**Chapter XII**

**Focus on OOP,**

**Class Interaction with Composition**

**Chapter XII Topics**

12.1 Introduction

12.2 Simple Composition Examples

12.3 Inheritance and Composition

The Jack O'Lantern Case Study

12.4 Inheritance and Composition

The Train Class Case Study

12.5 Using Arrays for Composition

12.6 GridWorld and Composition

12.7 GWCS, Re-Defined or Newly-Defined?

12.8 The Humble Rock Class

12.9 The Flower Class

12.10 The Bug Class

12.11 Summary

**12.1 Introduction**

One of the corner stones in Object Oriented Programming is *class interaction*, which is divided into two subtopics of *inheritance* and *composition*. The first subtopic of *inheritance* was presented back in Chapter IX. Now we continue and focus on *composition*.

Composition, the *has-a* relationship, does not get the same press and recognition as *inheritance*. Inheritance is the big deal in OOP and composition is mentioned rather casually without much respect. It is not really fair to composition, because the *has-a* relationship is used in many programs. Furthermore, knowing how to use inheritance and knowing how to pass information to superclass constructors does not help with composition. The process is very different.

Since the very first class that you created there was data in the class. These data elements were also called *attributes* or *instance variables*. The majority of early data types you saw were simple or primitive data types like **int**, **double** and **char**. It certainly makes sense that the attributes in a class can be the objects of another class.

Regardless of using inheritance, composition or using both in the same class, the primary goal of OOP is satisfied. In both cases reliability is achieved by using one or more classes that has already been tested.

There is a reason why this chapter did not follow the inheritance chapter immediately. Composition is very frequently used in a class, which has an array of objects. The two previous chapters on static and dynamic arrays needed to be presented first before a more complete chapter on composition is possible.

You will find that composition is also widely used in the GridWorld Case Study and you will look at some examples of classes in the GWCS that demonstrate the use of composition.

Class interaction requires that the existing class, which is being used either in an "is-a" relationship or a "has-a" relationship receives proper information for its constructor. You are using an existing class and that is not possible without first constructing an object of the existing class. This general concept of dealing with the existing class first is the same for inheritance and composition. The actual implementation to pass information to the constructor of the existing class is quite different. With inheritance the **super** method was used to pass information to the superclass. Composition also needs to pass information to another class, but the approach will be very different.

**12.2 Simple Composition Examples**

Consider program **Java1201.java**, in figure 12.1, which instantiates a **Car** class. However, a **Car** object cannot be constructed until an **Engine** object is available. An **Engine** object is an attribute in the **Car** class. A car "has-an" engine.

Now a good question comes to mind. How is information passed to the *attribute* constructor in composition? Think back to classes with only simple data types. Each one of the attributes is declared in the class and inside the constructor are assignment statements to provide information for the data during the instantiation of a new object. This same approach is used when an object contains an object.

Look at the program example in figure 12.1 and then continue to read additional details on the steps required to make composition work correctly. It may be a little trickier than inheritance, but it really follows the same process you have seen used in constructors for some time now.

**Figure 12.1**

|  |
| --- |
| // Java1201.java  // This program uses an <Engine> object in a "has-a" composition relationship.  public class Java1201  {  public static void main(String args[])  {  System.out.println("\nJAVA1201\n");  Car car = new Car("Ford",350);  System.out.println();  car.showData();  System.out.println();  }  }  class Engine  {  private int horsePower;    public Engine(int hp)  {  System.out.println("Engine Constructor Called");  horsePower = hp;  }  public int getHorsePower()  {  return horsePower;  }  }  class Car  {  private String type;  private Engine engine;    public Car(String t, int hp)  {  System.out.println("Car Constructor Called");  type = t;  engine = new Engine(hp);  }  public void showData()  {  System.out.println("Car Type: " + type);  System.out.println("Horse Power: " + engine.getHorsePower());  }  } |

**Figure 12.1 Continued**

|  |
| --- |
| Java1201.java Output JAVA1201  Car Constructor Called  Engine Constructor Called  Car Type: Ford  Horse Power: 350 |

To see how the construction of two objects is possible, let us follow the information starting with the **Car car = new Car("Ford",350);** statement in the **main** method.

The two parameters arrive in the **Car** constructor. Inside the **Car** constructor a new **Engine** object is constructed and it is the **Car** constructor that assigns the proper information to the class attributes. Parameter **Ford** is assigned to the **Car** **type** and **350** is used in the **engine = new Engine(hp);** statement to construct a new **Engine** object.

Program **Java1202.java** used a single class to construct a **Car** object. Program **Java1202.java**, in figure 12.2, once again presents the **Car** class. However, there are now multiple classes, **Wheel**, **Engine** and **Paint**,used in a program example that demonstrates multiple examples of composition.

**Figure 12.2**

|  |
| --- |
| // Java1202.java  // This program uses multiple classes not in an "is-a" inheritance relationship,  // but a "has-a" composition relationship.  public class Java1202  {  public static void main(String args[])  {  System.out.println("\nJAVA1202\n");  Car car = new Car("Ford",6,350,"Red");  System.out.println();  car.getData();  System.out.println();  }  }  class Wheel  {  private int wheelCount;  public Wheel(int wc)  {  System.out.println("Wheel Constructor Called");  wheelCount = wc;  }    public int getWheelCount() { return wheelCount; }  }  class Engine  {  private int horsePower;    public Engine(int hp)  {  System.out.println("Engine Constructor Called");  horsePower = hp;  }  public int getHorsePower() { return horsePower; }  }  class Paint  {  private String paintColor;    public Paint(String pc)  {  System.out.println("Paint Constructor Called");  paintColor = pc;  }    public String getPaintColor() { return paintColor; }  }  class Car  {  private String type;  private Wheel wheel;  private Engine engine;  private Paint paint;      public Car(String t, int wc, int hp, String pc)  {  System.out.println("Car Constructor Called");  type = t;  wheel = new Wheel(wc);  engine = new Engine(hp);  paint = new Paint(pc);  }  public void getData()  {  System.out.println("Car Type: " + type);  System.out.println("Wheel Count: " + wheel.getWheelCount());  System.out.println("Horse Power: " + engine.getHorsePower());  System.out.println("Paint Color: " + paint.getPaintColor());  }  } |

**Figure 1202 Continued**

|  |
| --- |
| Java1202.java Output JAVA1202  Car Constructor Called  Wheel Constructor Called  Engine Constructor Called  Paint Constructor Called  Car Type: Ford  Wheel Count: 6  Horse Power: 350  Paint Color: Red |

**12.3 Inheritance and Composition**

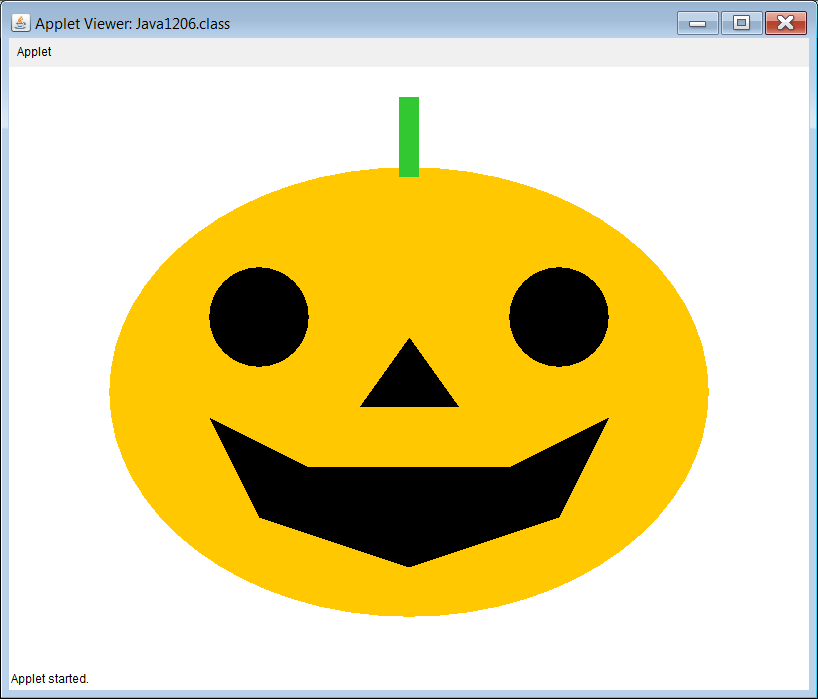
**The Jack O'Lantern Case Study**

I do know that you consider black-and-white text-output programs dull. Frankly, I also think that they are pretty boring. The key advantage of these text programs is that you can introduce a bunch of concepts with a minimum of program code. You have seen some program examples with inheritance and you have also seen a couple program examples with composition. This section and the next section will feature some programs that use both inheritance and composition.

Furthermore, these programs will be presented in the case study style and they will use graphics to make the output more attractive and visual.

The case study in this section involves the display of a *Jack O'lantern*. It is not a complex case study and there are only four stages in the total program. It is a nice straight forward program that I hope will help to illustrate the points made in this chapter. The intention of the program is to create a program that displays the image shown in figure 12.3 below. Notice the pleasant, smiling face. The world has too many gruesome images.

**Figure 12.3**



**Jack O'lantern Case Study, Stage #1**

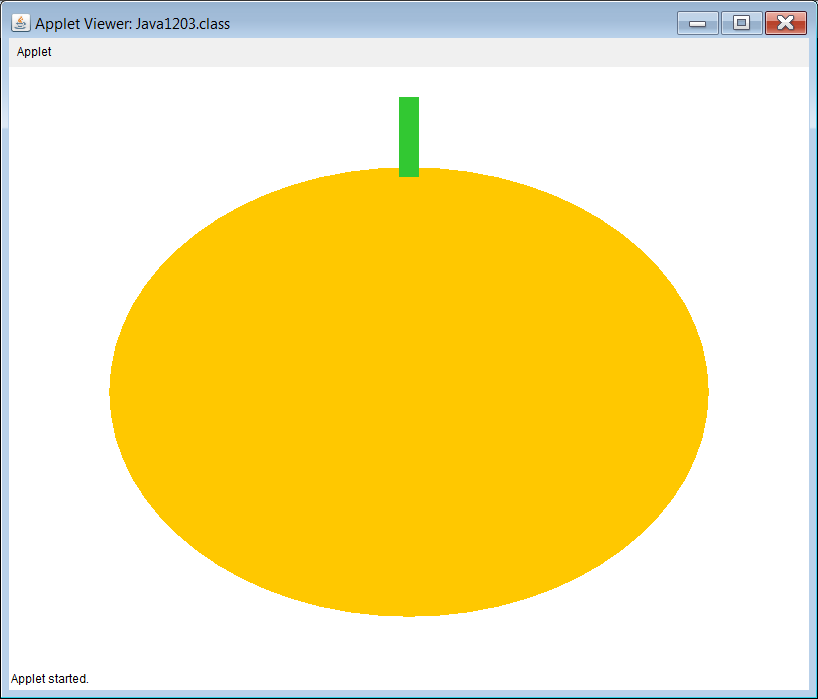
Program **Java1203.java**, in figure 12.4, starts by displaying a pumpkin. This is your classic pumpkin before any carving has started. The first stage does not demonstrate any type of inheritance and there is also nothing going on in the composition department. Stage #1 gets the ball rolling by showing you how the **Pumpkin** class works.

You also need to look carefully at the passing of the **Graphics** object **g**. If you do not connect the **paint** method with the constructor of the **Pumpkin** class, you will be able to compile the program, but you will also look at a lovely white applet window without any pumpkin in sight. Figure 12.5 shows a nice, fat pumpkin.

**Figure 12.4**

|  |
| --- |
| // Java1203.java  // Jack O'lantern Case Study, Stage #1  // This program draws a pumpkin.  // There is neither inheritance nor composition present.  import java.awt.\*;  import java.applet.\*;  public class Java1203 extends Applet  {  public void paint(Graphics g)  {  Pumpkin p = new Pumpkin(g);  }  }  class Pumpkin  {  public Pumpkin(Graphics g)  {  drawPumpkin(g);  }  public void drawPumpkin(Graphics g)  {  g.setColor(Color.orange);  g.fillOval(100,100,600,450);  g.setColor(new Color(50,200,50));  g.fillRect(390,30,20,80);  }  } |

**Figure 12.5**



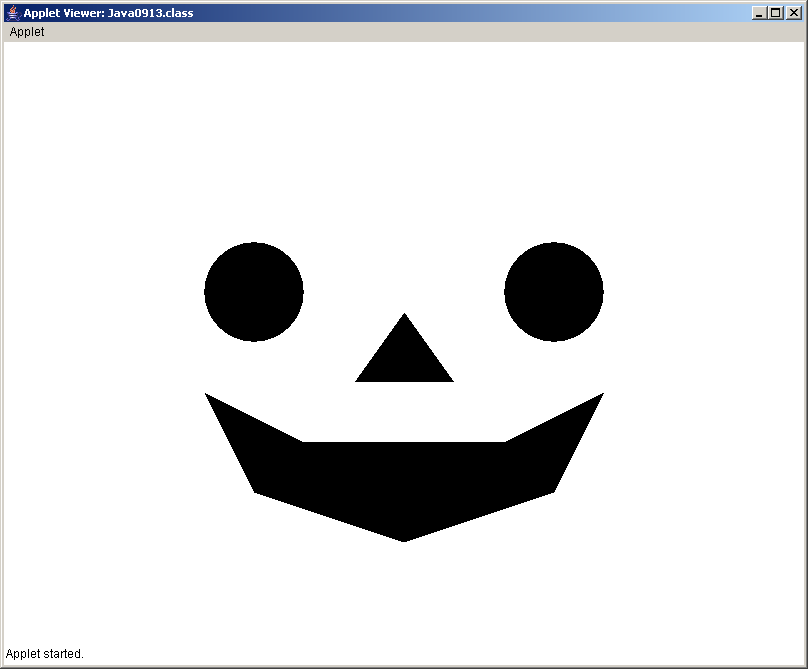
**Jack O'lantern Case Study, Stage #2**

Program **Java0913.**java, in figure 12.6 shows that once again there is only a single class presented. The display in figure 12.7 is a face and nothing else.

**Figure 12.6**

|  |
| --- |
| // Java1204.java  // Jack O'lantern Case Study, Stage #2  // This program draws a face.  // There is neither inheritance nor composition present.  import java.awt.\*;  import java.applet.\*;  public class Java1204 extends Applet  {  public void paint(Graphics g)  {  Face f = new Face(g);  }  }  class Face  {  public Face(Graphics g)  {  drawFace(g);  }  public void drawFace(Graphics g)  {  // Draw eyes  g.setColor(Color.black);  g.fillOval(200,200,100,100);  g.fillOval(500,200,100,100);  // Draw nose  Polygon nose = new Polygon();  nose.addPoint(350,340);  nose.addPoint(450,340);  nose.addPoint(400,270);  g.fillPolygon(nose);  // Draw mouth  Polygon mouth = new Polygon();  mouth.addPoint(300,400);  mouth.addPoint(200,350);  mouth.addPoint(250,450);  mouth.addPoint(400,500);  mouth.addPoint(550,450);  mouth.addPoint(600,350);  mouth.addPoint(500,400);  g.fillPolygon(mouth);  }  } |

**Figure 12.7**

****

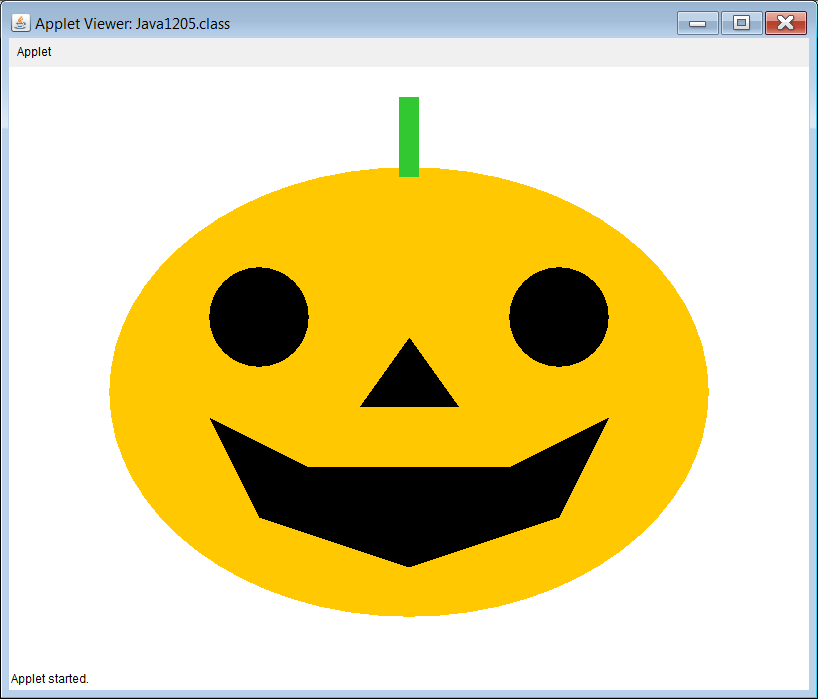
**Jack O'lantern Case Study, Stage #3**

Stage #3 finally gets down to the business of using class interaction. Stage #3 shows composition. Program **Java1205.java**, in figure 12.8 draws the complete Jack O'lantern. The **Pumpkin** class has been altered and now includes a **Face** class attribute. The result is a complete image shown in figure 12.9.

**Figure 12.8**

|  |
| --- |
| // Java1205.java  // Jack O'lantern Case Study, Stage #3  // This program demonstrates composition.  // The <Pumpkin> class now "has-a" <Face> object attribute.  import java.awt.\*;  import java.applet.\*;  public class Java1205 extends Applet  {  public void paint(Graphics g)  {  Pumpkin pumpkin = new Pumpkin(g);  }  }  class Pumpkin  {  private Face face;  public Pumpkin(Graphics g)  {  drawPumpkin(g);  face = new Face(g);  }  public void drawPumpkin(Graphics g)  {  g.setColor(Color.orange);  g.fillOval(100,100,600,450);  g.setColor(new Color(50,200,50));  g.fillRect(390,30,20,80);  }  }  class Face  {  public Face(Graphics g)  {  drawFace(g);  }  public void drawFace(Graphics g)  {  // Draw eyes  g.setColor(Color.black);  g.fillOval(200,200,100,100);  g.fillOval(500,200,100,100);  // Draw nose  Polygon nose = new Polygon();  nose.addPoint(350,340);  nose.addPoint(450,340);  nose.addPoint(400,270);  g.fillPolygon(nose);  // Draw mouth  Polygon mouth = new Polygon();  mouth.addPoint(300,400);  mouth.addPoint(200,350);  mouth.addPoint(250,450);  mouth.addPoint(400,500);  mouth.addPoint(550,450);  mouth.addPoint(600,350);  mouth.addPoint(500,400);  g.fillPolygon(mouth);  }  } |

**Figure 12.9**



**Jack O'lantern Case Study, Stage #4**

We are now ready for the last Jack O'Lantern stage where both composition and inheritance is used. The last stage did somewhat of a nono. It did show an example of composition, but not really in a desirable manner.

We took an existing class, **Pumpkin**, and then altered this class so that it would draw both a pumpkin and a face. That works, but it is not in the proper spirit of OOP. OOP is reliability, reliability and more reliability. The whole point of class interaction with inheritance and composition is to use existing classes. It is fine to create new classes with the help of existing classes. You may use inheritance, composition or both, but it is not correct to take an existing class and alter its behavior. You have lost the capabilities of the existing **Pumpkin** class.

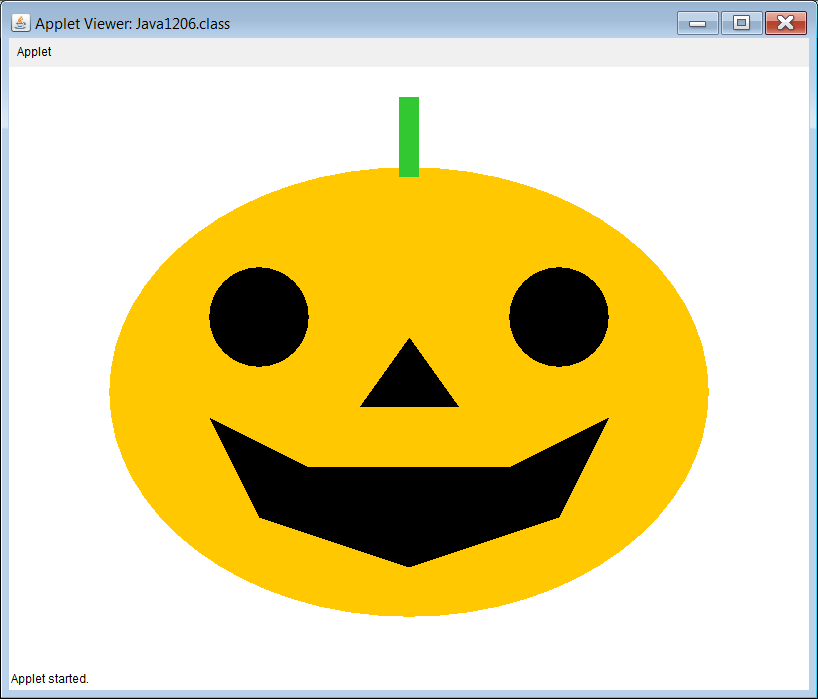
Stage #4 finally will write the program correctly and do this with a combination of inheritance and composition. We start with a **Pumpkin** class and a **Face** class. These are the two original classes and they will not be changed. Now we add a new **JackOLantern** class. Consider how this is stated in written English and you will realize the class interaction with inheritance and composition. ***A*** ***Jack O'Lantern "is-a" Pumpkin and it "has-a" Face***.

Program **Java1206.java**, in figure 12.10, shows the improved final stage of the Jack O'Lantern case study. You will see the original **Pumpkin** class and also the **Face** class. Neither of these two classes have any interaction. Then comes the new **JackOLantern** class, which **extends** the **Pumpkin** class and has a **Face** attribute. Figure 12.11 shows that it creates the same output.

**Figure 12.10**

|  |
| --- |
| // Java1206.java  // Jack O'lantern Case Study, Stage #4  // This program demonstrates both inheritance and composition.  import java.awt.\*;  import java.applet.\*;  public class Java1206 extends Applet  {  public void paint(Graphics g)  {  JackOLantern jack = new JackOLantern(g);  }  }  class Pumpkin  {  public Pumpkin(Graphics g)  {  drawPumpkin(g);  }  public void drawPumpkin(Graphics g)  {  g.setColor(Color.orange);  g.fillOval(100,100,600,450);  g.setColor(new Color(50,200,50));  g.fillRect(390,30,20,80);  }  }  class Face  {  public Face(Graphics g)  {  drawFace(g);  }  public void drawFace(Graphics g)  {  // Draw eyes  g.setColor(Color.black);  g.fillOval(200,200,100,100);  g.fillOval(500,200,100,100);    // Draw nose  Polygon nose = new Polygon();  nose.addPoint(350,340);  nose.addPoint(450,340);  nose.addPoint(400,270);  g.fillPolygon(nose);    // Draw mouth  Polygon mouth = new Polygon();  mouth.addPoint(300,400);  mouth.addPoint(200,350);  mouth.addPoint(250,450);  mouth.addPoint(400,500);  mouth.addPoint(550,450);  mouth.addPoint(600,350);  mouth.addPoint(500,400);  g.fillPolygon(mouth);  }  }  class JackOLantern extends Pumpkin  {  private Face f;  public JackOLantern(Graphics g)  {  super(g);  f = new Face(g);  }  } |

**Figure 12.11**



**12.3 Inheritance and Composition**

**The Train Case Study**

The second case study with inheritance and composition will be more sophisticated than the first one. The concepts of class interaction are fundamentally the same, but the actual classes are larger and include various features that were not shown in the last section.

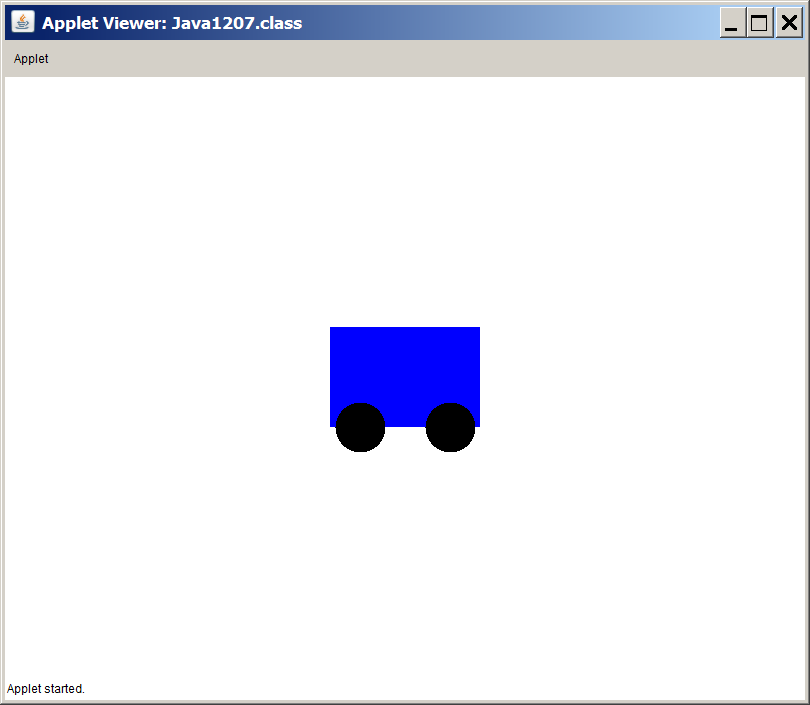
**Train Case Study, Stage #1**

Stage #1 starts with a **TrainCar** class. Program **1207.java**, in figure 12.12 presents a rather primitive **TrainCar** class, which always draws the same blue train car in the same location. Figure 12.13 shows the lonely, blue train car fixed in its one location.

**Figure 12.12**

|  |
| --- |
| // Java1207.java  // Train case study, Stage #1  // The first stage starts with the <TrainCar> class.  import java.awt.\*;  import java.applet.\*;  public class Java1207 extends Applet  {  public void paint(Graphics g)  {  TrainCar tc = new TrainCar();  tc.drawTrainCar(g);  }  }  class TrainCar  {  private Color carColor;  public TrainCar()  {  carColor = Color.blue;  }  public void drawTrainCar(Graphics g)  {  g.setColor(carColor);  g.fillRect(325,250,150,100);  g.setColor(Color.black);  g.fillOval(330,325,50,50);  g.fillOval(420,325,50,50);  }  } |

**Figure 12.13**



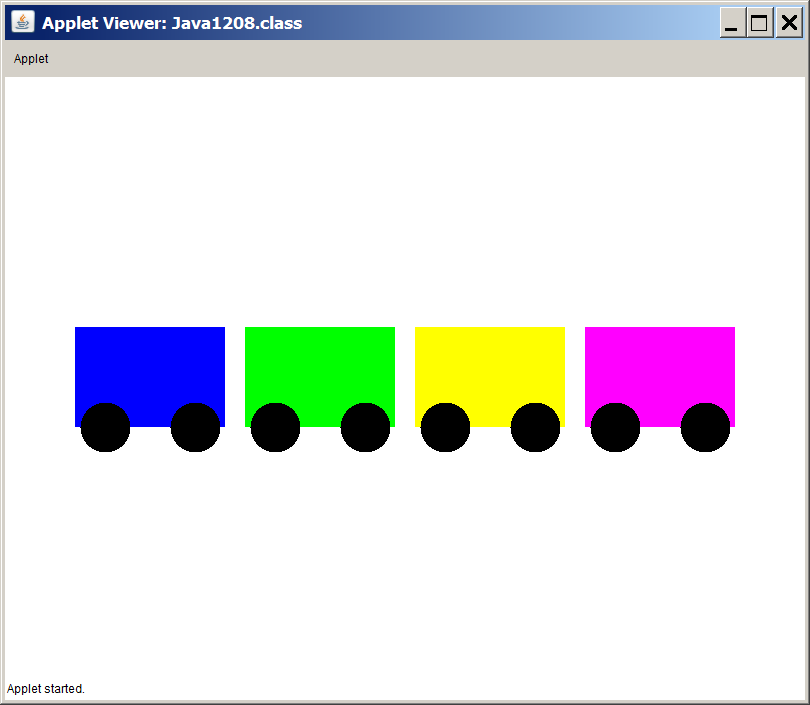
**Train Case Study, Stage #2**

Stage #2 shows that a case study stage is not always the next class created in a sequence of interacting classes. In this case the existing **TrainCar** class is improved. There is neither inheritance nor composition used. Program **Java1208.java**, in figure 12.14, improves the **TrainCar** class with a better constructor method. Now you can specify both the color and the location for the new **TrainCar** object. Figure 12.15 shows that there are now four **TrainCar** objects and they can be placed in different locations with different colors.

**Figure 12.14**

|  |
| --- |
| // Java1208.java  // Train case study, Stage #2  // This program improves the <TrainCar> class by constructing  // new objects with a specified color and a specified location.  import java.awt.\*;  import java.applet.\*;  public class Java1208 extends Applet  {  public void paint(Graphics g)  {  TrainCar tc1 = new TrainCar(Color.blue,70,250);  TrainCar tc2 = new TrainCar(Color.green,240,250);  TrainCar tc3 = new TrainCar(Color.yellow,410,250);  TrainCar tc4 = new TrainCar(Color.magenta,580,250);  tc1.drawTrainCar(g);  tc2.drawTrainCar(g);  tc3.drawTrainCar(g);  tc4.drawTrainCar(g);  }  }  class TrainCar  {  private Color carColor;  private int xPos;  private int yPos;  public TrainCar(Color cC, int xP, int yP)  {  carColor = cC;  xPos = xP;  yPos = yP;  }  public void drawTrainCar(Graphics g)  {  g.setColor(carColor);  g.fillRect(xPos,yPos,150,100);  g.setColor(Color.black);  g.fillOval(xPos+5,yPos+80,50,50);  g.fillOval(xPos+95,yPos+75,50,50);  }  } |

**Figure 12.15**



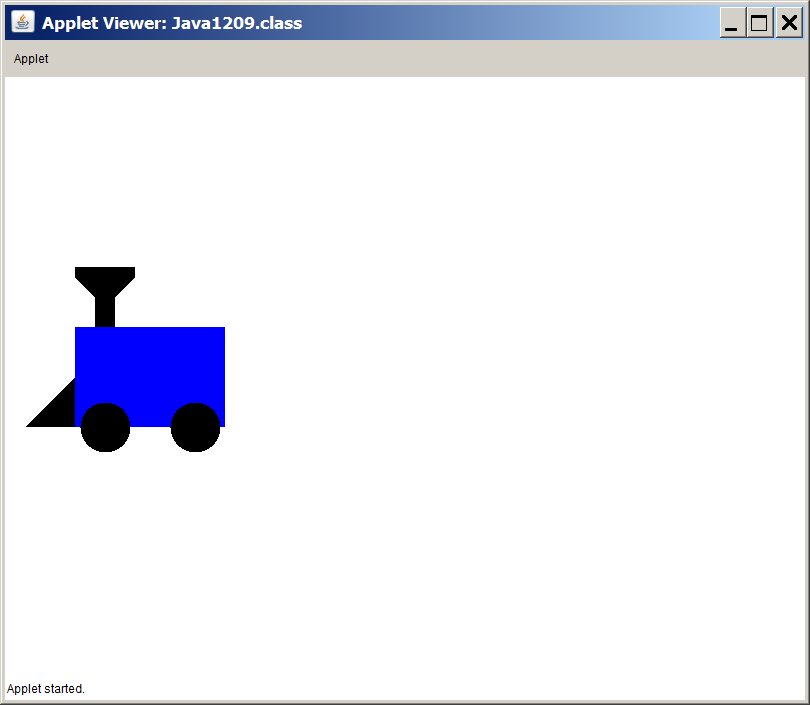
**Train Case Study, Stage #3**

Stage #3 adds the **Locomotive** class, which is a subclass of the **TrainCar** class. A **Locomotive** "is-a" **TrainCar**. Program **Java1209.java**, in figure 12.16, shows that the **Locomotive** class has its own constructor and it has re-defined the **drawTrainCar** method. Figure 12.17 shows the proud **Locomotive** object.

**Figure 12.16**

|  |
| --- |
| // Java1209.java  // Train case study, Stage #3  // This program adds the <Locomotive> class,  // using inheritance, since a locomotive is-a traincar.  import java.awt.\*;  import java.applet.\*;  public class Java1209 extends Applet  {  public void paint(Graphics g)  {  Locomotive loc = new Locomotive(Color.blue,70,250);  loc.drawTrainCar(g);  }  }  class TrainCar  {  protected Color carColor;  protected int xPos;  protected int yPos;  public TrainCar(Color cC, int xP, int yP)  {  carColor = cC;  xPos = xP;  yPos = yP;  }  public void drawTrainCar(Graphics g)  {  g.setColor(carColor);  g.fillRect(xPos,yPos,150,100);  g.setColor(Color.black);  g.fillOval(xPos+5,yPos+80,50,50);  g.fillOval(xPos+95,yPos+75,50,50);  }  }  class Locomotive extends TrainCar  {  public Locomotive(Color cc, int xP, int yP)  {  super(cc,xP,yP);  }  public void drawTrainCar(Graphics g)  {  super.drawTrainCar(g);  drawScoop(g);  drawFunnel(g);  }  private void drawScoop(Graphics g)  {  Polygon scoop = new Polygon();  scoop.addPoint(xPos,yPos+50);  scoop.addPoint(xPos,yPos+100);  scoop.addPoint(xPos-50,yPos+100);  g.setColor(Color.black);  g.fillPolygon(scoop);  }  private void drawFunnel(Graphics g)  {  Polygon funnel = new Polygon();  funnel.addPoint(xPos+20,yPos);  funnel.addPoint(xPos+20,yPos-30);  funnel.addPoint(xPos,yPos-50);  funnel.addPoint(xPos,yPos-60);  funnel.addPoint(xPos+60,yPos-60);  funnel.addPoint(xPos+60,yPos-50);  funnel.addPoint(xPos+40,yPos-30);  funnel.addPoint(xPos+40,yPos);  g.setColor(Color.black);  g.fillPolygon(funnel);  }  } |

**Figure 12.17**



**Train Case Study, Stage #4**

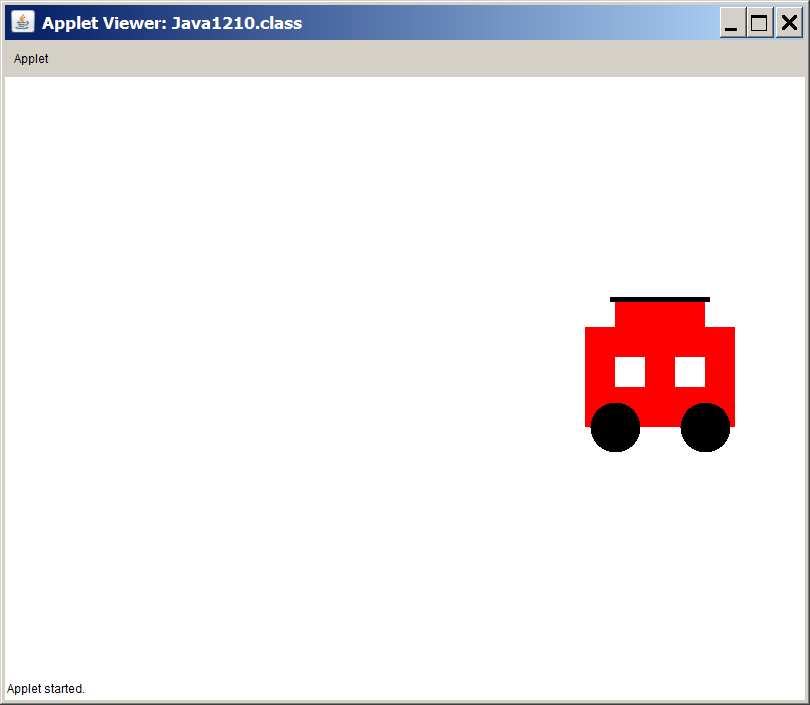
Program **Java1210.**java, in figure 12.18, shows that stage #4 is very similar to stage #3. Once again, inheritance is demonstrated by creating a **Caboose** class, which **extends** the **TrainCar** class.

Figure 12.19 shows the caboose, which is the special train car attached at the end of the train. This program only shows the caboose. In the next, and final stage all the classes will come together to demonstrate both inheritance and composition.

**Figure 12.18**

|  |
| --- |
| // Java1210.java  // Train case study, Stage #4  // This program adds the <Caboose> class,  // using inheritance, since a caboose "is-a" traincar.  import java.awt.\*;  import java.applet.\*;  public class Java1210 extends Applet  {  public void paint(Graphics g)  {  Caboose cab = new Caboose(Color.red,580,250);  cab.drawTrainCar(g);  }  }  class TrainCar  {  protected Color carColor;  protected int xPos;  protected int yPos;  public TrainCar(Color cC, int xP, int yP)  {  carColor = cC;  xPos = xP;  yPos = yP;  }  public void drawTrainCar(Graphics g)  {  g.setColor(carColor);  g.fillRect(xPos,yPos,150,100);  g.setColor(Color.black);  g.fillOval(xPos+5,yPos+80,50,50);  g.fillOval(xPos+95,yPos+75,50,50);  }  }  class Caboose extends TrainCar  {  public Caboose(Color cc, int xP, int yP)  {  super(cc,xP,yP);  }  public void drawTrainCar(Graphics g)  {  super.drawTrainCar(g);  drawWindows(g);  drawTop(g);  }  private void drawWindows(Graphics g)  {  g.setColor(Color.white);  g.fillRect(xPos+30,yPos+30,30,30);  g.fillRect(xPos+90,yPos+30,30,30);  }  private void drawTop(Graphics g)  {  g.setColor(Color.red);  g.fillRect(xPos+30,yPos-30,90,30);  g.setColor(Color.black);  g.fillRect(xPos+25,yPos-30,100,5);  }  } |

**Figure 12.19**



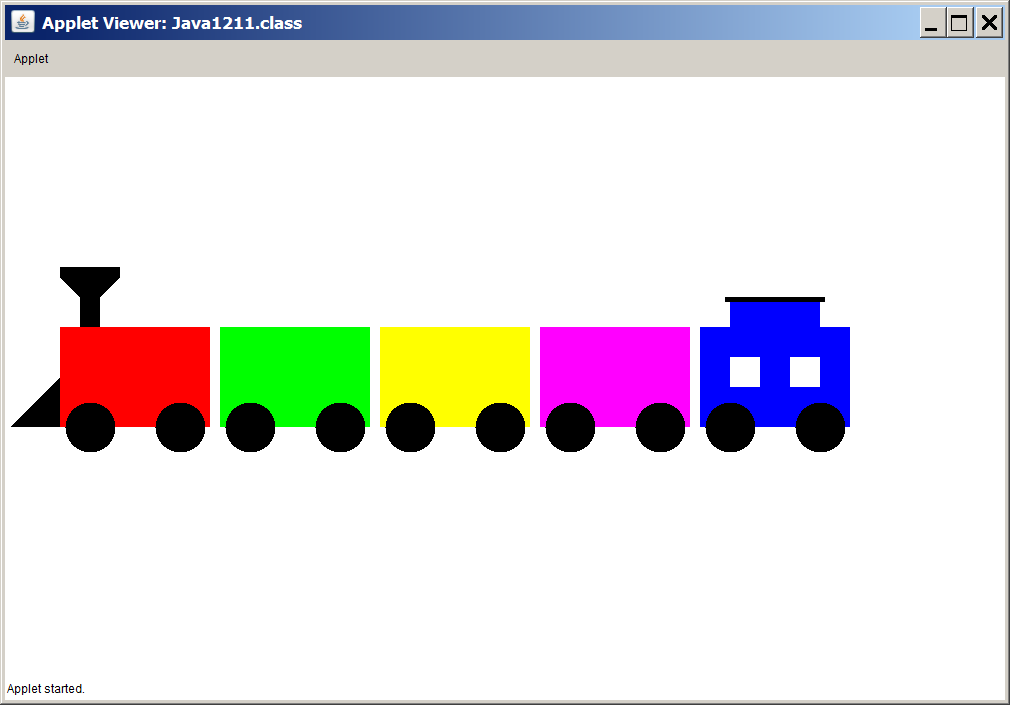
**Train Case Study, Stage #5**

You have now arrived at the exciting conclusion of this case study. Program **Java1211.java**, in figure 12.20, is quite long as it uses all the previous classes and a new **Train** class. You have already seen two cases of inheritance. A locomotive "is-a" train car and a caboose "is-a" train car. Now the **Train** class is added and this class shows composition, because a train has train cars. Figure 12.21 shows that this exciting final stage with inheritance and composition working together to create a nice five car display.

**Figure 12.20**

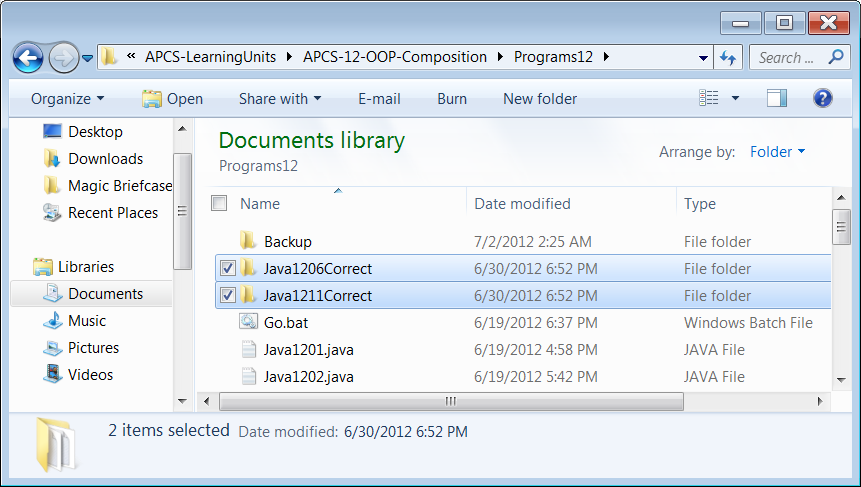
|  |
| --- |
| // Java1211.java  // Train case study, Stage #5  // This program concludes by adding the <Train> class  // A train has train cars. The first train car is the locomotive.  // The last train car is the Caboose.  // This program now combines inheritance with composition.  import java.awt.\*;  import java.applet.\*;  public class Java1211 extends Applet  {  public void paint(Graphics g)  {  Train t = new Train(55,250);  t.drawTrain(g);  }  }  class Train  {  private Locomotive loc;  private TrainCar tc1;  private TrainCar tc2;  private TrainCar tc3;  private Caboose cab;  private int tlX;  private int tlY;  public Train(int tlX, int tlY)  {  this.tlX = tlX;  this.tlY = tlY;  loc = new Locomotive(Color.red,tlX,tlY);  tc1 = new TrainCar(Color.green,tlX+160,tlY);  tc2 = new TrainCar(Color.yellow,tlX+320,tlY);  tc3 = new TrainCar(Color.magenta,tlX+480,tlY);  cab = new Caboose(Color.blue,tlX+640,tlY);  }  public void drawTrain(Graphics g)  {  loc.drawCar(g);  tc1.drawCar(g);  tc2.drawCar(g);  tc3.drawCar(g);  cab.drawCar(g);  }  }  class TrainCar  {  protected Color carColor;  protected int xPos;  protected int yPos;  public TrainCar(Color cC, int xP, int yP)  {  carColor = cC;  xPos = xP;  yPos = yP;  }  public void drawCar(Graphics g)  {  g.setColor(carColor);  g.fillRect(xPos,yPos,150,100);  g.setColor(Color.black);  g.fillOval(xPos+5,yPos+80,50,50);  g.fillOval(xPos+95,yPos+75,50,50);  }  }  class Locomotive extends TrainCar  {  public Locomotive(Color cc, int xP, int yP)  {  super(cc,xP,yP);  }  public void drawCar(Graphics g)  {  super.drawCar(g);  drawScoop(g);  drawFunnel(g);  }  private void drawScoop(Graphics g)  {  Polygon scoop = new Polygon();  scoop.addPoint(xPos,yPos+50);  scoop.addPoint(xPos,yPos+100);  scoop.addPoint(xPos-50,yPos+100);  g.setColor(Color.black);  g.fillPolygon(scoop);  }  private void drawFunnel(Graphics g)  {  Polygon funnel = new Polygon();  funnel.addPoint(xPos+20,yPos);  funnel.addPoint(xPos+20,yPos-30);  funnel.addPoint(xPos,yPos-50);  funnel.addPoint(xPos,yPos-60);  funnel.addPoint(xPos+60,yPos-60);  funnel.addPoint(xPos+60,yPos-50);  funnel.addPoint(xPos+40,yPos-30);  funnel.addPoint(xPos+40,yPos);  g.setColor(Color.black);  g.fillPolygon(funnel);  }  }  class Caboose extends TrainCar  {  public Caboose(Color cc, int xP, int yP)  {  super(cc,xP,yP);  }  public void drawCar(Graphics g)  {  super.drawCar(g);  drawWindows(g);  drawTop(g);  }  private void drawWindows(Graphics g)  {  g.setColor(Color.white);  g.fillRect(xPos+30,yPos+30,30,30);  g.fillRect(xPos+90,yPos+30,30,30);  }  private void drawTop(Graphics g)  {  g.setColor(carColor);  g.fillRect(xPos+30,yPos-30,90,30);  g.setColor(Color.black);  g.fillRect(xPos+25,yPos-30,100,5);  }  } |

**Figure 12.21**



In the **Programs12** folder are two folders called **Java1206** and **Java1211**, shown in figure 12.22. Both folders end with **Correct**. The programs for the **Jack O'Lantern** and the **Train** case studies each were written for convenience of teaching. Every one of the classes was placed in a single file. The "**Correct"** folders show the correct or proper style where each class is inside its own file ... that is, each **public** class should be inside its own file.

**Figure 12.22**

****

**12.5 Using Arrays for Composition**

Program **Java1212.java**, in figure 12.22, shows the truly practical use of composition as you will see it in many real-life programs. Consider this: you have a **Student** class. Objects of the **Student** class store name, address, grade and lots of other information pertaining to a student. We can call this class a un it class. It is a single unit of information, like a student, a passenger, a patient or an employee. Now what can you do with a unit class, like the **Student** class? Well ... by itself, not much unless you have a single private student. In a practical program another class is created, which uses has a data attribute that is an array data structure. Each element of the array is a single **Student** object. A very appropriate name for the array is **students**.

The program in figure 12.22 shows a **School** class. A school has students. That is composition. Using the technique of placing an array data structure as a data attribute in a class is very common. In **Java1212.java** the **Student** class is quite lean and only stores **name** and **gpa** data. That is not realistic, but the concept of using an array for composition in a class is the same regardless of the number of data fields that are stored by each array element.

**Figure 12.22**

|  |
| --- |
| // Java1212.java  // This program demonstrates composition by creating a class,  // which "has-an" array of objects of another class.  // In this program static arrays are used in the <School> class.  public class Java1212  {  public static void main(String args[])  {  System.out.println("\nJAVA1212\n");  String[] names = {"Tom","Sue","Joe","Meg","Bob","Ann","Dan","Jan","Ken","Kim"};  double[] gpas = {2.125,2.175,2.225,2.275,3.125,3.175,3.325,3.375,3.675,3.875};  School planoWest = new School(names.length);  planoWest.loadData(names,gpas);  planoWest.printStudents();  System.out.println();  }  }  class Student  {  private String name;  private double gpa;  public Student(String name, double gpa)  {  this.name = name;  this.gpa = gpa;  }  public void printStudent()  {  System.out.println("Name: " + name);  System.out.println("GPA: " + gpa);  }  }  class School  {  private int count;  private Student[] students;  public School(int n)  {  count = n;  students = new Student[count];  }  public void loadData(String[] names, double[] gpas)  {  for (int k = 0; k < count; k++)  {  Student temp = new Student(names[k],gpas[k]);  students[k] = temp;  }  }  public void printStudents()  {  for (Student s: students)  s.printStudent();  }  } |

**Figure 12.22 Continued**

|  |
| --- |
| Java1212.java Output JAVA1212  Name: Tom  GPA: 2.125  Name: Sue  GPA: 2.175  Name: Joe  GPA: 2.225  Name: Meg  GPA: 2.275  Name: Bob  GPA: 3.125  Name: Ann  GPA: 3.175  Name: Dan  GPA: 3.325  Name: Jan  GPA: 3.375  Name: Ken  GPA: 3.675  Name: Kim  GPA: 3.875 |

Program **Java1212.java** used a static array to store the **Student** objects. This did require that the size of the array is known and used at construction time. This is one handicap of using static arrays. An organization, like a school, has a flexible number of students that changes considerably. A dynamic array, like **ArrayList**, may be a better choice.

Program **Java1213.java**, in figure 12.23, uses the dynamic **ArrayList** class to store the **Student** objects. If you compare the **School** constructors of both programs, you will see that the previous program requires the student quantity when the new **School** object is instantiated. This is not the case when **ArrayList** is used for storage.

In figure 12.23 you might wonder about the **names** and **gpas** arrays. They both sure look like static arrays. They certainly are, but they have nothing to do with the design of the **School** class. The **names** and **gpas** arrays are used to test the **School** class for both programs. Static arrays have a convenient initializer list that is an excellent tool to test classes with specific data. This is much faster than entering data at the keyboard. Another convenient testing technique is to use data files, but you have not yet learned how to enter data from an external file.

**Figure 12.23**

|  |
| --- |
| // Java1213.java  // This program demonstrates composition by creating a class,  // which "has-an" array of objects of another class.  // In this program dynamic arrays are used in the <School> class.  import java.util.ArrayList;  public class Java1213  {  public static void main(String args[])  {  System.out.println("\nJAVA1213\n");  String[] names = {"Tom","Sue","Joe","Meg","Bob","Ann","Dan","Jan","Ken","Kim"};  double[] gpas = {2.125,2.175,2.225,2.275,3.125,3.175,3.325,3.375,3.675,3.875};  School planoWest = new School();  planoWest.loadData(names,gpas);  planoWest.printStudents();  System.out.println();  }  }  class Student  {  private String name;  private double gpa;  public Student(String name, double gpa)  {  this.name = name;  this.gpa = gpa;  }  public void printStudent()  {  System.out.println("Name: " + name);  System.out.println("GPA: " + gpa);  }  }  class School  {  private ArrayList<Student> students;  public School()  {  students = new ArrayList<Student>();  }  public void loadData(String[] names, double[] gpas)  {  for (int k = 0; k < names.length; k++)  {  Student temp = new Student(names[k],gpas[k]);  students.add(temp);  }  }  public void printStudents()  {  for (Student s: students)  s.printStudent();  }  } |

**Figure 12.23 Continued**

|  |
| --- |
| Java1213.java Output JAVA1213  Name: Tom  GPA: 2.125  Name: Sue  GPA: 2.175  Name: Joe  GPA: 2.225  Name: Meg  GPA: 2.275  Name: Bob  GPA: 3.125  Name: Ann  GPA: 3.175  Name: Dan  GPA: 3.325  Name: Jan  GPA: 3.375  Name: Ken  GPA: 3.675  Name: Kim  GPA: 3.875 |

**12.6 GridWorld and Composition**

The GridWorld Case Study is an example of a professional program that is designed and implemented by master programmers. You learn from a very good example. Many computer science concepts can be demonstrated with one or more of the classes in the GridWorld program.

This is a *Focus on OOP, Composition* chapter, so what about composition? Is it used in the GWCS? You have seen plenty of examples of inheritance. The superclass of all GridWorld occupants is the **Actor** class and classes like **Rock, Flower**, **Bug** and **Critter** are all subclasses of the **Actor** class.

The inheritance trend continues as the **BoxBug** is a subclass of the **Bug** class and the **CrabCritter** and **ChameleonCritter** classes are subclasses of the **Critter** class. In chapter IX the GWCS was used to introduce inheritance features.

Now let us look at composition and start with the very important **Actor** class. It is not necessary to look at the entire class. The majority of the **Actor** members are methods that are not a concern here. There are four data attributes that we will examine closer and they are shown in figure 12.24, which is a small section of the **Actor** class. We will look at each of the four attributes and decide if this is a case of encapsulation.

**Figure 12.24**

|  |
| --- |
| public class Actor{private Grid<Actor> grid;private Location location;private int direction;private Color color;public Actor(){color = Color.BLUE;direction = Location.NORTH;grid = null;location = null; **}** |

You may be tempted to say that with four attributes there must be four composition cases. After all the **Actor** class "has four data fields”. That is true, but keep in mind that composition is a type of class interaction where one class, the *container* class has an object of another, the *contained* class, as an attribute.

|  |
| --- |
| private int direction; |
| The attribute direction is a member of the Actor class, but it is not an object. This is a primitive data type that does not demonstrate any type of relationship between two classes. The direction field is not composition. |

|  |
| --- |
| private Location location; |
| The attribute location is an object of the Location class and it is a member of the Actor class. We can say that an Actor object "has-a" Location. The location field is an example of composition. |

|  |
| --- |
| private Color color; |
| The attribute color is an object of the Color class and it is a member of the Actor class. We can say that an Actor object "has-a" Color. The color field is an example of composition. |

|  |
| --- |
| private Grid<Actor> grid; |
| The attribute grid is an object of the Grid class and it is a member of the Actor class. We can say that an Actor object "has-a" Grid. The grid field is an example of composition. |

In the last section two programs were shown that demonstrated composition where one class "has-an" array of objects of another class. It was mentioned that this is a practical and common practice in many real-life programs. Does this also happen in the GWCS. Figure 12.25 shows part of the **BoundedGrid** class.

The **BoundedGrid** class is a *generic* class. You are used to seeing generics type declarations that look like **ArrayList<Actor> a = new ArrayList<Actor>();** What you have seen are examples of using generic classes. The program code you see in figure 12.25 is the declaration of a generic class. Much more will be explained about this in a later chapter, but right now look at **<E>** as a special variable **E**, which stands for Element and will become whatever class is provided at construction time.

Inside this class is an attribute called **occupantArray**. This is a two-dimensional static array of **Object** elements. The **Object** class is the superclass of all classes in Java and what you see here is an array that is capable of storing any type of element that comes its way.

Think about it. GridWorld is a classic example of composition. You have a two-dimensional grid of individual cells. Each cell has room for one object. The class that stores all the objects on the grid uses composition. The grid has objects. An ideal attribute for the **BoundedGrid** class is to use a two-dimensional array.

**Figure 12.25**

|  |
| --- |
| public class BoundedGrid<E> extends AbstractGrid<E>{private Object[ ][ ] occupantArray;public BoundedGrid(int rows, int cols){if (rows <= 0)throw new IllegalArgumentException("rows <= 0");if (cols <= 0)throw new IllegalArgumentException("cols <= 0");occupantArray = new Object[rows][cols]; **}** |

**12.7 GWCS, Re-Defined or Newly-Defined?**

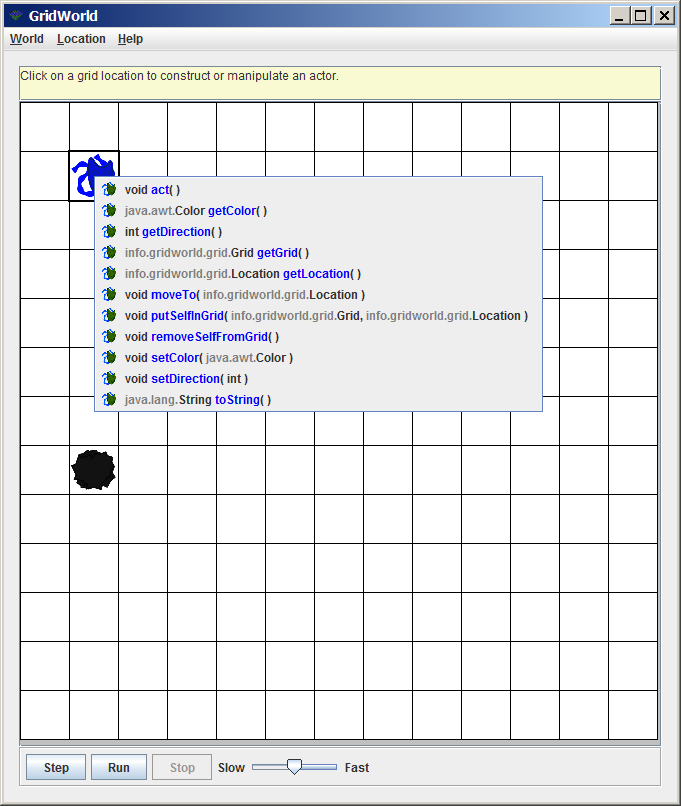
Figure 12.26 shows a partial grid display with four objects. There is an **Actor** object, a **Bug** object, a **Flower** object and a **Rock** object displayed. You learned in an earlier chapter that you can click on any Grid object and display the available methods of that class. The GridWorld program has been designed so that more is shown than just a set of methods. Let us look at each object in turn and find out its available methods.

**Figure 12.26**

|  |  |
| --- | --- |
|  | You should pretty much expect that each class has the same methods. It starts with a set of methods that are defined in the **Actor** class. These methods were explained in the previous chapter.  The **Bug**, **Flower** and **Rock** classes are all three subclasses of the **Actor** class. This means that *at a minimum* they have all the methods that are already defined for **Actor**.  Now there are two possible changes. First, one or more methods defined in the **Actor** class is *re-defined* in one of its subclasses to provide new and improved - or desired - functionality. Second, you may see one or more new methods in any of the subclasses that never existed in the **Actor** class. |

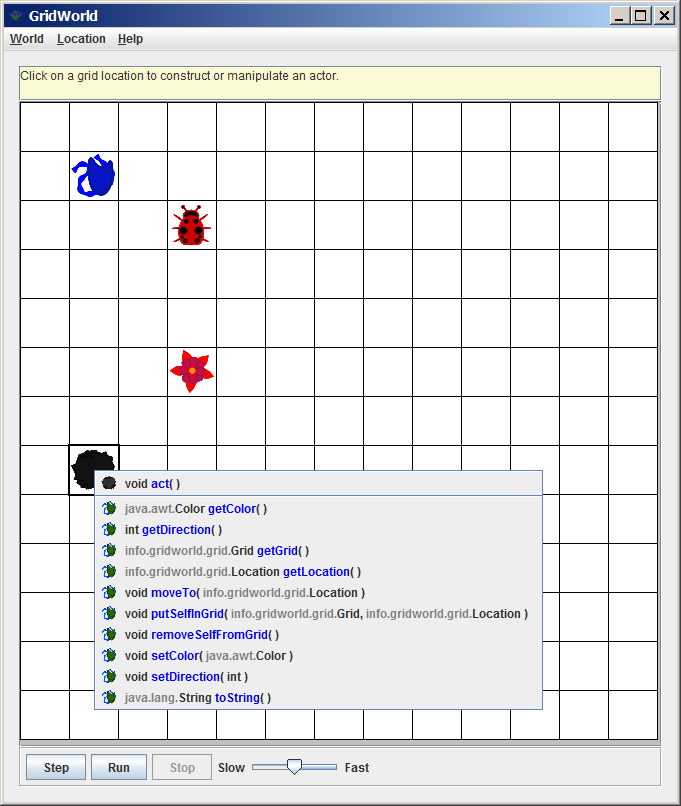
Figure 12.27 reviews the methods of the **Actor** class. The constructor method is intentionally not included. The list that is shown contains all the methods that can be called with the current object on the grid. A constructor was called to create the object initially and once the object exists you do not call the constructor again.

**Figure 12.27**



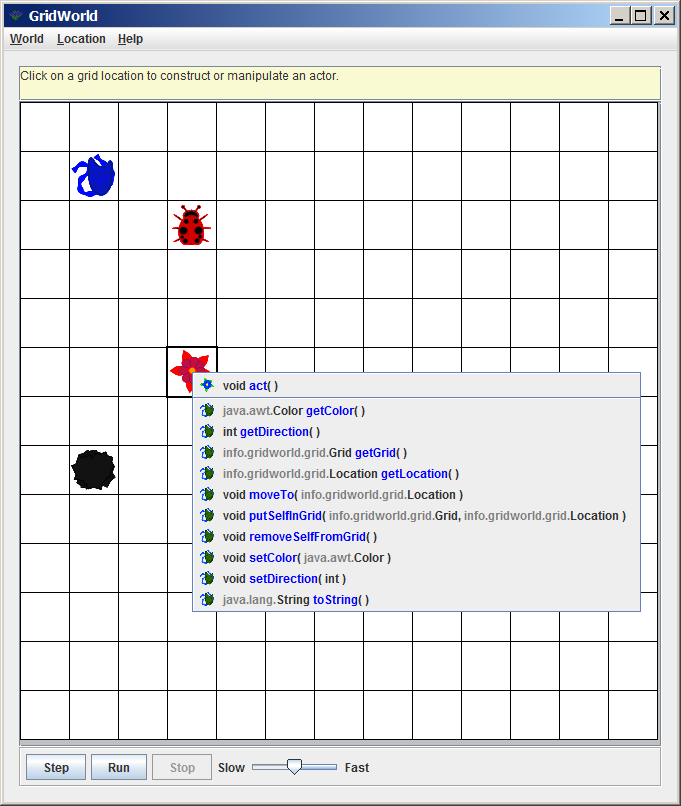
The **Rock** class displays the exact same methods shown already by the **Actor** class, but in a different manner, shown in figure 12.28. The **act** method is shown above a border and the icon is not the actormask, but a rock. This means that only the **act** method is re-defined in the **Rock** class.

**Figure 12.28**



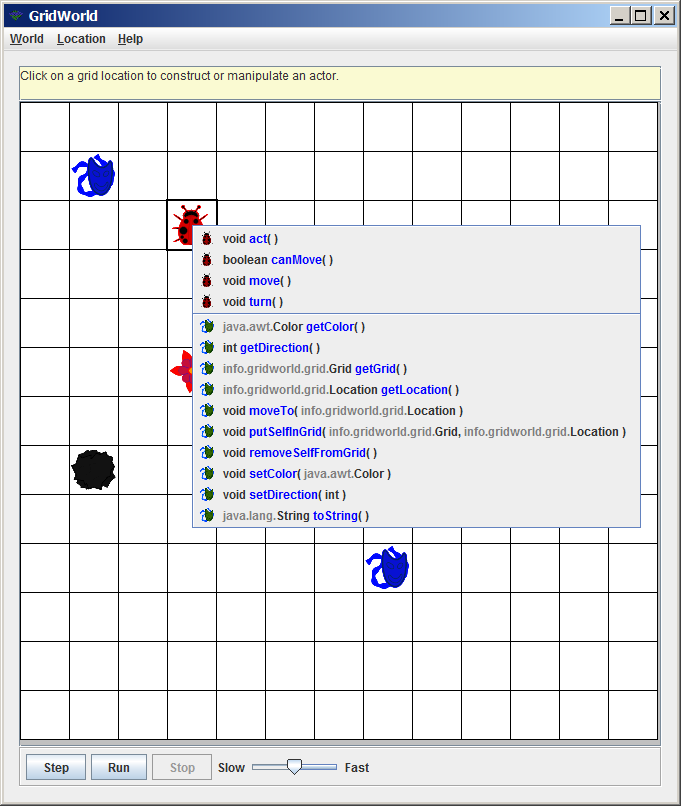
The **Flower** class is about as original as the **Rock** class. Figure 12.29 shows that once again all the **Actor** methods are available and once again only the **act** method is re-defined, as evidenced by the border and the **Flower** icon.

**Figure 12.29**



The **Bug** class, in figure 12.30, adds some excitement. The **act** method shows up with a **Bug** icon above the border, like the **Rock** and **Flower** classes. Above the border are also new **canMove**, **move** and **turn** methods. These last three methods never existed in the **Actor** class and that means that they must be newly-defined methods exclusive to the **Bug** class. Looking only at the **Bug** class you cannot tell if the methods with the bug icon are re-defined or newly defined. You can determine this indirectly, by looking at the **Actor** class methods and realize that any method with the same name is re-defined.

**Figure 12.30**



**12.8 The Humble Rock Class**

What does it take to use inheritance? At a minimum the Java keyword **extends** must be used. Figure 12.31 shows an example of *minimal* inheritance. There is a class heading and nothing else. The class heading states that *Rock extends Actor*. This means that **Actor** is the original, or *superclass* and **Rock** is the subclass, which inherits the **Actor** features. This empty class does compile and it will execute, but all you have done is create an **Actor** class with a different name. Nothing has changed besides the name. Not a single method is re-defined and not one new method is added. Java does not care and only checks for proper syntax. The class has a correct heading. The class has a "body" in the form of a set of braces and putting something sensible in the class container is optional. In other words, Java is happy and does not care that nothing has really changed.

**Figure 12.31**

|  |
| --- |
| **public class Rock extends Actor**  **{**  **}** |

|  |
| --- |
| **Class Headings with Inheritance** |
| The class heading indicates an ***is-a***inheritance relationship between two classes with the use of the keyword **extends**.  **public class Rock extends Actor**  The first class, **Rock** in this example, is the *subclass*.  The second class, **Actor** following extends, is the *superclass*. |

Figure 12.32 shows the actual **Rock** class declaration of the GridWorld program. The comments are removed to make the class members easier to see. The class has four members. There are three methods and one attribute. Each of the class members will be studied in detail.

**Figure 12.32**

|  |
| --- |
| public class Rock extends Actor  {  private static final Color DEFAULT\_COLOR = Color.BLACK;  public Rock()  {  setColor(DEFAULT\_COLOR);  }  public Rock(Color rockColor)  {  setColor(rockColor);  }  public void act()  {  }  } |

The first member in the **Rock** class is an attribute that stores a constant color. In the spirit of good program design this declaration exists for the purpose of better readability. Default is what you get when you do not specify. The **Rock** class has two constructors. One constructor lets you pick a color and the other constructor gives you no choices.

Figure 12.33 shows that the default color in this case is black. The **final** keyword makes the "variable" a constant. *Constant variable* sounds like an oxymoron, but it prevents anybody from changing the default color.

**Figure 12.33**

|  |
| --- |
| **private static final Color DEFAULT\_COLOR = Color.BLACK;** |

The **Rock** class has two constructors, shown in figure 12.34. When there are two or more methods, including constructor methods, with the same name, such methods are called *overloaded*. From Java's point of view the names may be the same, but the method *signatures* are different. One method has no parameters and the second method does have a parameter. So far the **Rock** class has made some minor changes from its **Actor** parent. If nothing is specified when a new object is constructed (the definition of default) you get a black rock. You do have the option to specify a color at construction time.

**Figure 12.34**

|  |
| --- |
| **public Rock()**  **{**  **setColor(DEFAULT\_COLOR);**  **}**  **public Rock(Color rockColor)**  **{**  **setColor(rockColor);**  **}** |

The only other change is the **act** method, shown in figure 12.35, which is re-defined in the **Rock** class. You may think that the method is not defined at all. It looks empty and incomplete. This is an unusual case where a totally empty method actually does mean a definition. The **act** method in the **Actor** class is originally defined to make an object turn 180 degrees with every step it takes. Rocks are expected to sit and do nothing. This is precisely how **act** is re-defined.

**Figure 12.35**

|  |
| --- |
| **public void act()**  **{**  **}** |

**12.9 The Flower Class**

You have seen enough GridWorld executions to know that **Bug** objects move along and leave a trail of flowers behind. You have probably also noticed that flowers steadily become darker with each execution step. Now is the time to see how these flowers manage to act the way that they do.

Figure 12.36 shows the **Flower** class. Once again the class is quite small. This is exactly what happens with inheritance. The superclass carries a lot of the weight by defining all the basic methods. A subclass can be any size, but frequently a subclass changes only a few critical methods. In the case of the **Actor** class, the primary change you will observe in its subclasses is the re-definition of the **act** method. You will also see one or more constructors, insuring the initial state of the new object

**Figure 12.36**

|  |
| --- |
| public class Flower extends Actor  {  private static final Color DEFAULT\_COLOR = Color.PINK;  private static final double DARKENING\_FACTOR = 0.05;  public Flower()  {  setColor(DEFAULT\_COLOR);  }  public Flower(Color initialColor)  {  setColor(initialColor);  }  public void act()  {  Color c = getColor();  int red = (int) (c.getRed() \* (1 - DARKENING\_FACTOR));  int green = (int) (c.getGreen() \* (1 - DARKENING\_FACTOR));  int blue = (int) (c.getBlue() \* (1 - DARKENING\_FACTOR));  setColor(new Color(red, green, blue));  }  } |

Figure 12.37 shows that the **Flower** class has two constants or **final** attributes. The first attribute **DEFAULT\_COLOR** is used for the initial pink color for flower construction. The second attribute, called **DARKENING\_FACTOR** may appear strange. You know that the flowers become darker with age. During execution each step makes the flower darker by a small degree or factor. In this case the darkening has a value of 0.05. This value will make more sense when we get to the **act** method.

**Figure 12.37**

|  |
| --- |
| **private static final Color DEFAULT\_COLOR = Color.PINK;**  **private static final double DARKENING\_FACTOR = 0.05;** |

The two constructors in figure 12.38 will become a familiar sight. First there is the no-parameter, default constructor and second, there is the constructor that allows the creation of a new object with a specified color.

**Figure 12.38**

|  |
| --- |
| **public Flower()**  **{**  **setColor(DEFAULT\_COLOR);**  **}**  **public Flower(Color initialColor)**  **{**  **setColor(initialColor);**  **}** |

Figure 12.39 shows the re-defined act method of the Flower class. Check it out first and then you will get a detailed explanation of the program statements.

**Figure 12.39**

|  |
| --- |
| **public void act()**  **{**  **Color c = getColor();**  **int red = (int) (c.getRed() \* (1 - DARKENING\_FACTOR));**  **int green = (int) (c.getGreen() \* (1 - DARKENING\_FACTOR));**  **int blue = (int) (c.getBlue() \* (1 - DARKENING\_FACTOR));**  **setColor(new Color(red, green, blue));**  **}** |

The truth is that the statements in the **act** method have nothing to do with acting in the normal sense. Every statement is connected to changing color.

What is the behavior of a flower? Does it appear behind a bug? Yes it does, but it is not the job of a flower to follow a bug. It is the behavior of a bug to drop a flower. The behavior of a flower involves becoming darker and that is what it does with each execution step. Now let us look at each execution step, shown in figures 12.40 - 12.42.

**Figure 12.40**

|  |
| --- |
| **Color c = getColor();** |
| For each step of the GridWorld execution a grid is shown with objects. Each object is displayed according to its own color. What is that color? The **Actor** class defined a clever method, called **getColor**, which returns the color of the current object. |

**Figure 12.41**

|  |
| --- |
| **int red = (int) (c.getRed() \* (1 - DARKENING\_FACTOR));**  **int green = (int) (c.getGreen() \* (1 - DARKENING\_FACTOR));**  **int blue = (int) (c.getBlue() \* (1 - DARKENING\_FACTOR));** |
| Computer colors are a mix of *red*, *green* and *blue* shades. Each color has a shade value from **0**, the least intense, to **255**, the most intense. The **Color** class, which is not a GridWorld class, has methods **getRed**, **getGreen** and **getBlue** to get the color shades of the current color.  Whatever the shade - a value in the range [0..255] is multiplied times (1 - **0.05)**. The value 0.005 is the DARKENING\_FACTOR of the **Flower** class. There are three **int** variables **red**, **green** and **blue**, which store a value that has a color shade slightly darker than previously. Note that the computation of the darker color is cast to an **int** type. |

**Figure 12.42**

|  |
| --- |
| **setColor(new Color(red, green, blue));** |
| This statement contains a *unanimous* object. A new **Color** object is created without a name and this new color is used to alter the color of the current **Flower** object. |

Will flowers continue to get darker? No color is darker than **0**, which is totally black, and **0 x 0.05** is still equal to **0**. Is it really ever possible to reach **0**? Mathematically, it is not possible, because the fraction keeps getting smaller for an infinite number of times. In computer science it is different. Color shades have integer values and they will certainly reach the minimum value of zero for a totally black flower.

**12.10 The Bug Class**

GridWorld classes fall in three responsibility categories for AP Computer Science testing purposes. The class is *not testable*, the class *code implementation is testable* and the class *API is testable*. So let us take an inventory with the classes presented so far.

|  |
| --- |
| **AP CS Examination GridWorld Requirements** |
| All the classes presented so far, meaning the **Actor**, **Location**, **Rock** and **Flower** classes, only ***test the API***.  This means that you only need to know the behavior of the methods and you need to know how to call the methods with the correct parameter information. |
| The **Bug** class is the first class, which will **test the class code**. In other words you are responsible for every detail of the program code found in the **Bug** class.  Initially, this is quite tricky since so many GridWorld classes interact with each other. Complete understanding will come steadily as you study more classes. |

We started very comfortably with the **Rock** class, moved on to the more interesting **Flower** class and now we arrive at the more involved **Bug** class. A quick glance at Figure 12.43 shows a much larger class than the **Rock** class.

**Figure 12.43**

|  |
| --- |
| public class Bug extends Actor  {  public Bug()  {  setColor(Color.RED);  }  public Bug(Color bugColor)  {  setColor(bugColor);  }    public void act()  {  if (canMove())  move();  else  turn();  }  public void turn()  {  setDirection(getDirection() + Location.HALF\_RIGHT);  }  public void move()  {  Grid<Actor> gr = getGrid();  if (gr == null)  return;  Location loc = getLocation();  Location next = loc.getAdjacentLocation(getDirection());  if (gr.isValid(next))  moveTo(next);  else  removeSelfFromGrid();  Flower flower = new Flower(getColor());  flower.putSelfInGrid(gr, loc);  }  public boolean canMove()  {  Grid<Actor> gr = getGrid();  if (gr == null)  return false;  Location loc = getLocation();  Location next = loc.getAdjacentLocation(getDirection());  if (!gr.isValid(next))  return false;  Actor neighbor = gr.get(next);  return (neighbor == null) || (neighbor instanceof Flower);  }  } |

The constructors are becoming boring. Once again figure 12.44 shows that there is a constructor that makes every unspecified **Bug** object red. There is a second constructor that lets you specify the new color when a new object is constructed.

**Figure 12.44**

|  |
| --- |
| **public Bug()**  **{**  **setColor(Color.RED);**  **}**  **public Bug(Color bugColor)**  **{**  **setColor(bugColor);**  **}** |

Figure 12.45 shows a simple **act** method. There are only a few very short statements. However, there is a problem. There are methods calls to **canMove**, **move** and **turn**. This is not a total surprise, because you did see some of the **Bug** methods in an earlier chapter. These methods are not re-defined from the **Actor** class and they are not in any standard Java library. They must be newly-defined for the **bug** class. Investigating a new method is somewhat like reading. A paragraph does not make sense if it includes unfamiliar vocabulary. You need to check a dictionary for the new words and then return to the reading. This is exactly what we do now. We look up the meanings of **canMove**, **move** and **turn** and then we can comprehend **act** better.

**Figure 12.45**

|  |
| --- |
| **public void act()**  **{**  **if (canMove())**  **move();**  **else**  **turn();**  **}** |

Figure 12.46 shows the first of a set of new methods that are chuck full of new stuff. Bugs want to move to the next cell that they are facing. Now that is easy enough when there is an empty cell available. It is also easy when the cell contains a flower, because bugs can move and destroy flowers. Any other object in a cell causes problems for a bug on the move. In other words, method **canMove** checks to see if the next cell is empty or contains a flower and then returns true, otherwise the method returns false.

**Figure 12.46**

|  |
| --- |
| **public boolean canMove()**  **{**  **Grid<Actor> gr = getGrid();**  **if (gr == null)**  **return false;**  **Location loc = getLocation();**  **Location next = loc.getAdjacentLocation(getDirection());**  **if (!gr.isValid(next))**  **return false;**  **Actor neighbor = gr.get(next);**  **return (neighbor == null) || (neighbor instanceof Flower);**  **}** |

The individual program statements of the **canMove** method will now be separated and explained in detail by Figures 12.47 - 12.51.

**Figure 12.47**

|  |
| --- |
| **Grid<Actor> gr = getGrid();**  **if (gr == null)**  **return false;** |
| The first program statement assigns the current **Grid** object to **gr.**  If it turns out that **gr == null** then there is no **Grid** object and there really is nowhere to move. This means that the method execution terminates and returns a **false** value. |

**Figure 12.48**

|  |
| --- |
| **Location loc = getLocation();**  **Location next = loc.getAdjacentLocation(getDirection());** |
| The first statement here is used frequently in various GridWorld method. It returns the location (row and column value) of the current **Actor** object.  Once the current location is known, the next location must be found. Any location has potentially eight adjacent locations. Method **getAdjacentLocation** returns the location of the adjacent location in the direction that the current object is facing. |

**Figure 12.49**

|  |
| --- |
| **if (!gr.isValid(next))**  **return false;** |
| The **canMove** method is starting to make some headway. We now know the location of the current object and we also know the next location of our object.  Now the question comes, is that next location valid? That may seem like an odd question, but grid can be *bounded* or *unbounded*. If a grid is bounded then it has borders and it is now possible that our bug is facing a border. Method **isValid** checks to see if the next location is valid. If it is not valid the method terminates and returns **false**. |

**Figure 12.50**

|  |
| --- |
| **Actor neighbor = gr.get(next);** |
| If the method execution makes it this far then our bug exists in a **Grid** object and we also know that there is a next cell location that is valid. This is all good progress, but we are not ready to move yet.  What is located at the next location? Method **gr.get(next)** checks out the next location and returns **null** if the next cell is not occupied and returns the object if it is occupied. |

**Figure 12.51**

|  |
| --- |
| **return (neighbor == null) || (neighbor instanceof Flower);** |
| All the information is now available to determine if our bug can move to the next location or not. The final statement uses a compound boolean statement that will return **true** or **false**. What makes a move possible at this stage? If the next location is empty, the bug can move. The bug can also move if the next location contains a flower. |
| **return (neighbor == null) || (neighbor instanceof Flower); <<< NOTE**    is identical to the next code that uses if ... else.  **if ((neighbor == null) || (neighbor instanceof Flower))**  **return true;**  **else**  **return false;**  **<<< NOTE** A double or compound condition is used here, which states in English *check if the neighbor is empty or the neighbor is a flower*. If either condition is true then the **Bug** object is able to move. |

The **move** method will not be as much work to explain as the **canMove** method. This may seem surprising, because the two methods contain about the same number of statements. Now take a look at figure 12.52 and check out the first bunch of statements. Do you observe something? Perhaps it helps to go back and look at the entire **Bug** class in figure 12.43.

Now that you see all the **Bug** methods in one convenient place compare the **canMove** method with the **move** method. It should now be easier to see that a whole bunch of statements are completely identical. This is precisely what happens throughout the GridWorld Case Study.

The **move** method, in figure 12.52, may surprise you by its length. After all, did we not establish that it is possible to move and how difficult is it to move to the next cell? Moving to the next cell is not that tricky, but a bunch of information is required for the movement that was determined in method **canMove** is not stored anywhere. Detailed explanations are shown in Figures 12.53 - 12.55.

**Figure 12.52**

|  |
| --- |
| **public void move()**  **{**  **Grid<Actor> gr = getGrid();**  **if (gr == null)**  **return;**  **Location loc = getLocation();**  **Location next = loc.getAdjacentLocation(getDirection());**  **if (gr.isValid(next))**  **moveTo(next);**  **else**  **removeSelfFromGrid();**  **Flower flower = new Flower(getColor());**  **flower.putSelfInGrid(gr, loc);**  **}** |

**Figure 12.53**

|  |
| --- |
| **Grid<Actor> gr = getGrid();**  **if (gr == null)**  **return;**  **// return false; is not necessary for a void method**  **Location loc = getLocation();**  **Location next = loc.getAdjacentLocation(getDirection());** |
| The **Bug** object moves from the current object location to the next location. All of that was done before, but it must be done again.  The commented and highlighted part shows a small difference between **canMove** and **move**. **canMove** is a **boolean** return method and terminating the method still requires returning **false**. **move** is a void method and only requires termination.  Now the same process must be repeated to find the current **Bug** location and the next location. |

**Figure 12.54**

|  |
| --- |
| **if (gr.isValid(next))**  **moveTo(next);**  **else**  **removeSelfFromGrid();** |
| If the next location is a valid location then our bug gets the green light to go ahead. The **moveTo** method of the **Actor** class is used to move the bug to the next location. Now if the next location is not valid, which happens when a bug runs into a boundary, then the bug removes itself from the grid. |
| Now comes a curious question. Is it possible for a bug to run into a wall and then leave the grid? If the **act** method is used, the answer is no, because the **canMove** method has already checked the validity of the next location. On the other hand if the **move** method is used directly without **canMove** yes then bugs can disappear as they run into walls. |

**Figure 12.55**

|  |
| --- |
| **Flower flower = new Flower(getColor());**  **flower.putSelfInGrid(gr, loc);** |
| Our **Bug** object is now comfortably settled in the next location, but two more statements need to be executed. A flower must be deposited in the last location of the object.  Two statements are needed. You may be under the impression that flowers all start as pink, but that is not actually what happens. The **Flower** class has two constructors and in this case the color is specified. The specified color comes from method **getColor**. What color is that? Method **getColor** comes from **Actor** and gets the color of the current object. In other words, the flower dropped behind will have the same color as the **Bug** object. |

Method **turn** in Figure 12.56, is so nice and small that it is not necessary to analyze the method behavior line-by-line. The **Location** class static attribute is used for a clockwise turn with **Location.HALF\_RIGHT** equals 45.

Method **setDirection** changes the orientation of the **Bug** object by first getting the current direction (the direction that the bug faces) and then adds 45 to that value. In practice, this means that a bug will make a 45 degree, clockwise turn.

**Figure 12.56**

|  |
| --- |
| **public void turn()**  **{**  **setDirection(getDirection() + Location.HALF\_RIGHT);**  **}** |

This chapter can now finish by returning to the **act** method in Figure 12.57. The method is short and simple, much simpler than the methods used. Now that the three methods have been studied the **act** method should make more sense.

First, the bug checks if movement is possible. You now know that will only happen if the next location is valid and furthermore, the next location is empty or a flower. If **canMove** is true, the bug moves to the next location. If **canMove** is false, the bug makes a 45 degree clockwise turn.

**Figure 12.57**

|  |
| --- |
| **public void act()**  **{**  **if (canMove())**  **move();**  **else**  **turn();**  **}** |

**12.11 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*. Inheritance is divided into the two subtopics of *inheritance* and *composition.*  This chapter placed the OOP focus on *composition*.

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 can say *a rectangle has lines*. We also say that *a car has-an engine*, but not that *a car is-an engine*.

Communication is simpler with inheritance. There is a superclass and there is a subclass. For ease of communication we shall say that with composition the class that has-an another class as an attribute is the *container* class and the other class is the *contained* class.

Information is passed to the constructor of the contained class by the constructor of the container class. When the container constructor is called, one of its job is to instantiate the object of the contained class. At the instantiation, information is passed from the container constructor to the contained constructor.

A practical application for composition is to create a *unit* class, which is a class that stores information and methods for a single unit, like a student, patient or passenger. A larger container class can then store an array of the unit class objects. It is frequently the convention to use an array with a plural name. For example a **School** class may include an attribute, which is an array called **students**. Each member of the **students** array is an **Student** object.

The GridWorld case study is an excellent example of class interaction both with inheritance and with composition. The **Actor** class becomes a container class and has three attributes that are objects of another contained class. The **BoundedGrid** class has a static two-dimensional array that stores all the object on the grid.

This chapter provided details of three **Actor** subclasses, **Rock**, **Flower** and **Bug**. Each one of these classes re-defined the **act** method. The **Rock** class alters behavior by doing absolutely nothing. The **Flower** class sits quietly in a cell and slowly wilts to a darker color. The **Bug** class moves to the adjacent cell that it is facing, provided that it is empty or occupied by a **Flower** object, otherwise it finds a new cell by making a 45-degree clockwise turn.

The **Rock** class creates two new constructors. There is a default constructor that makes every new **Rock** object black. A second, overloaded constructor, provides a parameter for a requested color. The **act** method is re-defined and has no statements, which makea the **Rock** object do nothing.

The **Flower** class has two constructors like the **Rock** class, one for the default pink color and a second constructor for a specified color choice. The **act** method is re-defined with various statements that darken the color a small amount with every step of the GridWorld execution.

The **Bug** class continues the constructor tradition and creates default red bugs and a constructor for specified colors. The Bug re-defines the **act** method differently. The **Rock** class and the **Flower** class took care of new behavior completely in the **act** method. The **Bug** method also re-defines the **act** method, but it involves calling re-defined methods **canMove**, **move** and **turn**.