Chapter 10
Objects II

A class is a definition of common characteristics and methods shared by all things that are of that class. We have the String class, the GRect class, the UFO class and others. Classes are defined in a separate file (usually) and we've been defining our own classes since Chapter 5.

An object is a particular instance of a class. If msg is an instance of the String class, created with a statement such as String msg = new String("Hello"); then we say ‘msg is a string’. We create classes and then instantiate objects of those classes.

The fundamental purpose of the class-object concept is to wrap all of the characteristics and behaviors of a single conceptual idea into one unit. This is known as encapsulation. The String class encapsulates everything we need for storing sequences of characters and all the methods needed to work with them. The UFO class encapsulates everything we need for the visual UFO that we used earlier.

We'll take a much deeper look at classes and objects now.

10.1 Graphic Objects

A graphic object is an object with a significant visual component and we’ve been working with them since day one – GRects, UFOs, etc. are all graphic objects.

Let's begin by recalling the UFO2 class from our earlier work. It constructs a visual representation of a UFO and includes two constructors. From it we'll develop the NewUFO object and test it.

UFO2

```java
//UFO2.java
import acm.graphics.*;
import java.awt.*;

public class UFO2 extends GCompound {
    private GOval body, bubble, alien;

    public UFO2() {
        body = new GOval(50, 25);
        body.setFilled(true);
    }
}
```

These are private instance variables. They are not accessible to clients of this class, but are accessible to the object itself.

The default constructor has no arguments.
To create a more interesting *NewUFO* with additional characteristics and capabilities we need to think about what we actually want in our new object. So let’s assume we need a UFO with these additional capabilities and characteristics

- fuel — our *NewUFO* has a maximum fuel capacity. Fuel is used as the *NewUFO* moves but it can be refueled by bumping into a fuel cell in the window.
- shields
  - our *NewUFO* has a shield that can be brought into action a maximum number of times and each shield deployment lasts for a brief time period.
  - a deployed shield prevents damage when the *NewUFO* bumps into something but doesn’t prevent refueling.
  - the shields for multiple *NewUFOs* are linked together, so that when one deploys they all deploy.
movement – as long as our NewUFO has fuel it is able to move but it quits moving when it is out of fuel. This isn't consistent with real world physics of course. So what? It's a game after all.

10.1.1 Instance Variables

To implement these characteristics and capabilities each NewUFO needs to store information about itself—the visual characteristics as well as fuel and shield information. Instance variables store values unique to the particular object.

We already have the instance variables for the shapes that make the visual components of our NewUFO.

private GOval body, bubble, alien;

Next, a NewUFO has an amount of fuel remaining and the time the shields will need to come down (because shields only go up for a fixed amount of time), and we need to track both of these using instance variables.

private int fuel, shieldsDownTime;

We also need to keep track of the up or down state of the shields because that state will be used to determine what happens when the NewUFO collides with something. If the shield is up when collision occurs then the NewUFO will bounces backward. If the shield is down then something more drastic happens of course. Shields are initially down.

private boolean shieldsUp = false;

Private and Public Instance Variables and Encapsulation

All of these instance variables above are private, which means that may be referenced only by methods within the NewUFO class. private instance variables cannot be referred to by clients of the class. They are client accessible only if we provide methods for that purpose and thus they preserve the concept of encapsulation.

Instance variables may also be public, which means that they may be referenced directly by clients of the class, instead of by using a method. For example, if body was declared as

public GOval body;

and u was a NewUFO, then the client of this class (the game that uses u) could include the statement

u.body.setColor(Color.GREEN);
which says "apply the setColor method to the body of u". **public** instance variables are almost always a **bad idea** as they allow completely unfettered access which violates the concept of encapsulation.

### 10.1.2 Static Constants and Variables

**Static constants** represent fixed values that are **shared** by every instantiated object of a particular class.

Our first **static** constant represent the maximum amount of fuel that a *NewUFO* holds. We’ll use this to initialize the current **fuel** and to refill the tank when needed.

```java
private static final int FUEL_MAX = 1000;
```

The second **static** constant represents the amount of time, measured in milliseconds, that a shield can be up. In a bit we’ll program the shields so that after a shield has been on for this amount of time it automatically comes down.

```java
private static final int SHIELD_TIME = 500;
```

And our last **static** constant represents the maximum number of shield uses.

```java
private static final int MAX_SHIELDS_USES = 8;
```

Note that if there are *N* *NewUFOs* their shields are linked together, so that if shields are deployed this actually counts as *N* shield uses.

These constants are **private**, which means that their scope is limited to the class itself, the same as described for **private** instance variables. They are also **static**, which means that each constant is **shared** by all of the objects of the class. In this case, every *NewUFO* shares the same constants *FUEL_MAX*, *SHIELD_TIME* and *MAX_SHIELDS_USES*, which makes sense as certainly each *NewUFO* doesn’t need a separate copy of values that can’t change.

If we had left off the **static** keyword everything would work the same in this example, we’d just be wasting a little memory because every *NewUFO* would have its own copy of these values.

Finally recall that shields can be used only a certain number of times so clearly we need a variable to track the number of uses. Further all of the shields are linked, so this variable must be shared by all *NewUFOs* that we create. We need a **static** variable as shown here.

```java
private static int shieldUsedCnt = 0;
```

The `shieldUsedCnt` is initialized to 0 only when the class is first used in a run of the client program.
Static constants are also known as *class constants*, and static variables are also known as *class variables*.

### 10.1.3 Constructors

Our constructors are identical to those we used for the *UFO2* except that the constructor must initialize the *fuel* to the *FUEL_MAX* value.

### 10.1.4 Instance Methods

*Instance methods* are methods declared in a class and applied to an object in the form `object.method()`. For example, when we wrote `rect1.setColor(Color.RED)` we applied the instance method `setColor()` from the `GRect` class to the `rect1 GRect` object. When we create an object we generally need to create instance methods to manipulate the object.

#### Fuel Methods

For our *NewUFO* we need two methods related to fuel. The first returns the amount of fuel remaining and the second completely fills the fuel tank. All of these methods are *public* so that they may be used by clients of the class.

```java
public int getFuel() {
    return fuel;
}
```

This method is particularly useful for testing and debugging.

```java
public void refuel() {
    fuel = FUEL_MAX;
}
```

We’ll use this whenever we refuel our UFO.

#### Shield Methods

Next we need a method to raise the shields. This method needs the current game time as an argument because the shields will automatically come down a short time afterwards and the current game time is needed to calculate the time the shields will come down. And the shields can only be used *MAX_SHIELDS_USES* times and of course they only go up if they're not already up. So we have the following method.

```java
public void raiseShields(int currentTime) {
    if (shieldUsedCnt < MAX_SHIELDS_USES && shieldsUp == false) {
        shieldUsedCnt++;
        shieldsUp = true;
        shieldsDownTime = currentTime + SHIELD_TIME;
    }
}
```

//raiseShields

`raiseShields()` is *public* so that it is available to clients of the class.

We also need to check to see if the shields are up, and for that we need...
public boolean shieldsAreUp() {
    return shieldsUp;
}

Shields come down automatically, not in response to the client, and so the lowering method will be private—it cannot be called by the client. The shields come down if they've been up the right amount of time. We already set the time they need to come down when the shields last went up. And of course this happens only if the shields are already up.

private void shieldsDownIfNeeded(int currentTime) {
    if ((currentTime >= shieldsDownTime) && (shieldsUp == true))
        shieldsUp = false;
} //shieldsDownIfNeeded

For testing and debugging, if not for our final game, it is useful to be able to retrieve the number of times the shields have been used.

public int getShieldUses() {
    return shieldUsedCnt;
}

Moving the NewUFO
Moving the NewUFO is more complex and requires more discussion. There are several considerations

- movement can only happen if there's fuel of course, at least in our little model of the universe where there's no momentum
- if there is movement
  - the NewUFO burns a little fuel
  - the visual elements of the NewUFO must move on screen
- whether or not there is movement the shields are lowered if their time is up. Since lowering of the shields is independent of NewUFO movement it is not really a required part of the move() method. On the other hand, move() will execute with each pass through the game loop so it is convenient to check for shield lowering in the move() method.
- since the NewUFO may not actually move on the screen (out of fuel?) perhaps we really should name the method ‘act’ or perhaps ‘interactWithUniverse’, but we’ll stick with move.

We have

public void move(int x, int y, int currentTime) {
    if (fuel > 0)
    {
        move(x, y);
    }
fuel = fuel - Math.abs(x) - Math.abs(y);

} //move

lowerShields(currentTime);

Note that this move( ) method contains a call to another move( ) method. What’s going on? To answer that let’s begin by assuming u is a NewUFO. Then in a client of the NewUFO class the statement

u.move(someValueX, someValueY, someValueCurrentTime);

says ‘to the u object apply the NewUFO move method with three int arguments.’ Well, there is such a move( ) method; it’s the one written above. So that move( ) method gets applied to u.

Inside the move( ) method we see the statement

move(x, y);

First question—what are we moving? It’s not obvious because the statement isn’t in the usual object.move(x, y) form. When the object is not specified a method call refers to the current object, u in our discussion. This statement can also be written in a completely equivalent manner as

this.move(x, y);

The Java keyword this explicitly refers to the current object. Its use provides a way of stating very clearly ‘move this object’.

But what move is being applied? Well, u is a NewUFO and we haven’t and won’t define a move method for NewUFOs that has two int parameters. That’s not a problem however, as u is also a GCompound object (since NewUFO will extend GCompound) just like those we’ve worked with before. GCompounds have a move( ) with two int arguments and that move is the move that is being used. Thus u moves visually.

In short, Java applies the correct method to an object (NewUFO or GCompound) based on the class of the object.

Some Common Terminology
Instance methods that set the value of an object’s variables are known as mutators because they mutate (change) the object. Less formally they are called setters because they set characteristics of the object. In our example, the refuel( ), shieldsUpIfNeeded( ), shieldsDownIfNeeded( ) and move( ) methods are mutators/setters. Note that constructors are not considered to be mutators.
Instance methods that return the value of an object's variable are known as **accessors** because they access a value, and less formally as **getters**. `getFuel()`, `shieldsAreUp()` and `getshieldUses()` are accessors.

Our finished version of the `NewUFO` class is presented here.

```
NewUFO
//NewUFO.java
import acm.graphics.*;
import java.awt.*;

public class NewUFO extends GCompound {
    //*** private variables
    private GOval body, bubble, alien;
    private int fuel, shieldsDownTime;
    private boolean shieldsUp = false;

    //*** private constants
    private static final int FUEL_MAX = 1000;
    private static final int SHIELD_TIME = 500;
    private static final int MAX_SHIELDS_USES = 8;

    //*** private static variables
    private static int shieldUsedCnt = 0;

    //*** constructors
    public NewUFO( ) {
        body = new GOval(50, 25);
        body.setFilled(true);
        body.setColor(Color.RED);
        bubble = new GOval(25, 25);
        alien = new GOval(10, 10);
        alien.setFilled(true);
        alien.setColor(Color.GREEN);
        add(bubble, 13, 0);
        add(alien, 20, 5);
        add(body, 0, 13);
        fuel = FUEL_MAX;
    } //NewUFO()

    public NewUFO(Color bodyColor, Color alienColor) {
        body = new GOval(50, 25);
        body.setFilled(true);
    }
```
body.setColor(bodyColor);
bubble = new GOval(25, 25);
alien = new GOval(10, 10);
alien.setFilled(true);
alien.setColor(alienColor);
add(bubble, 13, 0);
add(alien, 20, 5);
add(body, 0, 13);
fuel = FUEL_MAX;

// NewUFO(Color, Color)

/*** fuel methods
public int getFuel()
{   return fuel;   }

public void refuel()
{   fuel = FUEL_MAX;   }

/*** shields methods
public void raiseShields(int currentTime)
{   if (shieldUsedCnt < MAX_SHIELDSUSES && shieldsUp == false)
    {
    shieldUsedCnt++;
    shieldsUp = true;
    shieldsDownTime = currentTime + SHIELD_TIME;
    }
} // raiseShields

public boolean shieldsAreUp()
{   return shieldsUp;   }

private void lowerShields(int currentTime)
{   if ((currentTime == shieldsDownTime) && (shieldsUp == true))
    shieldsUp = false;
} // lowerShields

public int getShieldUses()
{   return shieldUsedCnt;   }

/*** move methods
public void move(int x, int y, int currentTime)
{   if (fuel > 0)
    {
    move(x, y);
}
\[
\text{fuel} = \text{fuel} - \text{Math.abs}(x) - \text{Math.abs}(y);
\]

//move

Testing the NewUFO
The NewUFO may eventually be part of a game, but we should do some testing before we get started on that project. We need to check

- does it move correctly and does moving use fuel and automatically bring the shields down if needed?
- do the shields raise properly?
- can we refuel the UFO?

And because some behaviors depend on the existence of multiple NewUFOs (shields go up and down together) we'll need to instantiate at least two ufos. We'll control one of the ufos with the arrows keys and the other with the E, X, S and D keys.

We’ll write a game-like activity that will allow us to test our NewUFO object. Here’s the logic of our testing program.

\[
\begin{align*}
\text{initialization method} \\
\{ \\
\quad \text{set up the two ufos} \\
\quad \text{set up a bomb to test shields} \\
\quad \text{set up a fuel cell to test refueling} \\
\} \\
\text{keyPressed method} \\
\{ \\
\quad \text{set movement for each ufo using the arrows keys and E-X-S-D keys} \\
\quad \text{raise shields with space key} \\
\} \\
\text{run method} \\
\{ \\
\quad \text{while (running the test program)} \\
\quad \{ \\
\quad \quad \text{if (ufo1 hits bomb)} \\
\quad \quad \{ \\
\quad \quad \quad \text{if (shields are up)} \\
\quad \quad \quad \quad \text{set movement so that ufo1 bounces backward} \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad \text{test is over} \\
\quad \quad \} \\
\quad \quad \} \\
\quad \} \\
\} \\
\end{align*}
\]
display collision without shields message
}
}  
else if (ufo2 hits bomb)
{
    same as for ufo1
}
else if (the ufo1 hits a fuel cell)
{
    refill ufo1
    remove the fuel cell from the game
    display refueled message
}
move ufo1
move ufo2
display the current fuel level for both ufos
display shields up or shields down message for both ufos
}
}

Our test program below implements this logic. If the NewUFO passes this test then we have good reason to believe that it will behave as expected in a game.

TestNewUfo

//TestNewUFO.java
import acm.program.*;
import acm.graphics.*;
import java.awt.event.*;
import java.awt.*;

public class TestNewUFO extends GraphicsProgram
{
    final int PAUSE = 50;
    GRect bomb;
    GOval fuelCell;
    GLabel fuelSign1, fuelSign2, shieldUses;
    NewUFO u1, u2;
    int u1XVel, u1YVel, u2XVel, u2YVel, time;
    boolean exit;

    public void init()
    {
        u1 = new NewUFO();
        add(u1, 0, 200);
        u2 = new NewUFO(Color.BLUE, Color.RED);
add(u2, 0, 250);

bomb = new GRect(100, 100);
bomb.setColor(Color.RED);
bomb.setFilled(true);
add(bomb, 300, 200);

fuelCell = new GOval(30, 30);
fuelCell.setColor(Color.GREEN);
fuelCell.setFilled(true);
add(fuelCell, 400, 300);

fuelSign1 = new GLabel("u1 fuel: "+u1.getFuel( ));
add(fuelSign1, 10, 20);
fuelSign2 = new GLabel("u2 fuel: "+u2.getFuel( ));
add(fuelSign2, 10, 460);

shieldUses = new GLabel("shield uses: "+u1.getShieldUses( ));
add(shieldUses, 650, 20);

u1XVel = u1YVel = u2XVel = u2YVel = time = 0;
exit = false;

addKeyListeners( );
waitForClick( );

} //init

public void keyPressed(KeyEvent e)
{
    int key = e.getKeyCode( );

    //u1 direction changes
    if (key == KeyEvent.VK_UP) u1YVel = u1YVel - 1;
    else if (key == KeyEvent.VK_DOWN) u1YVel = u1YVel + 1;
    else if (key == KeyEvent.VK_RIGHT) u1XVel = u1XVel + 1;
    else if (key == KeyEvent.VK_LEFT) u1XVel = u1XVel - 1 ;

    //u2 direction changes
    else if (key == KeyEvent.VK_E) u2YVel = u2YVel - 1;
    else if (key == KeyEvent.VK_X) u2YVel = u2YVel + 1;
    else if (key == KeyEvent.VK_D) u2XVel = u2XVel + 1;
    else if (key == KeyEvent.VK_S) u2XVel = u2XVel - 1;

    //raise the shields
    else if (key == KeyEvent.VK_SPACE)
    {
        u1.raiseShields(time);
        u2.raiseShields(time);
        shieldUses.setLabel("shield uses: " + u1.getShieldUses( ));
    }
public void run()
{
    while(!exit) //game loop
    {
        time = time + PAUSE;
        pause(PAUSE);
        if (collides(u1, bomb) == true)
        {
            if (u1.shieldsAreUp() == true)
            {
                u1XVel = -u1XVel;
                u1YVel = -u1YVel;
            }
            else
            {
                exit = true;
                println("Collision without shields");
            }
        }
        else if (collides(u2, bomb) == true)
        {
            if (u2.shieldsAreUp() == true)
            {
                u2XVel = -u2XVel;
                u2YVel = -u2YVel;
            }
            else
            {
                exit = true;
                println("Collision without shields");
            }
        }
        else if ((fuelCell != null) && (collides(u1, fuelCell) == true))
        {
            u1.refuel();
            remove(fuelCell);
            fuelCell = null;
            println("refueled");
        }
        else if ((fuelCell != null) && (collides(u2, fuelCell) == true))
        {
            u2.refuel();
            remove(fuelCell);
        }
    }
```java
fuelCell = null;
println("refueled");
}

u1.move(u1XVel, u1YVel, time);
u2.move(u2XVel, u2YVel, time);
fuelSign1.setLabel("u1 fuel: "+u1.getFuel());
fuelSign2.setLabel("u2 fuel: "+u2.getFuel());
if (u1.shieldsAreUp( ))
    println("u1 shields up");
else
    println("u1 shields down");
if (u2.shieldsAreUp( ))
    println("u2 shields up");
else
    println("u2 shields down");
} //game loop
println(u1);
println(u2);
} //run

public boolean collides(GObject ufo, GObject x)
{
    GRectangle ufoBox = ufo.getBounds( );
    GRectangle xBox = x.getBounds( );
    if (ufoBox.intersects(xBox) == true) return true;
    else return false;
} //collides
} //TestNewUFO
```

Spend some time flying the NewUFO around. Run it over the bomb with and without shields up. Remember that the shields only work a few times. Run the NewUFO over the fuel cell to see if the fuel recharges. Does it quit moving after it runs out of fuel? Use it in a game only when it has passed this testing program.

About that println(u1) and println(u2) statements—if you run the testing program and look at the console window you’ll see that the println( ) displays basic location information about the NewUFO when the loop ends. We’ll explain how this works after we’ve talked about the toString( ) method in the next section.

### 10.2 Data Objects

The term data object describes a programmer created object whose purpose is to hold information and provide methods for manipulating that information. A data object does not
have a significant visual component, unlike a graphic object. Since we've already laid a firm foundation by studying graphic objects we'll jump right into an example.

## 10.2.1 Complex Numbers Example

Recall that a complex number is a number in the form \( a + bi \), where \( a \) and \( b \) are real numbers (such as 4, -3.15 and so on), \( i = \sqrt{-1} \) and that we can perform arithmetic with these numbers. We shall refer to the \( a \) coefficient as 'the real part' and the \( b \) coefficient as 'the imaginary part', even if these terms make a mathematician somewhat ill.

Complex numbers are extremely useful in some areas of science and engineering, so it might be useful to have a class that represents them and the arithmetic operations we might wish to perform.

Let's consider what we want to be able to do with a complex number and the methods will need to perform these tasks.

- **constructors** – a default complex number and a specified complex number
- **accessors** – to retrieve the real and imaginary parts
- **mutators** – to set the real and imaginary parts
- **mathematics methods** – for addition, subtraction, multiplication

Let us consider two complex numbers: \( a + bi \) and \( c + di \). Then we define:

- **addition**: \( (a+bi) + (c+di) = (a+c) + (b+d)i \)
- **subtraction**: \( (a+bi) - (c+di) = (a-c) + (b-d)i \)
- **multiplication**: \( (a+bi) \cdot (c+di) = (ac - bd) + (a \cdot d + b \cdot c)i \)

We'll skip division in this discussion to save a little time and space. Dig out the math if you are interested.

- **conjugate** – the conjugate of \( a + bi \), denoted \( \bar{a + bi} \), is defined as \( a - bi \). So 2 - 4.2i and is -6 + 3i.
- **determine if a complex number has a 0 real part** and determine if a complex number has a 0 imaginary part. For example, The complex number 2 + 0i is also the real number 2, and the complex number 0 – 6i is also the imaginary number -6i.
- **test two complex numbers for equality**
- **return a String representation of a complex number**. We’ll want this to see the result of all the other operations.

The `Complex` class is implemented below.

```java
//Complex.java
import acm.program.*;

public class Complex
{
    private double realPart, imagPart;
}
```
//***** constructors
public Complex()
{
    //default constructor, constructs 0 + 0i
    //example: Complex c = new Complex();
    realPart = 0;
    imagPart = 0;
} //Complex

public Complex(double a, double b)
{
    //non-default constructor, constructs a representation of a + bi
    //example: Complex c = new Complex(3, -4);
    realPart = a;
    imagPart = b;
} //Complex

//***** basic accessors/getters
public double getReal()
{
    //example: double p = c.getReal();
    return realPart;
} //getReal

public double getImag()
{
    //example: double m = c.getImag();
    return imagPart;
} //getImag

//***** basic mutators/setters
public void setReal(double a)
{
    //set the real part of a complex number
    //example: c1.setReal(3.43);
    realPart = a;
} //setReal

public void setImag(double b)
{
    //set the imaginary part of a complex number
    //example: c1.setImag(-1.2);
    imagPart = b;
} //setImag

public void set(double a, double b)
{
    //set both real and imag part of a complex number
    //example: c1.set(-4.5, 17);
    realPart = a;
    imagPart = b;
} //set
//***** arithmetic methods

public void add(Complex x)
{
    //add x to current object
    //example: c1.add(c2);
    realPart = realPart + x.realPart;
    imagPart = imagPart + x.imagPart;
}

public void sub(Complex x)
{
    //subtract x from current object
    //example: c1.sub(c2);
    realPart = realPart - x.realPart;
    imagPart = imagPart - x.imagPart;
}

public void mult(Complex x)
{
    //multiply current object by x
    //example: c1.mult(c2);
    double temp = realPart;
    realPart = realPart*x.realPart - imagPart*x.imagPart;
    imagPart = temp*x.imagPart + imagPart*x.realPart;
}

public void conjugate()
{
    //transforms a complex to its conjugate: 2+3i becomes 2-3i
    //example: c1.conjugate();
    setImag(-imagPart);
}

//***** utility methods

public boolean isReal()
{
    //if the complex number has a 0 coefficient of I, 2 + 0i for example,
    //return true, otherwise return false
    //example: if (c.isReal() == true) doSomething; else doSomethingElse;
    if (imagPart == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

public boolean isImag()
{
    //if the complex number has a 0 real value, 0 - .13i for example,
    //return true, otherwise return false
    //example: if (c.isImag() == true) doSomething; else doSomethingElse;
    if (realPart == 0)
    {
        return true;
    }
    else
Most of this class is very straightforward but some methods need discussion.

The equals( ) Method
It is very useful to be able to compare two objects, such as Complex numbers, for equality, but what ‘equal’ means is a little deeper than you might expect.

The == operator is one way to compare for equality. == compares the contents of two variable’s memory locations and for a primitive data type this works perfectly. But for an object a variable’s memory location contains, not the value, but rather the memory address where the value is stored. So if x and y are objects, the comparison x == y compares the memory locations, not the values. And that’s not what we want. We need a more meaningful way to compare objects for equality and we do that with the equals( ) method.

Every object we create, including the Complex object, is automatically a subclass of the standard Java Object class and that class contains an equals( ) method. If we don’t define our own equals( ) method in the Complex class then the Object equals( ) will be used. Unfortunately this method uses == for comparison, and as we said before that’s not much help with objects.

Almost every class we create should include its own equals( ) method. Further, other standard Java classes such as ArrayList depend on it being in the form

```java
public boolean equals(objectType)
```
so that’s the form we'll always use. For the Complex class we test two complex numbers for
equality by testing the real parts for equality and the imaginary parts for equality.

```java
public boolean equals(Complex x) {
    //example: if (c1.equals(c2) == true) ...
    if (realPart == x.realPart && imagPart == x.imagPart) {
        return true;
    } else {
        return false;
    }
} //equals
```

We did not define an equals() method for the NewUFO class because it just didn’t seem
needed for anticipated gaming applications.

The toString() Method
The toString() method deserves comment. The goal of the toString() method is to construct
and return a text representation of the object. Let us say that we have a complex number \( c \)
with the value of 2 – 5.5i. Then what is stored in memory is

<table>
<thead>
<tr>
<th>realPart</th>
<th>imagPart</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>-5.5</td>
</tr>
</tbody>
</table>

which is good enough for the computer but not good enough for humans, who expect a little
more. A good String representation is ‘2 - 5.5i’, with the i required to represent the \( i \) to the
human viewer. That’s what our toString() method above constructs and returns. We’ll put it to
use shortly.

### 10.2.2 Testing the Complex Class
Here's a very small testing program. It's just enough to demonstrate some of the basic
functionality of the Complex class.

**TestComplex1**
```java
//TestComplex1.java
import acm.program.*;

public class TestComplex1 extends ConsoleProgram {
    public void run() {
        Complex c1 = new Complex(4, 5);
        Complex c2 = new Complex();
        Complex c3 = new Complex(-1, -12);
    }
}
```
println("*** constructor and toString test");
println("Expected c1: 4 + 5i");
println("Actual c1: " + c1.toString());
println("Expected c2: 0 + 0i");
println("Actual c2: " + c2);
println("Expected c3: -1 - 12i");
println("Actual c3: " + c3);
} //run
} //TestComplex1

Many of the println( ) statements output a Complex number, which is an object that we’ve created, not a standard Java object or primitive data type. So there’s no reason to believe that Java knows how to output one. That’s why we first have

println(“Actual c1: “ + c1.toString());

The toString( ) method constructs a String representation of c1, returns it, and the println( ) outputs this String. println( ) does know how to output Strings; we’ve been doing it since day one.

But what about

println("Actual c2: " + c2);

This doesn’t call the toString( ) method, so what happens? In Java, when an object needs to be treated as a String, such as on output, the toString( ) method is automatically invoked, so the above statement executes exactly as if we have written it with the explicit c2.toString( ) in the argument list.

What happens if the toString( ) is automatically called and there is no defined toString( ) method? The answer is that every object we create is automatically a subclass of the Java Object class, which has a toString( ) method. So if we don’t supply one for our class the Object toString( ) method is used. Unfortunately the String returned by this method includes the class name and a usually useless hexadecimal number. For example we might have Complex@15b7986. The lesson to be learned is to write your toString( ) for any class you create.

Now recall println(u1); from the TestNewUFO program. There is no toString( ) method for the NewUFO object. But that’s not a problem because u1 is a GCompound (it extends GCompound), and a GCompound is a GObject. There is a toString( ) method for GObjects, and it returns the location of the GObject as a String, so that’s what we get when we println(u1).

At last, let’s look at the output of our test program.
As the expected and actual results match (minor formatting differences aside) we conclude that the constructors and the toString( ) method work properly. The formatting differences are unimportant for our purposes but if you are interested in fixing them look into the NumberFormat class.

Now that we have some indication that the basics work let’s look at a more complete, comprehensive test program. The basic idea is to test each method as much as possible completely independently of other methods. Not everything is tested though; the rest is left as an exercise.

TestComplex2

//TestComplex2.java
import acm.program.*;

public class TestComplex2 extends ConsoleProgram
{
    public void run()
    {
        Complex c1 = new Complex(4, 5);
        Complex c2 = new Complex();
        Complex c3 = new Complex(-1, -12);

        println("*** constructor and toString test");
        println("Expected c1: 4 + 5i");
        println("Actual c1: 4.0 + 5.0i");
        println("Expected c2: 0 + 0i");
        println("Actual c2: 0.0 + 0.0i");
        println("Expected c3: -1 - 12i");
        println("Actual c3: -1.0 - 12.0i");

        println("*** getReal & getImag test");
        println("Expected c1: " + 4 + " + " + 5);
        println("Actual c1: " + c1.getReal() + " + " + c1.getImag());
        println();

        println("*** setReal & setImag test");
    }
}
c1.setReal(12);
c1.setImag(-10);
println("Expected c1: 12 - 10i");
println("Actual c1: " + c1);
println();

println("*** add test");
c1.add(c3);
println("Expected c1: 11 - 22i");
println("Actual c1: " + c1);
println();

println("*** isReal test");
c1.setImag(0);
println("c1: " + c1);
println("Expected: true");
println("Actual: " + c1.isReal( ));
c1.setImag(3.3);
println("c1: " + c1);
println("Expected: false");
println("Actual: " + c1.isReal( ));
println();

} //run
} //TestComplex2

And here's the corresponding output of the test.

*** constructor and toString test
  Expected: 4 + 5i
  Actual c1: 4.0 + 5.0i
  Expected: 0 + 0i
  Actual c2: 0.0 + 0.0i
  Expected: -1 - 12i
  Actual c3: -1.0 - 12.0i

*** getReal & getImag test
  Expected c1: 4   5
  Actual c1: 4.0   5.0

*** setReal & setImag test
  Expected c1: 12 - 10i
  Actual c1: 12.0 - 10.0i

*** add test
  Expected c1: 11 - 22i
Comparison of the expected and actual results verify that the testing has been passed. If we finish this test suite and generate a print we have evidence that the class works correctly. Careful testing is absolutely essential before mission critical software is deployed.

10.3 More About Objects

Java is a very expressive language with a lot of power and we’ll look at just a little bit more of this now.

10.3.1 Copy Constructors

Let’s go back to the Complex class and assume \( x_1 \) is a Complex with the value 2 + 3i. Now suppose we want to make a copy of \( x_1 \).

Here’s a first attempt.

\[
\text{Complex } x_2 = x_1; \quad \text{WRONG}
\]

Why is it wrong? Because \( x_1 \) is a Complex object (not a primitive data type) it does not contain the values 2 and 3. \( x_1 \) actually contains the memory address where the 2 and the 3 are stored. And the assignment statement copies this address value into \( x_2 \). The result is that \( x_1 \) and \( x_2 \) both point to the memory containing 2+ 3i. We have

\[
\begin{array}{ccc}
\text{x1} & \rightarrow & 2+3i \\
\text{x2} & \rightarrow & \text{WRONG}
\end{array}
\]

When a variable points to the same object as another variable, it is an alias. An alias is very often a bad thing and always causes confusion. Avoid them like the plague.

The proper way to make a copy of an object is with a copy constructor, which makes a completely different copy in a new memory location. Copy constructors are written in the form

\[
\text{public objectName(objectName localVariableName);} \]

The copy constructor for the Complex class is
public Complex (Complex x)
{
    realPart = x.realPart;
    imagPart = x.imagPart;
} //Complex(Complex)

and it’s used like this

Complex x1 = new Complex(x2);

x1 and x2 are completely separate objects in different memory locations.

Things can get more difficult with copy constructors when the instance variables are objects. For example, let's consider a UFO2 object as discussed before, with a body, a bubble and an alien, all GOvals. Our first attempt at a copy constructor might be

public UFO2(UFO2 u1)
{
    body = u.body;
    bubble = u.bubble;
    alien = u.alien;
} //UFO2(UFO2)

UFO u2 = new UFO(u1);

Why is this wrong? Because the components themselves are objects, so all we've done is create an alias for the body of u1, an alias for the bubble of u1 and an alias for the alien of u1 and put those aliases in u2. u2 is not a completely independent copy of u1. What we actually have is

The (almost) correct way is

public UFO2(UFO2 u)
{
    body = new GOval(u.body);
    bubble = new GOval(u.bubble);
    alien = new GOval(u.alien);
} //UFO2(UFO2)
It's almost perfect. Each component uses the copy constructor of the \texttt{GOval} class to make a \texttt{new GOval}, exactly the same as the original, but in a new memory location. The idea is exactly correct. \textbf{Except that there is no copy constructor} for the \texttt{GOval} class or the other ACM Java graphics objects. So we're out of luck this time, but keep the idea in mind.

Also note that copy constructors and their cousin cloning can get very, very ugly. It's a deep subject and we didn't bring a shovel so let's get out of here before somebody gets hurt.

\subsection*{10.3.2 Garbage}
Assume this statement executes.

\begin{verbatim}
Complex r1 = new Complex (2, 3);
\end{verbatim}

We have

\begin{center}
\text{r1} \quad \text{2+3i}
\end{center}

and now this statement, executing just after the above statements executes.

\begin{verbatim}
r1 = new Complex ( 4, 5);
\end{verbatim}

We have

\begin{center}
\text{r1} \quad \text{4 + 5i}
\end{center}

\text{r1} used to point to the memory location where 2 + 3i is stored, but now it has been reassigned to point to the memory location where 4 + 5i is stored.

All fine and good but what happened to the 2 + 3i memory and the 2 + 3i value? Did it disappear? Has it been unallocated? Or is it just sitting there in memory?

The answer is that the memory location and the value are 'just sitting there' but we can't get to them. We can't because \texttt{r1} is the only thing that knew where 2 + 3i was stored, and it doesn't know that anymore due to the second assignment statement. The 2 + 3i is still occupying memory, getting in the way, causing the program to trip and fall, and generally being undersirable.

Objects that have been allocated but have no pointers to them are lost to the program. They are known as \textit{garbage}. Garbage wastes memory and causes bugs, but Java runs a garbage
collection program to automatically reclaim this memory, in effect picking up the garbage. Really!

10.3.3 Static Methods
Sometimes Java is more than a little obtuse. Think back to our Complex class. Expressing the mathematical operation \( c_1 = c_1 + c_2 \) as

\[
c1.add(c2);
\]

is accurate but ugly. Expressing \( c_1 = c_2 + c_3 \) as

\[
c1 = \text{new Complex}();
c1.add(c2);
c1.add(c3);
\]

is clumsy to say the least. Let's learn a more naturally expressive way.

Begin by defining a new version of the \( \text{add}() \) method.

```java
public static Complex add(Complex x, Complex y) {
    Complex temp = \text{new Complex}();
    temp.realPart = x.realPart + y.realPart;
    temp.imagPart = x.imagPart + y.imagPart;
    return temp;
}
```

Note the static modifier. It identifies this method as not attached to an instance of a Complex object. This is not an instance method, it is a static method and belongs to the whole class, much as static constants and variables were shared by every object in a class. static methods are also known as class methods.

We use our new \( \text{add}() \) method like this.

```java
Complex c3 = Complex.add(c1, c2);
```

which says \( c_3 \) is assigned the result of applying the \( \text{add} \) method from the Complex class to the \( c_1 \) and \( c_2 \) objects. This is exactly how we used the methods in the Math classes.

static methods allow us to write statements that are sometimes more expressive than if written with instance methods. And at times it's very useful to create an entire class of static methods and that's what we'll consider next.
10.4 Static Classes

Recall the Java Math class, which contains two constants PI and E, as well as many methods for math functions such as sin, cos, absolute value, ... We use these by writing Math.PI, Math.sin(angle), Math.abs(number), etc.

Unlike other objects, you can't make a Math object. The statement Math m = new Math(); results in a compilation error.

In Java an object like this is a static object. A static object may be thought of as a machine for storing constants and performing operations.

Below is a static class Math2, created to stash some useful methods and a useful constant in.

Math2

//Math2.java

class Math2 {

    public static final double LN10 = Math.log(10);

    private Math2() {

    }

    public static double square(double b) {
        return b*b;
    }

    public static double cube(double b) {
        return b*b*b;
    }

    public static double fourth(double b) {
        return square(b)*square(b);
    }

    public static double cubeRoot(double b) {
        return Math.pow(b, .3333333);
    }

    public static double fourthRoot(double b) {
        return Math.sqrt(b)*Math.sqrt(b);
    }

}

The constructor is worth some discussion.

private Math2() {
    
};
The **private** access modifier makes the constructor inaccessible to clients. The *Math2* class therefore has no usable constructor and that prevents instantiation of *Math2* objects. A client *can't* do the following.

```java
Math2 m = new Math2();  // WRONG!
```

Using the *Math2* library is straightforward.

```java
println(Math2.LN10);
println(Math2.cubeRoot(10));
double z = Math2.cubeRoot(4+t) - Math2.fourthRoot(p);
```

Once you've got the idea it is easy to imagine many uses for **static** classes. For example, a business application might need a **static** class of business methods for calculating interest rates, future values, annuity payments, etc. A mailing application might use a **static** class of methods for formatting the name and address information in a mailing label. **static** classes can be very useful.

## 10.5 More UML

In Chapter 5 we introduced basic objects and the UML to describe them. Now that we've covered more involved objects so it's time to dive a little deeper into UML.

Recall that a simple UML diagram consists of a three cell table layed out as shown here.

<table>
<thead>
<tr>
<th><strong>name of the object</strong></th>
<th><strong>instance variables of the object</strong></th>
<th><strong>methods of the object</strong></th>
</tr>
</thead>
</table>

The UML diagram for the *UFO2* object we created earlier is thus.

```
UFO2
body : GOVal
bubble : GOVal
alien : GOval
<<constructor>> UFO()
<<constructor>> UFO(Color, Color)
```

UML uses additional notation to convey further information.
- a leading + indicates that a constant, variable or method is **public**
- a leading − indicates that a constant, variable or method is **private**
- underlining indicates that a constant, variable or method is **static**
Using this notation the original *UFO2* would be shown as

```
UFO2
  - body : GOval
  - bubble : GOval
  - alien : GOval
+ <<constructor>> UFO2()
+ <<constructor>> UFO2(Color, Color)
```

and the UML for the *NewUFO* class is

```
NewUFO
  - body : GOval
  - bubble : GOval
  - alien : GOval
  - fuel : int
  - shieldsDownTime : int
  - shieldsUp : boolean
  - FUEL_MAX : int
  - SHIELD_TIME : int
  - MAX_SHIELDS_USES : int
  - shieldUsedCnt : int
  + <<constructor>> NewUFO()
  + <<constructor>> NewUFO(Color, Color)
  + getFuel() : int
  + refuel()
  + raiseShields(int)
  + shieldsAreUp() : boolean
  - lowerShields(int)
  + getShieldUses() : int
  + move(int, int, int)
```

UML can be used for much more than describing classes. It can also represent the relationship between objects as well as sequence of operations but that will be deferred for another time.

## Problem Set

**Die** Below is a partial implementation of a class for representing a six-sided die as well as a short testing program. Finish the incomplete methods and make sure it works correctly by compiling and running it.

```java
//Die.java
import acm.program.*;
```
import acm.util.*;

public class Die extends ConsoleProgram {
    private RandomGenerator rg = new RandomGenerator();
    private int value;

    public Die() {
        value = rg.nextInt(1,6);
    }

    public Die(int n) {
        if (n < 1 || n > 6) { n = 1; }
        else { value = n; }
    }

    public void roll() { //value is assigned a random from 1 to 6
        ********
    }

    public int getValue() { //return the value
        return ********;
    }

    public String toString() {
        return "" + value;
    }
}

//TestDie.java
import acm.program.*;

public class TestDie extends ConsoleProgram {
    public void run() {
        Die d1 = new Die();
        println("D1 value: " + d1);
        println();

        Die d2 = new Die();
        print("D2 values: ");
        for(int i=0; i<10; i++)
            {print(d2 + " ");
             d2.roll();
             }
        println(); println();
    }
}
Die d3 = new Die(4);
println("D3 value: " + d3);
println();
println("The sum of D1 and D3 is: " + d1.getValue() + d3.getValue());
} //run

Date – write a simple representation of a Date, consisting of
• day of the month – an int
• month – a String
• year – an int
• methods – Date(), Date(month, day, year), setDate(month, day, year), toString(),
  setDay(day), getDay()
• in a separate program, TestDate, write a testing suite that tests all of these methods

CompleteComplexTesting – the TestComplex2 above is incomplete. Finish it so that all methods
are tested.

Rational - using the Complex class as an example, develop a class for representing rational
numbers. Also write a separate testing program TestRational, similar to the testing for the
Complex class. You'll need instance variables for the numerator and denominator, as well as
these methods
• default constructor - is 0/0 okay?
• non-default constructor - Rational(n, d)
• setNum
• setDen
• add
• sub
• mult
• divide
• isZero - is the rational number equal to 0?
• toString
• equals

BugBot2 – recall the BugBotWithMethods assignment from chapter 8. Modify the BugBot
object so that it changes appearance in a way that indicates the direction it is heading. For
example, you might have

and a left turn would result in

and so on.
**LunarLander** – create a lunar lander that you can fly around the window. The lander looks something like this:

The lander is controlled by left, bottom and right thrusters which visually appear on the lander when the appropriate arrow key is pressed. The lander descends under the force of gravity and its speed accelerates due to gravity. Momentum is preserved.

A thruster is fired by pressing one of the left, right or up arrow keys. When a thruster is on it remains on until another thruster key is pressed. If the up/left/right thrust is on and the up/left/right arrow key is pressed then that thruster turns off. The lander has a finite amount of fuel which is burned as a thruster is fires. Once the fuel is empty the lander falls under the force of gravity.

The space key activates an emergency thruster that boosts the lander straight up with twice the force as the usual bottom thruster.

Write a testing program (TestLunarLander) that allows you to fly the lander around the window.

**LunarLanderGame** – write a game program in which you fly your lunar lander around a window, earning points for every safe landing you make. A safe landing occurs when the down velocity is appropriately small and both legs of the lander are on the building.

The flying environment looks something like this.
The emergency thruster can be pressed only once in a round of play.

Create a game with three rounds. A round ends when either fuel runs out or the lander has landed on all of the shapes. There is a limited amount of time for each round, and that amount of time gets smaller each round.

At the end of the game display a message that rates the piloting. You might perhaps use:
- 0 - 5 points → Not too good for a droid!
- 6-10 points → Not bad for a wookie!
- 10-15 points → I didn't recognize you Han Solo
- 16-20 points → Luke, the force is with you.

**Quadratic** - below is the shell of a static class for calculating roots of a quadratic equation and a complete TestQuadratic. Fill in the Quadratic methods and test your Quadratic class with the TestQuadratic program.

//Quadratic.java
public class Quadratic
{
    private Quadratic()
    {
    
    }

    private static double calcDiscriminant(double a, double b, double c)
    {
    }

    public static boolean hasRealRoots(double a, double b, double c)
    {
    }

    public static double calcRoot1(double a, double b, double c)
    {
    }

    public static double calcRoot2(double a, double b, double c)
    {
    }
} //Quadratic

//TestQuadratic.java
import acm.program.*;

public class TestQuadratic extends ConsoleProgram
{
}
public void run() {
    println("*** hasRealRoots test");
    println("Expected: true");
    println("Actual: " + Quadratic.hasRealRoots(1, 5, 6));
    println("Expected: true");
    println("Actual: " + Quadratic.hasRealRoots(4, 0, -9));
    println("Expected: false");
    println("Actual: " + Quadratic.hasRealRoots(3, 4, 5));
    println("Expected: false");
    println("Actual: " + Quadratic.hasRealRoots(1, 0, 6));
    println();

    println("*** root1 and root2 test");
    println("Expected: -2  -3");
    println("Actual: " + Quadratic.calcRoot1(1, 5, 6) + " "
            + Quadratic.calcRoot2(1, 5, 6));
    println("Expected: 1.5  -1.5");
    println("Actual: " + Quadratic.calcRoot1(4, 0, -9) + " "
            + Quadratic.calcRoot2(4, 0, -9));
}