. Right now you are seeing the programs in this section for two reasons. First, it sets the stage for the future polymorphism chapter. Second, you will see this type of code the GirdWorld Case Study. You need to understand that it is quite normal, and desirable, to declare multiple classes under one umbrella. For now the understanding part is not at all complete, but please accept that it is appropriate and used for *polymorphism* in Object Oriented Programming.

**Figure 9.31**

|  |
| --- |
| // Java0918.java  // This program solves the problem of the program Java0916.java.  // The <Animal> superclass has a "fake" <getType> method that is  // re-defined for each subclass.  public class Java0918  {  public static void main(String args[])  {  System.out.println("\nJAVA0918\n");  Animal tiger = new Cat("Tiger");  System.out.println(tiger.getType());  Animal eagle = new Bird("Eagle");  System.out.println(eagle.getType());  Animal shark = new Fish("Shark");  System.out.println(shark.getType());  }  }  class Animal  {  public Animal()  {  System.out.println("Animal constructor called");  }  public String getType()  {  return "";  }  }  class Cat extends Animal  {  protected String catType;  public Cat(String ct)  {  catType = ct;  }  public String getType()  {  return catType;  }  }  class Bird extends Animal  {  protected String birdType;  public Bird(String bt)  {  birdType = bt;  }  public String getType()  {  return birdType;  }  }  class Fish extends Animal  {  protected String fishType;  public Fish(String ft)  {  fishType = ft;  }  public String getType()  {  return fishType;  }  } |

**Figure 9.31 Continued**

|  |
| --- |
| Java0918.java Output JAVA0918  Animal constructor called  Tiger  Animal constructor called  Eagle  Animal constructor called  Shark |

**9.8 Summary**

This chapter provided a little information into a variety of topics that are all closely related. Java is an object oriented programming language, and the three corner stones of OOP are *encapsulation*, *class interaction* and *polymorphism*. In this chapter the first part of class interaction, *inheritance*,was introduced.

With inheritance it is possible to use existing classes with all their available features, both methods and attributes. The existing class is called the *superclass* and the new class derived from the superclass is called the *subclass*. Inheritance involves an "is-a" relationship. This means that first and foremost every subclass "is-a"superclass. Geometry provides good examples, such as *a square is-a rectangle*. In Java the reserved word **extends** is used to indicate that a class declaration is a subclass of an existing superclass.

It is very important not to confuse *inheritance*with *composition*. A new class can be declared that includes existing components. A new class may have multiple members, which are existing classes. In such a case the new class demonstrates a "has-a"relationship, because it has certain other members. We can also say that the new class is composed of other components and hence the term *composition*. In Geometry we do not say *a rectangle is-a lines****,*** but we cetainly can say *a rectangle has lines*.

The GridWorld Case Study uses inheritance extensively. The **Actor** class is the superclass for all the objects on the grid. Projects with new class definitions were used to demonstrate inheritance concepts.

This chapter also introduced the **protected** reserved word, which is frequently used with inheritance. Superclass attributes that are **private** cannot be accessed by a subclass. However, if a superclass attribute is declared **protected** it can be accessed by subclass methods, while simultaneously denying access by any other program areas outside the superclass and subclass.

The biggest benefit of inheritance is the ability to use program components that have already been created and tested. New and improved versions of existing components can be designed without starting from scratch. This improves program design efficiency as well as increase program reliability.

A subclass does not reinvent the wheel. You create a new class with new methods. Some methods have the same identifier as the superclass methods. In such a case you are re-defining existing methods. In some other cases there are completely new methods with new identifiers. In the second case you are *newly-defining* methods.

Careful attention must be paid to information that is passed to other constructors. With inheritance, information is passed from the subclass constructor to the superclass constructor with the reserved word **super**. The **super** statement must be the first statement in the subclass constructor.

The constructor of the superclass is always called before the constructor of the subclass. If no information is passed to the superclass constructor, then it is not necessary to include a call to **super**. Java will make this call automatically for you. If you do pass information to the superclass, you must use a call to **super** with the required information for the constructor passed as parameters. The call to **super** must be the first statement in the subclass constructor.