Chapter Two: Classes and Objects

Section 1: Real Objects Versus Virtual Objects

Computer games do not have to match reality. In them we can violate the rules of physics (or make up our own), pigs can fly (well virtual pigs can virtually fly), fire can be hot or cold (or both), and people can be 100 feet tall. The game developers choose what is reality inside the program. We can create any virtual objects we like and we can have them interact and behave in any way we like.

Thinking about programs in terms of objects that interact is known as object oriented programming. The Java language supports this style of programming and is called an object oriented programming language. There are many other different programming languages and ways of designing and creating programs but object oriented programming is arguably the dominant programming models.

We have been talking about classes, objects, and methods without defining them. In this chapter we explain these concepts. In Object-Oriented-Programming the term “Class” is an abstract concept used to signify what data objects contain and what objects can do. The code for an object is found in the class description. All instances of the class, otherwise known as objects, share the same code. You can think of a class as a cookie cutter and objects as the cookies. The cookie cutter determines what the cookies will look like. One uses the cutter (class) to make one or more cookies (objects). You can also think of a class as a mold, something that is used to create object multiples. We start with another real-world analogy and then give specific Java examples.

Example 2.1: The Person Class

In life it is a bad idea to treat people as objects, but it is a helpful learning analogy for our purposes. Lets say there is a mold for making people. The mold specifies the data members that a person object will hold and the methods (actions) that a person object can use. Lets say I create a Person class that specifies **members** and **methods.** The members are the data items an object of that type of class has. The actual data values in the differ-

ent objects can, and usually does, differ. The methods are the actions that an object can perform.

Assume the following (note, this is not exact Java syntax, but it is close):

Person Class

Members:

```
Integer height ;   // in inches
Integer weight;   // in  pounds
Integer age ;      // in years
String name;      // a string of characters that make up the name
```

Methods:

```
bakeCake() ;
stepForward();
stepForward(int numSteps) ;
stepBackward();
stepBackward(int numSteps) ;
jumpInPlace();
raiseLeftHand() ;
raiseRightHand() ;
sayHello() ;
getAge() ;
```

```
setAge(int newAge) ;

getName() ;

setName(char newName) ;

getHeight() ;

setHeight(int newHeight) ;

getWeight() ;

setWeight(int newWeight) ;
```

Given this class one can create objects of this class type and invoke methods on those objects. For example, I could say (again, not exact Java code, but similar):

```
Person p1 = new Person( 72,190,23,”Bob”) ;

Person p2 = new Person(64,115,22,”Sue”) ;

Person p3  = new Person(75,225,30,”Jimmy”) ;
```

The above “code” would create three person objects: p1, p2, and p3. Object p1 is initialized with data member values of 72, 190, 23, and “Bob” for height, weight, age and name respectively. Similarly for objects p2 and p3.

One could then say:

```
p1.stepForward() ;

p2.sayHello();

p3.stepBackward();

 p2.bakeCake() ;
```

The code for these methods can contain anything the programmer wishes, but usually one has the method name match what the code does, so presumably the p1 object, i.e. the person “Bob”, would move one step forward, the p3 object would move one step back, and the p2 object would say hello.

Sometimes the method code is more complex and has many steps. For example, the bakeCake() method might be a long list of things the person object does such as:

- preheat oven to 350
- get out flour
- get out sugar
- get out butter
- get out salt
- get out baking soda
- get out eggs
- mix dry incredients
- beat sugar and butter together
- add eggs to butter and sugar and mix
- mix in dry ingredients
- pour batter into prepared 9" round cake pans
- bake at 350 for 25 minutes or until toothpick comes out clean

So now when I as a programmer want a person object p1 to bake a cake all I have to do is say: p1.bakeCake(). This is a very powerful way of simplifying coding.

Often methods have **parameters**, which allow the programmer to pass information into the method. For example

```
p1.stepForward(3) ;

p2.stepBackward(2) ;

p3.setAge(16) ;
```

The values of the parameters in the above three lines of code are 3, 2, and 45 respectively. Here presumably the p1 object would now take 3 steps for-

ward, the p2 object would take two steps backward, and the p3 object would have it's age data member changed to 16.

Section 2: Object Oriented Programming

There are two main parts to an object: the information that it "knows" and the different ways it can "change" or "behave". The information is stored in **member variables**. The behavior is found in **method** code.

Member Variables

These are a collection of "boxes" or memory storage locations which hold values. Each object owns the information stored in these "boxes". For example, each Person object (from the example above) has its own "box" which holds the current value of *that* person's height. We can make a person taller or shorter by changing the value stored in the person object's height member variable. The person objects above also have "boxes" that hold the object's name, age, and weight.

Methods and Behavior

The different ways that an object can behave are encoded in its methods. Methods are instructions written in code. For example, each Person object can raise their right hand or step forward. The raiseRightHand method defines what it means for a Person object to do this in our virtual world. Often methods exist which change the values of the member variables, such as the setAge(int newAge) method.

Example: Invoking methods on a Person object

We will create a Person object and then show how to get it to perform various actions. First we create a Person object named "george":

```
Person george = new Person();
```

The code to the left of the = creates a variable (or "box) capable of holding a Person object and gives the memory location the name of "george". The code on the right of the = creates a new Person object. The new Person object automatically contains the memory space, or "boxes", that can hold the height, weight, age, and name member variables that belong to every Person object.

Then we can have "george" perform actions by invoking different methods:

```
george.raiseRightHand();
george.jumpInPlace();
george.sayHello();
george.bakeCake() ;
```

Notice in each case, we specify the name of the variable, followed by a dot (or period), and then the name of the method we wish that object to perform.

The collection of methods of an object is often referred to as the *interface* of the object. The methods represent all of the ways an object can interact or interface with the rest of the virtual world. The methods completely define the behavior of the object. It cannot do anything else. If we try something like:

```
george.singTheNationalAnthem();
```

We will get an error because there is no singTheNationalAnthem() method defined for Person objects.

Classes of Objects

Since many individual objects are similar (they store the same kinds of information and exhibit the same kinds of behavior, there is a way to define member variables and methods for whole groups of objects at once. Objects in the same group are said to be in the same class or have the same class type.

In this style of programming, we create *classes* which describe information and behavior of *objects*. Classes are templates or cookie cutters from which individual objects are created. One cookie cutter can be used to create many cookies. Each cookie is a different individual cookie and exists on its own. Each cookie may be different in some way (they contain different molecules of cookie dough, or one may have a slightly larger diameter) and similar in other ways (same shape (circle) and contents (four, sugar, butter, egg, vanilla, baking soda).

Similarly, a class can be used to create many objects in a program. The objects are different individual entities in the virtual world of the program. Each object may be different in some ways and similar in other ways. In particular, each object will store the same types of information in its member variables but may have different values stored in those member variables.

For instance, every person has an age and weight but each person may have different age and weight values.

A Java Person Class

In our example above we stated that the code was close to java. Here is a template for actual Java code for the Person class. Note, there are still missing statements inside each method:

```
class Person
```

```
{
  // Member Variables
  int age;
  int height;
  int weight;
  string name;

  // Methods
  public void stepForward()
  {
     // Add code here to Move One Step Forward
  }
  public void stepForward(int numSteps)
  {
     // Add code here to Move “numSteps” steps forward
  }
  public void stepBackward()
  {
    // Add code here to Move One Step Backward
  }
 public void stepBackward(int numSteps)
  {
    // Add code here to Move “numSteps” steps backward
  }

  public void jumpInPlace()
  {
    // Add code here to Jump in Place
  }
  public void raiseLeftHand()
  {
     // Add code here to Raise Left Hand
  }
  public void raiseRightHand()
  {
     // Add code here to Raise Right Hand
  }
  public void sayHello()
  {
     // Add code here to Say Hello
  }
  public int getAge()
  {
    return age;
  }
  public void setAge(int newAge)
  {
     age = newAge ;
  }
  public int getWeight()
  {
    return weight;
  }
 public void setWeight(int newWeight)
  {
     weight = newWeight ;
```

```
  }
  public int getHeight()
  {
    return height;
  }
 public void setHeight(int newHeight)
  {
     height = newHeight ;
  }
  public string getName()
  {
    return name;
  }
 public void setName(int newName)
  {
     name = newName ;
  }

}
```

Notice there are two stepForward() and two stepBackward methods. One does not take a parameter, and according to the comments in the method moves the object forward/backward one step, the second does take a parameter and according to the comments in the class definition moves the object forward/backward "numSteps" steps, where numSteps is passed into the method when the method is called.

COMMENTS: Note, in the java language and words after a the "//" characters are completely ignored. The "//" is called a comment because one can write "comments" about the code.

HELPFUL HINT: In Greenfoot, we can invoke methods on individual objects using the object menu (right-click on the object's image in the World and choose which method should be executed or performed). This is one way to test a method to see if it works (and does what it is supposed to do).

Section 3: Adding Methods and Members to An Actor

Now that we understand more about methods and members, we can improve our simple Actor class definition. Lets assume I want to make a Bee actor a bit more robust. First, lets add methods for moveLeft and moveRight:

```
public class Bee extends Actor
{
   public void act()
   {
     moveRight();
     // moveUp();
   }
```

```
    public void moveRight()
    // This  method changes the location from (x,y) to (x+1,y)
    {
       setLocation( getX() + 1 , getY() );
    }

    public void moveLeft()
    // This  method changes the location from (x,y) to (x-1,y)
    {
       setLocation( getX() - 1 , getY() );
    }
}
```

EXERCISE 2.1:

(A): Type in the above code for a Bee actor. Compile. Place an object in the world and right click on the object and call the moveLeft() and moveRight() methods.

In the code above are two methods, moveLeft and moveRight. Notice in the act( ) method is a call to the moveRight( ) method. There is also a commented out call to a non-existent moveUp( ) method. In the exercise below you will add that method and then you can uncomment the method call inside of act( ). If you remove the comment characters before wirting the code for the moveUp( ) method you will get an error message. Try it, you won't break anything, just comment it out again or remove the line of code and recompile.

(B): Add two more methods for moveUp() and moveDown(). Compile. Place an object in the world and right click on the object to call the moveUp() and moveDown() methods.

(C) Change the moveRight() method to take a parameter "numSteps". Compile. Place an object in the world and right click on the object to call the moveRight() method. When you do this Greenfoot will pop up a window asking

CLASSROOM KINETIC EXERCISE 1:

Instructor: Create a grid on the floor of 1' x 1' or 2' x 2' square. Make the grid about 6x5. Now write code for Person objects using the setLocation( ), moveLeft( ) , moveRight( ), moveUp( ), and moveDown( ) methods. Use actual students as person objects so they must figure out where they are to move and hence understand the coordinate system.

Section 4: Creating Actor Objects and Placing Them In The World Using Code

So far we have added objects to the world by hand. We click on the box of one of the defined classes and choose “new object” and then drag the newly created object out into the world. It is possible to add new objects to the world as part of our program.

Example: Populating the World Automatically

Create a new scenario similar to BugScenario from Chapter One. Create **BugWorld** as a subclass of World and **Bug** as a subclass of Actor. We will now change the code in BugWorld so that it will automatically create 3 Bug objects when the BugWorld is created. To do this, bring up the code window for BugWorld by double-clicking on its class box.

Change the code to look like this (you just need to add three lines of code) :

```
public class BugWorld extends World
{

   /**
    * Constructor for objects of class BugWorld.
    *
    */
   public BugWorld()
   {
      // Create a new world with 50x50 cells with a cell size of 10x10 pixels.
      super(50, 50, 10);

      // Add these three lines
      Bug theBug = new Bug();
      addObject( theBug, 10, 15 );

      Bug theSecondBug = new Bug();
      addObject( theSecondBug,  12, 20 );

      Bug theThirdBug = new Bug();
      addObject( theThirdBug,  23, 45 );
   }
}
```

Each of the new lines of code creates a new Bug object and adds it to the world at the specified coordinates. The command **new ClassName()** will

create a new object of type ClassName. In the example above we create new Bug objects. The command **addObject(obj , x , y)** will add the object "obj" into the world at grid cell location (x,y). In the example above the "obj" that is added is first theBug, then theSecondbug, and finally theThirdBug.

EXERCISE 2.2:

Modify your scenario so that when it starts up three or more objects automatically populate the world. In other words, modify your world constructor to create three Actor objects and add them to the world as above.

Section 5: Making an Actor go back and forth using Member Variables, Booleans and Boolean Methods

In this section we make an actor move back and forth on it's own. To do this we will use a member data variable of type Boolean (true or false) that specifies if the actor is currently moving to the right. Whether the actor is moving right or left determines whether the moveLeft or moveRight method is called. Before calling moveRight or moveLeft the code first checks if the actor can move left/right by calling the canMoveLeft or canMoveRight method. Here is the code:

```
public class Bee extends Actor
{

   private boolean movingRight = true ;        // a member variable specifying if currently
                                                    // moving to the right initialize to true

   public void act()
   {

      if (movingRight)
      {
         if (canMoveRight() )
            moveRight() ;
         else
            movingRight = false ;
      }
      else
      {
         if (canMoveLeft() )
            moveLeft();
         else
            movingRight = true ;
```

```
      }
   }    // end of the act() method

   public void moveRight()
   {
      setLocation( getX() + 1 , getY() );
   }

   public void moveLeft()
   {
      setLocation( getX() - 1 , getY() );
   }

public boolean canMoveRight()
   //  returns true if object can move one square to the right without leaving the world boundary
   {
      if ( ( getX() + 1 ) < getWorld().getWidth() )    // if new location will be in the world
            return(true) ;
      else
         return(false);
   }

   public boolean canMoveLeft()
   {
      if ( ( getX() - 1 ) > 0 )    // if the new x-location will be > 0
         return(true) ;
      else
         return(false);
   }

}   // end of class definition
```

Section 5.1: Java Concept: Variables

We have been thinking of variables as “boxes” of computer memory which hold values of a specified type. The type of the value is specified when the variable is declared.

The built-in types for numbers are: int, float, and double. Variables of type int can hold integer (positive or negative whole numbers including zero). Variables of type float can hold decimal numbers. Variables of type double can hold decimal numbers which are larger than those that float variables can hold.

To change the value of a variable use = (called the assignment operator). This is not the same as the equal sign used in math. You should read the following statement:

```
int someNumber = 5 ;
```

as “the variable someNumber gets the value of 5” not “the variable someNumber equals 5”.

The arithmetic operators are +, -, * (multiplication), / (division).

Section 5.2: Java Concept: Boolean Variables

A boolean variable contains exactly one of two values: true or false. The following code will create a boolean variable named "hungry" and then set its value to true:

```
boolean hungry;
hungry = true;
```

Boolean variables may also be used as the "test" in if statements:

```
if ( hungry )
{
  lookForFood();
}
else
{
  sleepUntilHungry();
}
```

If the variable hungry holds the value of true, the method lookForFood() will be called otherwise the method sleepUntilHungry() will be called.

When we compare two numeric values using <, <=, >, >=, or ==, the result is a boolean value (either true or false). These symbols are called relational operators because they determine or test the relationship between two values. This is why we can write if statements such as:

```
if ( getX() < 30 )    // if new location will be less than 30
     return(true) ;
else
   return(false);
```

Section 5.3: Java Concept: Methods Can Return a Value

We know methods can take in information (by passing parameter values inside the parenthesis). Methods can also return information. We have already seen examples of this in the canMoveLeft() and canMoveRight() methods. These return true or false based on whether The actor can move and still stay inside the world.

Returning a value from a method allows objects to answer questions asked by other objects. For instance, a method called getHeight() could be added to the Person class:

```
int getHeight()
{
  return height;
}
```

The first line of code says that this method is called getHeight and that it returns an integer value back to whoever calls it. So it is now possible to say:

```
Person george = new Person();
int georgeHeight = george.getHeight();
```

and the variable georgeHeight will now hold the value of george's height.

Section 6: Moving Left/Right Using a Velocity Member

In the proceeding examples The actor was moving left or right by one space. We might want the actor to move left/right by two spaces, or three spaces, or more. We can do this by creating a data member we have name xVelocity that specifies how many spaces The actor should move on each call to moveX(). By allowing xVelocity to hold either positive or negative values we can specify if the actor is moving right (positive value) or left (negative value). Here is the code:

```
public class BeeUsingVelocity extends Actor
{
   private int xVelocity = 2 ;   // how for little red will move each time moveX() is called

    public void act()
   {

      if( canMoveX()  )
      {
          moveX();   // move to the right, how far right is determined by xVelocity
      }
      else
      {
         // else we would go off the screen, so, negate the xVelocity to go in the other direction
         xVelocity = -1 * xVelocity ;
      }
   }

   public boolean canMoveX()
   {
      if (xVelocity > 0)    // we want to move right
      {
         if ( (getX() + xVelocity) < getWorld().getWidth() )
```

Exercise 2.2:
Change the Bee actor example so the object repeatedly moves up and down. Add a movingUp member to the class and methods canMoveUp(), canMoveDown(), moveUp(), and moveDown().

```
                // if proposed new location is less than right side of the world
            return (true) ;
          else
            return( false) ;
      }
      else // we want to move left
      {
          if ( (getX() + xVelocity) >= 0)   // if proposed new location is greater than zero
            return (true) ;
          else
            return( false) ;
      }

   }

   // The following method does the same as canMoveX, but does it in less, and
   // arguably more readable, code by using a compound boolean expression

   public boolean canMoveXSecondWay()
   {
      if ( ( (getX() + xVelocity) < getWorld().getWidth() ) && ( (getX() + xVelocity) >= 0) )
         return (true) ;
      else
         return( false) ;
   }

   public void moveX()
   {
      setLocation( getX() + xVelocity , getY() );
   }

}
```

Now if we want The actor to move by 3 cells, we only need to change the value for xVelocity once, all the rest of the code works correctly!

Section 7: Random Movement

Often we want game sprites to move around on their own behalf. We have made actor objects move back and forth or up and down, but we might prefer a more random movement. Later in the book we will show how to create a patrolling type of movement behavior.

In order to get random movement we need random numbers. The built-in Greenfoot method **Greenfoot.getRandomNumber(N)** returns a random integer between 0 and N-1. Thus, if you say Greenfoot.getRandomNumber(5), it will return one of the numbers {0, 1, 2, 3, 4} randomly and with equal probability. We can use this to create random motion by getting a random number between 0 and 3, and then if the number is 0 moving right, if it is 1 move left, if 2 move down, and if 3 move up. Below is the code.

```
public void act()
   {
      int rnum = Greenfoot.getRandomNumber(4) ;
      if (rnum == 0)
      {
         setLocation( getX() + 1 , getY() ) ;
      }
      if (rnum == 1)
      {
         setLocation( getX() - 1 , getY() ) ;
      }
      if (rnum == 2)
      {
         setLocation( getX()  , getY() + 1) ;
      }
      if (rnum == 3)
      {
         setLocation( getX()  , getY() - 1 ) ;
      }
   }
```

Section 8: Constructing Objects

Sometimes we wish to control what happens when an object is created. This is accomplished by defining special methods for a class called constructors. Usually constructors assign initial values to the various member variables defined in the class. Constructor methods always have the same name as the name of the class itself. For example, we could define the following constructor for the Person class:

```
public Person()
{
  age = 45;
  height = 12;
  weight = 15;
  name = “Kermit”;
}
```

This constructor sets the age to 45, the height to 12, the weight to 15, and the name to Kermit. If the code:

```
Person somebody = new Person();
```

then the somebody object would be a person with the age, height, weight, and name defined above. It is also possible to have constructors which take parameters (or specified information). An example of this is:

```
  public Person( string aName, int anAge, int aHeight, int aWeight )
  {
    name = aName;
    age = anAge;
    height = aHeight;
    weight = aWeight;
  }
```

This constructor requires values to be specified when it is called:

```
  Person anotherbody = new Person( “Kermit”, 65, 12, 15 );
```

The constructor would then set anotherbody’s name to Kermit, age to 65, height to 12, and weight to 15. Consider the following Redhood1 world constructor from the Redhood1 scenario:

```
public class RedHood1 extends World
{
   // Constructor for objects of class RedHood1
   public RedHood1()
   {
      super(60, 50, 10);    // make the world 60x50 cells with a 10x10 pixel cell size

      SimpleRed aSimpleRed = new SimpleRed() ;
      addObject(aSimpleRed,6,12);

      BooleanRed aBooleanRed = new BooleanRed();
      addObject(aBooleanRed,6,20);

      VelocityRed aVelRed = new VelocityRed() ;
      addObject(aVelRed,6,28);
   }
}
```

One can see that RedHood1 is a class definition for a Greenfoot world as it “extends World”. The constructor method has the exact same name as the class name. This is required for a constructor. In the constructor we set the world size to (60,50,10), create three objects using “new:” and place the objects in the world using the “addObject()” method.

Exercise 2.3:

Modify your Bee class so that it has a constructor that takes as an argument the initial velocity for the Bee objects and sets the xVelocity data member value to this argument.

Modify your Bee class so that the Bee moves with random motion and the distance it moves on each call to act is determined by the argument to the constructor.