Chapter 5
Methods & Objects

As our programs grow in size and complexity we need more powerful tools to tackle tougher problems. In this chapter we'll take a quick look at methods and objects, both important tools in the fight against code madness. In later chapters we'll visit them again for a much more in depth look.

5.1 Introduction to Methods

To this point our programs have typically been fairly small, perhaps a few dozen lines. Small programs like these are typically simple to design, code, debug and maintain.

In the real world programs are much longer—often 50,000 or perhaps even a million lines of code, written by teams of designers and programmers and taking months or years to develop. They may be used for years and be modified many times, fixing bugs or adding features. The complexity can be overwhelming.

One tool for dealing with complexity in software is the method. A method is basically lines of code, grouped together and given a name. Methods allow us to take a large programming job and cut it up into bite sized tasks. Note that the terms function, procedure or module are sometimes used also in other programming languages.

In this chapter we'll start with the simplest methods.

5.1.1 void Methods without Arguments

Consider the following example which uses a very simple void method.

```
import acm.program.*;

public class HelloWorldWithAMethod extends ConsoleProgram
{
    public void run()
    {
        println("Jumping to displayHello method"); //println #1
        displayHello();  //method call
    }
```

©Daniel L. Schuster, 2009, all rights reserved
println("Back from displayHello method");  //println #3
}  //run

public void displayHello()  //definition of method
{
    println("Hello world");  //println #2
}  //displayHello

}  //HelloWorldWithAMethod

Here's a trace of the path of execution.

1) execution begins at the run method.
2) the first println is executed.
3) the displayHello() method call is encountered and execution jumps to the beginning of displayHello().
4) displayHello() executes, and so the second println statement is executed.
5) when the end of displayHello() (a right brace '}') is encountered, execution jumps back to the run method, and continues at the statement immediately after the displayHello() method call.
6) the third println is executed.
7) the end of the run method is encountered and the program ends.

The output is

Jumping to displayHello method
Hello world
Back from displayHello method

Defining a method
The method displayHello() is a model for defining a simple void method and includes all of the fundamental parts of such a method.

public void displayHello()  //method header
{
    println("Hello world");  //println #2
}  //displayHello

Let's consider the first line of the method, called the method header, and examine each of its parts.

public void displayHello()  //access modifier
The access modifier describes the accessibility of the method. For now we will always use public. Later we'll learn what this means and also about the access modifier private.

**public void displayHello()**

The return type tells us the data type (such as int, boolean, etc.) or class type (such as GOval, String, etc.) that is returned by the method. In this case, the displayHello() method just executes and doesn't return anything. We denote this with void. More about this later.

**public void displayHello()**

The method name is a programmer declared identifier that names the method. It should describe the task or effect of the method.

**public void displayHello()**

And finally, in this case the parenthesis are empty, but they still must be there.

Now let's look at the rest of the method, the **method body**.

```java
{ 
    println("Hello world"); //println #2
} //displayHello
```

The method body begins and ends with braces. It contains Java statements. Generally the method body performs one task or a very closely related set of tasks. It can contain any number of statements, but is normally fairly short, perhaps no more than 25 statements or so and often just a few.

**Calling a void method**

Defining a method does not execute the method. To do that we need a method call. The method call

```java
displayHello();
```

is a model for calling a void method. Such a call consists of

1) the name of the method, exactly as written in the method definition.
2) the ( ), even though empty. The empty parenthesis indicates that the method call is not sending any information to or receiving any information from the displayHello method.

Methods like this are commonly used in games to display information to the user. For example, we might give game instructions as shown below.

```java
public void run()
{
    displayGameInstructions();
    ...game code...
} //run

public void displayGameInstructions()
{
    GLabel line1 = new GLabel("How to Play Weasel Hunt");
    line1.setFont("Serif-PLAIN-25");
    add(line1, 10, 30);
    ...more instructions...
} //displayGameInstructions
```

A method can be called as many times as necessary. If the same instructions must be displayed at two different levels of a game we might have:

```java
public void run()
{
    //level 1
    displayGameInstructions();
    ...level 1 game code...
    //level2
    displayGameInstructions();
    ... level 2 game code ...
} //run
```

So what is a void method?
That's easy. A `void` method is simply a method that returns no value.

To illustrate this, let's first look at a method that *isn't* a `void` method. Recall from Chapter 2 the `getX()` method. We might use it like this.

```java
int xLoc = (int) rect1.getX();
```

`getX()` returns a `double` representing the x coordinate of `rect1`, the `int` type cast converts that `double` to an `int` and then the `int` is assigned to `XLoc`. The important point is that the `getX()` method *returns* a value which we use in an assignment statement.
A **void** method does not return anything. We call the method, execution jumps to the method and then continues after the call.

... *code executed before the method call*

displayGameInstruction( );

... *more code executed after the method call*

There is no assignment. There is nothing to assign as the method just does its work, not returning any value.

### 5.1.2 void Methods with Arguments

Consider again the call

displayGameInstruction( );

This method behaves exactly the same every time, which is not very flexible. To gain flexibility we need to add **arguments**, which supply information to the method, and that's the next thing we'll look at.

Examine the program below.

```java
public class HelloCustom extends ConsoleProgram {

    public void run( ) {
        displayHello("Sam");
        String name = readLine("Name please?");
        displayHello(name);
        displayHello("and " + "goodbye");
    } //run

    public void displayHello(String msg) {
        println("Hello " + msg);
    } //displayHello
}
```

And here's a sample run.

**Hello Sam**
Consider the relationship between the method call

```
displayHello("Sam");
```

and the method itself. The method call sends "Sam" to the method. The value "Sam" is assigned to the `String msg`. "Sam" is displayed and then execution returns to the `run` method, where a value is assigned to `name`.

Now consider the method call

```
displayHello(name);
```

The value of `name` is sent to the method. That value is assigned to `msg`, is displayed and execution returns to the `run` method.

Finally consider

```
displayHello("and" + "□goodbye");
```

The expression "and" + "□goodbye" is evaluated, resulting in the value "and□goodbye", which is sent to the method and assigned to `msg`. Execution proceeds as described.

### 5.1.3 More to Know About Methods

Methods have many complexities and quirks and there's a lot to know. We'll get into more gory details later but here's few things that are really important.

**Scope, Methods and Local Identifiers**

Recall our discussion of blocks and scope in Chapter 3 Making Decisions—an identifier declared in a block is visible only in that block. A method is a block and so an identifier declared within a method is visible only in that method and is said to be a `local identifier`.

**Arguments and Parameters**

Consider the `displayHello( )` method above. We call it with statements like

```
displayHello("Sam");  or  displayHello(name);
```

In both cases we are sending information to the method and that information is assigned to the variable `msg`. The information being sent is the `argument` ("Sam" or `name`) and the variable it
is assigned to (msg) is a parameter. However, the distinction between argument and parameter is often lost, so don't be surprised if you see both referred to as arguments.

About Arguments Lists
Methods can have many parameters and they can have different data types. We might have

```java
pleaseDoSomething(10, 20.37, "Great");

public void pleaseDoSomething(int x, double r, String s)
{
    ...do something with x, r and s...
} //pleaseDoSomething
```

The arguments in the method call are assigned to the parameters in the method definition in order from left to right. The number of arguments must match the number of parameters and the data or object types usually should, but sometimes don't have to, match exactly.

Methods Calling Methods
It is perfectly legitimate for a method to call another method.

```java
void one(...arguments...)
{
    ...statements...
    two(...arguments...);
    ...statements...
    three(...arguments...);
    ...statements...
}

void two(...arguments...)
{
    ...statements...
    four(...arguments...);
    ...statements...
}
```

In fact a method may even call itself, as in

```java
void fix(...arguments...)
{
    ...statements...
    fix(...arguments...);
    ...statements...
}
```
Recursion occurs when a method calls itself either directly (fix calls fix) or indirectly (fix calls fox which calls fax which calls fix). Recursion is a very useful technique but it leads to programmer insanity very quickly so we’ll avoid it like the plague.

The Order of Methods
Java is very flexible about the top-to-bottom order of methods in a program. A common and simple convention, and one that we will more or less follow, is to write the main method (our public void run( ) method) at the top of program and then define other methods in the order they are first used. Here’s an example.

Boom

```java
//Boom.java
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class Boom extends GraphicsProgram {
    final int WAIT = 1000;

    public void run()
    {
        displayTick();
        displayBoom();
    } //run

    public void displayTick()
    {
        GLabel l1 = new GLabel("...tick...");
        add(l1, 200, 250);
        pause(WAIT);
        l1.move(100, 0);
        pause(WAIT);
        l1.move(100, 0);
        pause(WAIT);
        remove(l1);
    } //displayTick

    public void displayBoom()
    {
        GLabel l2 = new GLabel("B O O M!");
        l2.setFont(\"Serif-BOLD-100\");
        l2.setColor(Color.BLACK);
        add(l2, 120, 250);
    } //displayBoom
} //Boom
```
Since Java doesn't care about the order of methods in the source code we could have put the method definitions in any order in the source code—`run()`, `displayBoom()`, `displayTick()` or `displayBoom()`, `run()`, `displayTick()` and so on.

5.1.4 Why Use Methods?

Methods offer several improvements to the programmer:

- **program modularization**—a difficult program can be cut into smaller methods, allowing the programmer to focus on one task, developing it in isolation from the rest of the code.
- **code reuse**—anytime the lines of code within a method are needed they may be executed simply by calling the method. Hundreds of lines of code can be invoked with a one line method call.
- **code simplification**—a section of code can be simplified by deferring the complexity, pushing a complex task into another method.
- **code readability**—by transforming a complex set of statements into a method call that part of the code becomes more readable.
- **less source code**—a method used several times is defined once in the source code. Without a method the appropriate code would be repeated each time it was needed within the program.
- **debugging ease**—good use of methods isolates tasks from one another. When a bug is discovered, this isolation makes it much easier to identify the specific code causing the problem and thus to debug the code.
- **maintenance ease**—methods that need an update need to be edited in only one place.
- **software libraries**—multiple methods, collected together, can form a software library that can be used in many programs. The ACM Java library is just such a library.

There is one downside to method use—potentially slower execution time. When a method is called the current state of the executing program must be stored, with values temporarily held in memory. Execution then jumps to the method code. After the method code executes, the previously executing code must be restored. This process takes a very, very small amount of time, and in the vast majority of applications this very small time penalty isn’t important. But for some applications, in particular real-time control of physical objects (flight controls on a modern aircraft for example), the time penalty might be important. Since we won't be writing code for any 777s, we won't worry about slowing down our code.

5.1.5 How Arguments Work

Understanding how arguments are passed to methods is key, so let's take a look at that now.

**Rule for Argument Passing**—when a method call contains as argument, the content of the argument is *copied* and this copy is assigned to the parameter in the method definition. This is true whether the argument is a primitive data type or an object.
Passing Primitive Data Types as Arguments

In Java a primitive data type variable or constant may be accurately viewed as a memory location that holds the value of that variable. Thus the statement

\[
\text{int } x = -57;
\]

gives us

\[
\begin{array}{c}
\text{x} \\
-57
\end{array}
\]

The Rule for Argument Passing tell us that a copy of the content of the argument in the method call is assigned to the parameter in the method definition. This means that changes to parameters inside the method do not change the argument outside the method because they are changes to a copy, not to the original. Thus the code segment below does not work as intended, even though the values inside the \texttt{swap()} method are swapped, because \textit{x} and \textit{y} in the method contain copies of the original values in \textit{x} and \textit{y}.

```java
public void run( )
{
    int \textit{x} = 10, \textit{y} = 20;
    \texttt{swap}(\textit{x}, \textit{y});
    \texttt{println}("X: "+ \textit{x} + " Y: " + \textit{y});
}

public void \texttt{swap} (int \textit{x}, int \textit{y})
{
    \textit{int} \texttt{temp} = \textit{x};
    \textit{x} = \textit{y};
    \textit{y} = \texttt{temp};
}
```

The output would be

\[
\begin{array}{c}
\text{x: 10 Y: 20}
\end{array}
\]

We may take a more detailed look at this if we consider memory locations. For the sake of discussion let us assume the variables in the above code segment are at the memory locations 10,000, 11,000 and so on as shown below.

Note that \textit{x} and \textit{y} in the \texttt{run()} method are different variables, in different memory locations, than \textit{x} and \textit{y} in \texttt{swap()}, even though they have the same name. At the beginning of execution we have
The **swap** ( ) method call creates the memory locations for that method and then copies the contents of \( x \) and \( y \) in \texttt{run} ( ) to \( x \) and \( y \) inside the **swap** ( ) method. We have:

<table>
<thead>
<tr>
<th>run method</th>
<th>swap method</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( y )</td>
<td>( y )</td>
</tr>
<tr>
<td>10,000</td>
<td>12,000</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11,000</td>
<td>13,000</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>14,000</td>
</tr>
</tbody>
</table>

Execution of the statements inside the method gives us:

<table>
<thead>
<tr>
<th>run method</th>
<th>swap method</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( y )</td>
<td>( y )</td>
</tr>
<tr>
<td>10,000</td>
<td>12,000</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11,000</td>
<td>13,000</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>12,000</td>
<td>14,000</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>13,000</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>14,000</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

and the method ends. We've successfully switched \( x \) and \( y \) inside the **swap** ( ) method. But what's happened to \( x \) and \( y \) in \texttt{run} ( )? Nothing. They are completely separate variables in different memory locations. They do not change.

Passing the actual value of an arguments is known as **pass by value**. In Java all primitive data types are passed this way.

**Passing Objects as Arguments**

To understand how objects are passed to methods, we need to look at the memory allocation that occurs when an object is created. The statement

\[
\text{GRect } r = \text{ new GRect( );}
\]

finds memory to hold a GRect and puts the address of this new object in \( r \). We might have:

```
memory location 20,000
```

The important thing to note is that the variable \( r \) doesn't contain the GRect itself, it contains the address of the GRect.

A common way to visually represent the relationship between an object and its location in memory is:

\[
\text{contents of GRect object}
\]
and we say "r points to the object" which is a short way of saying "r contains the memory address of the object."

Consider the method call and method definition shown below.

```java
public void run( )
{
    GRect r = new GRect( );
    setup(r, Color.RED);
}

public void setup(GRect r, Color cv)
{
    r.setColor(cv);
}
```

We may take a more detailed look at this if we consider memory locations. For the sake of discussion let us assume the variables in the above code segment are at the memory locations 10,000 and 11,000 and the `GRect` is at memory location 20,000. We won't consider the memory location for `cv`.

Note that `r` in the `run( )` method is a different variable, in a different memory location, than `r` in `setup( )`, even though they have the same name. At the beginning of execution we have

<table>
<thead>
<tr>
<th>run method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code></td>
</tr>
<tr>
<td>10,000</td>
</tr>
<tr>
<td>20,000</td>
</tr>
<tr>
<td>20,000</td>
</tr>
<tr>
<td>default GRect</td>
</tr>
</tbody>
</table>

The `setup( )` method call creates the memory locations for that method and then copies the contents of `r` in `run( )` to `r` inside the `setup( )` method. We have

<table>
<thead>
<tr>
<th>run method</th>
<th>setup method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code></td>
<td><code>r</code></td>
</tr>
<tr>
<td>10,000</td>
<td>11,000</td>
</tr>
<tr>
<td>20,000</td>
<td>20,000</td>
</tr>
<tr>
<td>default GRect</td>
<td>20,000</td>
</tr>
</tbody>
</table>

`r` in `run` and `r` in `setup` refer to the same `GRect` at memory location 20,000.

Execution of the statement inside the method gives us

<table>
<thead>
<tr>
<th>run method</th>
<th>setup method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code></td>
<td><code>r</code></td>
</tr>
<tr>
<td>10,000</td>
<td>11,000</td>
</tr>
<tr>
<td>20,000</td>
<td>20,000</td>
</tr>
<tr>
<td>red GRect</td>
<td>20,000</td>
</tr>
</tbody>
</table>
and the method ends. We’ve successfully colored the GRect. Below is an example program that puts this technique to use.

TwoGRects

```java
//TwoGRects
import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class TwoGRects extends GraphicsProgram
{
    public void run()
    {
        GRect r1 = new GRect(10, 10);
        GRect r2 = new GRect(20, 20);
        add(r1);
        add(r2, 100, 100);
        setup(r1, Color.RED);
        setup(r2, Color.BLUE);

        while(true)
        {
            pause(20);
            r1.move(1, 1);
            r2.move(0, 1);
        } //move the rectangles
    } //run

    public void setup(GRect u, Color cv)
    {
        u.setColor(cv);
    } //setup
}
```

Passing the memory location of an object is known as pass by reference. In this context reference means ‘memory location.’ In Java all objects are passed this way.

**Summary**

Primitive data types are passed by value and the arguments are not changed by changes to the parameters. Objects are passed by reference and the arguments are changed by changes to the parameters. The Rule for Argument Passing applies to both primitive data types and objects.

### 5.2 Introduction to Graphic Objects

Consider the problem of moving a simple UFO on the screen.
This UFO consists of three graphical elements

- red body of the UFO
- black ring showing the outside of the UFO bubble
- green alien head

To fly this UFO across the screen we have to move all three graphical elements individually, which is clumsy with three elements and ridiculous if we create a really complex UFO. There's got to be a better way. And there is.

Object-oriented languages such as Java allow us to create an object, a code representation of an entity, its characteristics and the methods that we can apply to it. We need to create a single object to represent the UFO. Then to fly the UFO we need to move only this single object, not three separate elements. Creating our own objects provides another tool for dealing with complexity.

We've already been working with objects. The statements

```java
GOval r1 = new GOval(40, 50);
r1.setFill(true);
r1.setColor(Color.RED);
```

declare `r1` to be a `GOval` object and manipulate some of its characteristics with the `setFill` and `setColor` methods. So objects are nothing new.

Now we'll learn how to define our own graphic objects, that is objects that have significant visual components.

### 5.2.1 A Simple UFO Object

To create the UFO above we need to create the body, the bubble and alien. But that's easy since we already know how to create the `GOval` objects we need. The only difficulty lies in positioning these elements so that they go together to give the correct appearance to the UFO.

Each graphic object we create has its own coordinate system, laid out the same as the coordinate system for the application window. We need to add the body, bubble and alien in the proper locations on this coordinate system to create the UFO.

A little experimenting and perhaps some work on graph paper might lead us to
- body dimensions 50x25
- bubble dimensions 25x25
- alien dimensions 10x10

and the positioning as shown below.

Finally, if we consider the final look of the UFO we realize the body has to lay on top of the alien, and the alien on top of the bubble and so we will add these GOvals to the UFO in the order bubble, alien and body.

The file below defines a new UFO object to match our design. Note that this file isn't a runnable program—there isn't even a run( ) method. It just defines what a UFO is, and we'll use that definition in a runnable program shortly.

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

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

   public UFO()
   {
      body = new GOval(50, 25);
   }
```

©Daniel L. Schuster, 2009, all rights reserved
Now let's look at it in detail.

import acm.graphics.*;
import java.awt.*;

class UFO extends GCompound {

    private GOval body, bubble, alien;

    public UFO() {
        body = new GOval(50, 25);
        body.setFilled(true);
        body.setColor(Color.RED);
        bubble = new GOval(25, 25);
        add(bubble, 13, 0);
        add(alien, 20, 5);
        add(body, 0, 13);
    }

} //UFO
alien = new GOval(10, 10);
alien.setFilled(true);
alien.setColor(Color.GREEN);

add(bubble, 13, 0);
add(alien, 20, 5);
add(body, 0, 13);
}
} //UFO

UFO.java is a separate file. You must compile it, and you will get errors if the code isn't right. But you can't run it. To do that you create another file that uses the UFO object you just created. A program or class that uses another object is said to be a client of that object and so RunUFO below is a client of the UFO object.

RunUFO creates two UFOs and moves them in different directions.

//RunUFO.java
import acm.program.*;
import acm.graphics.*;

public class RunUFO extends GraphicsProgram {
    public void run() {
        final int WAIT = 20;

        UFO u1 = new UFO();
        add(u1, 0, 0);

        UFO u2 = new UFO();
        add(u2, 700, 0);

        while(true) {
            pause(WAIT);
            u1.move(1, 1);
            u1.move(-1, 1);
        }
    }
} //run
//RunUFO
Note that \textit{UFOs} are treated just like the other objects we've created. We create one with \texttt{new UFO( )}, add it with \texttt{add( )}, and move it with \texttt{move( )}. And a \textit{UFO} is a \texttt{GCompound} (it extends \texttt{GCompound}), which is a kind of \texttt{GObject}, so it gets all of the \texttt{GObject} methods—\texttt{move( )}, \texttt{sendToFront( )}, \texttt{setVisible( )}, \texttt{getX( )}, etc. By extending the \texttt{Gobject} class you get to use all the work that's been done before.

\subsection*{About folders and paths}
When \textit{RunUFO} is compiled and executed Java also needs to be able to find the \textit{UFO} file. Put the \textit{UFO} file in the same folder as \textit{RunUFO}. Java will automatically look in this folder for the information it needs for a complete program.

In the real world more complex programs are written by teams of programmers and may contain many objects. The objects may be stored in many different folders on many different networked computers. To glue this all together at compile and execution time you have to create \texttt{class paths}, which tell Java where to look for the components it needs. This is a pain that you don’t want to deal with right now, so keep it simple and put everything in one folder.

\subsubsection*{5.2.2 More About Constructors}
The \texttt{UFO( )} constructor above creates all \textit{UFOs} equally. Every \textit{UFO} is exactly the same and so this is called the \texttt{default constructor}.

If we need to be able to create individualized \textit{UFOs} then we need another constructor, and the \textit{UFO2} file below illustrates this.

\begin{verbatim}
//UFO2.java
import acm.graphics.*;
import java.awt.*;

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

    public UFO2( ) {
        ...same code as the UFO constructor above...
    } //UFO

    public UFO2(Color bodyColor, Color alienColor) {
        body = new GOval(50, 25);
        body.setFilled(true);
        body.setColor(bodyColor);
    }
}
\end{verbatim}
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);

The ability to have multiple constructors with the same name but different arguments, as shown above, is called constructor overloading.

The program below demonstrates the use of this second constructor.

RunUFO2

import acm.program.*;
import acm.graphics.*;
import java.awt.*;

public class RunUFO2 extends GraphicsProgram
{
    public void run( )
    {
        final int WAIT = 20;
        UFO2 u1 = new UFO2();
        UFO2 u2 = new UFO2(Color.BLUE, Color.ORANGE);
        add(u1, 0, 0);
        add(u2, 700, 0);
        displayStart( );
        waitForClick( );

        while(true)
        {
            pause(WAIT);
            u1.move(1, 1);
            u2.move(-1, 1);
        } //fly the ufos
    } //run

    public void displayStart( )
    {
        GLabel clickAndGo = new GLabel("Click to start");
An object can have many constructors. They all must have the same name and the argument lists must be unique.

5.3 About Data Objects
Graphic objects have a visual representation such as the body of the UFO, its bubble and the alien inside. Data objects contain only information and don't have a significant visual representation. For example, a StudentRecord object might contain biographical information, gpa info, etc. but almost certainly wouldn't have wings or a flame exhaust or any of the other stuff we use in graphic objects.

Data objects introduce their own particular techniques and requirements and we will consider them in a later chapter when we take a much more in depth look at objects.

5.4 Basic UML for Simple Objects
Unified Modeling Language (UML) is a system used to document the characteristics of a object in a quick, organized and partially visual manner. UML is used to provide a birds-eye view of the main characteristics of the component in development. It comes in several varieties and strengths, and we'll start with the easy stuff.

A simple UML diagram consists of a three cell table laid out as shown here.

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

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

<table>
<thead>
<tr>
<th>UFO2</th>
</tr>
</thead>
<tbody>
<tr>
<td>body : GOVal</td>
</tr>
<tr>
<td>bubble : GOval</td>
</tr>
<tr>
<td>alien : GOval</td>
</tr>
<tr>
<td>&lt;&lt;constructor&gt;&gt; UFO2( )</td>
</tr>
<tr>
<td>&lt;&lt;constructor&gt;&gt; UFO2(Color, Color)</td>
</tr>
</tbody>
</table>

Note that the instance variables are described in the following pattern:

variableName : data/object type
as you see with body : GOval. Default constructors are written

```
<<constructor>> nameOfConstructor( )
```

as in

```
<<constructor>> UFO( )
```

and non-default constructors have the data/object types of the arguments in a comma separated list

```
<<constructor>> nameOfConstructor( argument1Data/ObjectType, ...)
```

as in

```
<<constructor>> UFO2(Color, Color)
```

We'll use UML diagrams for many of the objects that we construct.

**Graphics Problem Set**

1) **BounceAround3** Repeat BounceAround2 but this time with two objects to represent the balls that bounce in the window. Write the UML for the ball class.

2) **Dribble3** Using the two ball objects from BounceAround3, bounce those balls from left to right across the bottom of the screen, one bouncing higher than the other, as if on a basketball court. The height of the bounces reduce as the balls move to the right.

3) **MoveObject** - create an object that represents something that moves - a UFO, a tank, a car, a bird, anything that is *real* with a least four graphical elements in different colors. In another program make the object move across the screen. *Note:* I expect some effort in making the object. It should look 'cartoon' real and not just be a set of blobs stuck together!

4) **UFOStory** – create objects to represent trees, a cow and a UFO, then use them to create a pastoral background. Using methods for the intro, middle and end, tell a story of a mysterious alien abduction of the cow. The UFO flies over the pasture (with trees) and beams the cow up to the ship, then it flies off. Write the UML for the tree, cow and UFO classes.