PHAS3459: Scientific Programming Using Object Oriented Languages

Module 2: Objects and Classes

Dr. B. Waugh and Dr. S. Jolly
Department of Physics & Astronomy, UCL

Contents

1. Aims of Module 2

- Be able to define classes and create and use objects.
- Understand and apply the concept of encapsulation.
- Know when and how to use `static` variables and methods.
- Know when and how to use `final` variables.
- Know about and be able to use the `String` class and the numeric data-type wrapper classes.
- Know how to find out about other classes in the Java API.

2. Objects

2.1 Why Are Objects Used?

As stated in Module 1, a *class* is a data type that can represent more complicated entities than simply single real or integer numbers, characters or boolean variables.

In the exercises for Module 1 you wrote functions to carry out some common operations on 3-vectors, such as calculating the magnitude of a vector and the angle between two vectors. Each time you wanted a function to use a single 3-vector as input you had to pass it three arguments, representing the cartesian coordinates of the vector. So a function acting on two vectors had to take six arguments. Now suppose you wanted to deal with 4-vectors, or even higher-dimensional vectors. You can see how the number of variables to be dealt with could easily become cumbersome.

Even more awkward can be getting the result of a calculation back from the function that performs it. If the result is a single number such as a magnitude, a scalar product or an angle there is no problem, but if we want to add two vectors together or calculate a vector product then the result is itself a vector needing three numbers to define it. However, a function can only return a single value.

By defining a new type of *object* to represent a 3-vector, we can simplify our code. We create a *class* that defines the *data* needed to store a 3-vector (three real numbers representing cartesian or polar coordinates) and the *methods* (or *functions*) that can be used to manipulate 3-vectors. When we pass a vector to a function we only need one variable (of type *3-vector*) to do so. If we want to add two 3-vectors together we need only pass two arguments, and we can get the answer back as a single object.

To perform calculations using 3-vectors we will be able to write code like this to calculate $\underline{v} = \underline{u} + \underline{a}t$:

```
ThreeVector u = new ThreeVector(1,0,0);
ThreeVector a = new ThreeVector(0,0,-9.8);
double t = 0.5;
ThreeVector v = u.add(a.multiply(t));
```

The `new` keyword is used to create an object of the given class, also referred to as an *instance* of that class.

2.2 Examples From Module 1

In the examples and exercises in Module 1 you have already used objects and classes, although we glossed over the details so that we could concentrate on the basic syntax of Java.

Even in the simplest Java program in Module 1 you had to define a class, which we called `MainProgram`. This would not be necessary in some other programming languages, but in Java every bit of code belongs to one class or another.

When we added some lines to print out the date, we created a `Date` object:

```
Date myDate = new Date();
```

before printing it to the screen.

When we created the `quadratic` method in the `Simple` class, we had to create an object of the class before calling the method belonging to that object:

```
Simple myS = new Simple();
double xx = 2.0;
double yy = myS.quadratic(xx);
```

2.3 Objects vs Built-in Types

It may be useful to compare briefly the ways in which objects and built-in types are used, before we go on to define a simple class.

```
// creates a variable x to hold
// a built-in data type, int
int x;

// create an object of
// user-defined type MyClass,
// referred to by a variable xc
MyClass xc = new MyClass();

// assign a value to the
// variable x
x = 10;

// use a method of the class
// MyClass
xc.setValue(10);
```

3. Defining Classes

3.1 A Simple Class

We'll start with a very simple class, which simply holds an integer number and allows its value to be set and retrieved. We don't really need to create a new class to do that since the `int` data type already does this and more, but the example illustrates some of the ideas that will be needed in creating classes that provide more useful behaviour.

```
public class SimpleCounter {
  // member variable
  int counter;

  // constructor (ignore for now)
  public SimpleCounter() {}

  // method to set value
  void setCounter(int val) {
    counter = val;
  }

  // retrieve value
  int getCounter() {
    return counter;
  }
}
```

The variable `counter` is a *member variable* or *data member* of the class, as opposed to the *local variables* that were used in Module 1. A local variable is defined within a function and can only be used within that function. To pass its value to another function it must be passed as an argument. In contrast, a member variable is defined within a class and can be used by and shared between the functions defined in that class. In fact there is a separate, independent copy of the variable for each instance of the class (unless it is a *static* variable: see the Static Variables section later in this module). In this case the functions `setCounter` and `getCounter` both have access to the same `counter` variable.

We will use the word *method* to refer to a function that is part of a class, also known as a *member function* of the class. In fact this applies to all functions in Java, although not in some other programming languages.

There is a convention that names of classes start with a capital letter while variables and method names start with a lower-case one. The rest of the name then uses “camel case”, with each subsequent word starting with a capital letter, as in `SimpleCounter` and `setCounter` above.

Now we have a class, which can be used as a blueprint for creating objects. To try it out we need a class with a `main` method, which we can call `TestSimpleCounter`:

```
public class TestSimpleCounter {
  public static void main(String[] args) {
    SimpleCounter count1 = new SimpleCounter();
    count1.setCounter(10);
    System.out.println("value of counter is "+count1.getCounter());
  }
}
```

When we run this program we are creating an object of type `SimpleCounter` and referring to it using a newly defined variable `count1`. We send a message to the object by calling one of its methods using the syntax `object_variable_name.method_name(arguments)`. The object responds by, in this case, changing the value of its internal member variable `counter`.

We can create as many independent instances of this data-type as we want:

```
public class TestSimpleCounter {
  public static void main(String[] args) {
    SimpleCounter count1 = new SimpleCounter();
    SimpleCounter count2 = new SimpleCounter();
    count1.setCounter(10);
    count2.setCounter(15);
    System.out.println("value of counter 1 is "+count1.getCounter());
    System.out.println("value of counter 2 is "+count2.getCounter());
  }
```

There is another way that we could change the value of the `counter` variable inside a `SimpleCounter` object. We could simply access the variable directly from our code in the `TestSimpleCounter` class using the syntax `object_variable_name.member_variable_name`:

```
count1.counter = 10;
```

In the section on Encapsulation we will explain why this is not usually a good idea, although there may be cases where you do want to access a member variable of an object directly. Most commonly you might do this within the class in question. If we wanted to add two `SimpleCounter` objects together we might add a method like the following to the class:

```
int add(SimpleCounter x) {
  return x.counter + this.counter;
}
```

Here `x.counter` means the variable `counter` within the `SimpleCounter` object referred to by the variable `x`.

The keyword `this` has a special meaning: it represents the current instance of the class in which it appears. It provides a way of referring to member variables and methods of a given class from other methods within that class. So in the `add` method given above, it is necessary to refer to the `counter` variable of both the input `SimpleCounter` object `x` *and* the `counter` variable that is a member of the object whose `add` method is being called[1].

For example, the following code calls the `add` method of the `SimpleCounter` object `count1` to add together the counter variables from the objects `count1` and `count2`:

```
SimpleCounter count1 = new SimpleCounter(15);
SimpleCounter count2 = new SimpleCounter(30);
int sum = count1.add(count2);
```

Since the `add` method is being called by the `count1` object, the `count1.counter` member variable within the `add` method is referred to as `this.counter` and has the value `this.counter` = 15, whereas the input `SimpleCounter` object `count2` provides the input variable `x.counter` = 30.

3.2 Constructors

In our simple example there is a line labelled “constructor”. A constructor is like a method but it is not called in the same way. It is used by the computer to set up an object when we tell Java to create a new object using the `new` command.

There are two characteristics that tell us this is a constructor and not a normal method:

- it has the same name as the class itself;
- it has no return type, such as `int` or `void`.

If all that is needed is to declare some variables and we are happy with the default values (zero for numbers, false for boolean variables) then the constructor doesn't need to do anything, as is the case above. When we create a `SimpleCounter` its `counter` variable is simply set to 0. In this case we could in fact have omitted this line and the Java compiler would have filled it in for us.

However, if we want to create counters with other starting values then we need to provide a constructor that takes the desired value as an argument:

```
public SimpleCounter() {}
public SimpleCounter(int value) { counter = value; }
```

In other classes we may need to perform calculations in the constructor rather than simply changing the values of variables.

Note that we have also kept the original *default* constructor (which takes no arguments) to provide an easy way of creating a counter with the value zero:

```
// set to 0
SimpleCounter count1 = new SimpleCounter();

// set to 3
SimpleCounter count2 = new SimpleCounter(3);
```

If we do not define any constructors then Java will automatically add a default constructor that takes no arguments and does nothing, like the first one in the example above. However, if we write any of our own constructors then Java will no longer do this and we have to add our own default constructor if we still want it.

Often we need to create a new object of a class from within a method of the same class. If we wanted to add together two `SimpleCounter` objects, as in the section A Simple Class, but return the result as a `SimpleCounter` object instead of an integer, then we would use something like the following:

```
SimpleCounter add(SimpleCounter x) {
  int sum = x.counter + this.counter;
  return new SimpleCounter(sum);
}
```

3.3 Method Overloading

The code above in which we wrote two constructors is an example of *overloading*, in which we define several methods with the same name but taking different lists of arguments. These methods are completely independent and can in principle perform completely different functions. However, it is more common for them to do more-or-less the same thing, from the point of view of someone using the method, but to do it in a different way depending on the input.

You have come across another example:

```
System.out.println("Program starting");

Date myDate = new Date();
System.out.println(myDate);

double x=1.0;
System.out.println(x);

int ix = 1;
System.out.println(ix);
```

There are many different versions of the `println` method, each specialising in printing a particular type of variable. The resulting code is much more readable than it would be if we needed many methods with different names, e.g. `printlnInt`, `printlnDouble` ...

By different lists of arguments we mean that the number of arguments is different, the types are different or the order is different. The following functions can all co-exist with the same name:

```
public void aMethod() {}
public void aMethod(int ia, float fb) {}
public void aMethod(float fb, int ia) {}
public void aMethod(float fa, float fb) {}
public void aMethod(int ia) {}
```

Different methods with the same name can even return different data types:

```
public int aMethod(int iVal) {
  return 2*iVal;
}

public double aMethod(double dVal) {
  return 2.0*dVal;
}
```

However, you cannot have more than one method with the same name and the same argument list, so two methods that take the same input and return different types must have different names, e.g.

```
public int aMethodInt(int iVal) {
  return 2*iVal;
}

public double aMethodDouble(int iVal) {
  return 2.0*iVal;
}
```

3.4 Encapsulation and Access Control

In the section A Simple Class we showed how you could directly access and change the value of the `counter` variable within a `SimpleCounter` object without using the `setCounter` method.

While this might seem a much simpler way of doing things, it is in fact usually a bad idea to access the member variables of an object from a method in another class. *Encapsulation* or *data hiding*, i.e. preventing the data inside an object from being modified except through carefully written methods, is one of the most important principles of object-oriented design. Particularly when designing large and complex applications, breaking encapsulation can make software much harder to debug and to modify.

For example, a method such as `setCounter` can carry out additional checks such as making sure the value is within a certain allowed range (maybe only positive numbers are allowed) or counting the number of times the counter has been modified. Directly changing the variable `counter` would bypass these checks.

In more complicated classes, we might even want to change the way the data is stored without forcing users of the class to change their own code. We might decide to store vectors using polar instead of cartesian coordinates and rewrite the methods (`getX`, `getTheta` etc.) appropriately. Other code could still use the same method calls (e.g. `double x3 = position.getX();`) without knowing that the x is in fact now being calculated from the polar coordinates.

A way to make sure that no-one can change the value of a variable directly is to declare it `private`. Instead of:

```
int counter;
```

we write:

```
private int counter;
```

A `private` variable can only be accessed (read or changed) from within the class in which it is defined. A `public` variable can be accessed from any class. If you don't specify either, the variable is *package-private*: it can be accessed from any class in the same *package* as the class in which it is defined. (Packages will be discussed in more detail later in the course.) A variable can also be declared `protected`, and we will explain what this means when we discuss inheritance later in the course.

The same access control specifiers (`public`, `private` and `protected`) can be applied to methods as well as to member variables. While some methods are part of the advertised behaviour of a class and should be `public`, others are part of the internal workings of the class and should not be available from outside. This simplifies the use of the class from other code, and enables the developer of a class to make changes to such internal workings without causing errors in the "client code".

In our `SimpleCounter` example, the `setCounter` and `getCounter` methods are clearly a vital part of the functionality of the class, and should be made public so that they can be used from outside the package in which it is defined:

```
public void setCounter(int val) {
  counter = val;
}

public int getCounter() {
  return counter;
}
```

While on some occasions the default package-private behaviour may be what is required, it is important to consider in each case what the appropriate access level is.

3.5 Static Variables and Methods

3.5.1 Static Variables

Sometimes it is useful to have a variable that is shared between all objects of a given class, rather than each instance (object) having its own independent copy. We might want to make sure the `counter` variable in our `SimpleCounter` objects does not exceed some limit, while keeping the ability to change this limit later.

This can be done using a *static* variable, also called a *class* (as opposed to *object*) variable because it belongs to the class as a whole rather than to a particular instance.

You have already used one static variable: `System.out` is a static member variable of the `System` class. It is an object of type `PrintStream` and represents the "standard" output stream, which normally directs output to the screen. You have used the `println` method of this class to print output a line at a time. The `Math` class, which you have used for its

mathematical functions, also has two static variables, representing the values e and π. They are referred to as `Math.E` and `Math.PI`.

Note the syntax: `class_name.variable_name`.

We could modify our counter class as follows:

```
public class SimpleCounter {

  // same val for all SimpleCounters
  private static int max = 1000;

  private int counter;

  public SimpleCounter() {}

  public int getCounter() {
    return counter;
  }

  public void setCounter(int val) {
    // only change if new value <= max
    if (val <= max) {
      counter = val;
    }
  }
}
```

A simple example illustrates that there is only one copy of a static variable, shared between all instances of a class:

```
public class StaticTest {
  private int varA;
  private static int varB;

  public static void main(String[] args) {
    StaticTest objectX = new StaticTest();
    StaticTest objectY = new StaticTest();
    objectX.varA = 1;  objectX.varB = 2;
    objectY.varA = 10; objectY.varB = 20;
    System.out.println("objectX: varA="+objectX.varA+
                       " varB="+objectX.varB;
    System.out.println("objectY: varA="+objectY.varA+
                       " varB="+objectY.varB;
  }
}
```

Here we are using the same syntax, `object_variable_name.member_variable_name`, to access both static and non-static member variables. It is possible to refer to static variables this way, but it can lead to confusion and so `class_name.variable_name` is preferred in the static case.

Note also that we are changing the value of a private variable by accessing it directly. This is allowed because the code that changes it is in the same class as the private variable.

3.5.2 Static Methods

A method that doesn't depend on any non-static variables, i.e. one that doesn't depend on a particular object, can itself be declared `static`. It can be used without even bothering to create an object at all. One example you have already used several times is the `main` method of a class, the method that kicks off everything that happens when you run a Java program. It is run before any objects have even been created, so it can't belong to a particular object but must belong to the class itself.

The mathematical functions you have used from the `Math` class are also static methods. The value of sin θ depends only on θ, not on any information held by a specific object. Similarly `currentTimeMillis()` is a static method of the `System` class.

You can call a static method of a class using the syntax: `class_name.method_name(arguments)`. Compare this to the syntax shown earlier for calling a non-static method.

```
long tStart = System.currentTimeMillis();
double cos_90 = Math.cos(Math.toRadians(90));
```

In our `SimpleCounter` class we could have a method to print the maximum allowed value:

```
public static void printMaximum() {
  System.out.println("maximum = "+max);
}
```

Note that as a static method is not associated with a particular instance of a class, it cannot access any non-static variables or methods, as it doesn't have access to any object containing them. So the following will not compile:

```
public static void printCounter() {
  System.out.println("counter = "+counter);
}
```

Often there are two ways of writing a method to perform a given function, one using a non-static method — a member function of one of the objects involved — and another using a static method that takes all the required inputs as arguments. In the section on Constructors we wrote a non-static function to add two `SimpleCounter` objects. We could write a static version as follows:

```
public static SimpleCounter add(SimpleCounter x, SimpleCounter y) {
  int sum = x.counter + y.counter;
  return new SimpleCounter(sum);
}
```

Then we have a choice of two ways of carrying out the same addition:

```
// non-static
SimpleCounter sumA = sc1.add(sc2)

// static
SimpleCounter sumB = SimpleCounter.add(sc1,sc2)
```

To avoid duplicating code, it is customary to do the calculations in one version, say the static one, and use the `this` keyword to create a simple non-static version that just "wraps" the static function:

```
public SimpleCounter add(SimpleCounter z) {
  return add(this,z);
}
```

3.6 `final` Variables

If it is important that the value of a variable is fixed and not (accidentally or misguidedly) changed during the running of a program, the variable can be declared `final`. It is unlikely that changing the value of π in the middle of a program would be desirable, so the corresponding variable in the `Math` class is declared `final`:

```
public static final double PI;
```

Note also that this member variable is declared public, so it can be accessed directly as `Math.PI` without using a method to find its value. This is only safe because no-one can change the value of a `final` variable even if they try.

In our counter class, we might want to make sure that the maximum value cannot be changed within one `SimpleCounter` object, thus changing the value applying to all other `SimpleCounter` objects:

```
private static final int max = 1000;
```

3.7 Converting Objects to Strings

It is often useful to have a way of converting an object to a string so that it can be printed to the screen in a comprehensible form. By defining a `toString` method in a class we can give Java a way of doing this conversion, which it will automatically call on if we try to print the object, e.g. in our `SimpleCounter` class:

```
public String toString() {
  return "counter = "+counter+ ", max = "+max;
}
```

Now we can easily see the state of our `SimpleCounter` objects:

```
SimpleCounter c = new SimpleCounter();
System.out.println("state of c: "+c);
```

---

[1] For more information and examples, see the Oracle Java Tutorials: http://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html. ↩

---

4. Useful Classes

4.1 The Java API

The Java *Application Programming Interface (API)* comprises a large set of useful classes which, together with the *Java Virtual Machine (JVM)*, form the *Java Runtime Environment* (JRE) which can be downloaded from the Sun website or other sources. We have already been using a number of classes from the Java API, including `System` and `Math`. The Java API also includes classes to handle graphics, networking, database access, cryptography and more.

For full details of all the available classes, and all the methods and values they provide, you can refer to the Java API documentation at http://docs.oracle.com/javase/8/docs/api/, also linked from the "Java Resources" page on the course website.

This documentation has been generated using the *Javadoc* tool, which processes specially formatted comments in the Java code to produce a reference guide in HTML form. Later in the course you will learn how to use Javadoc to document your own classes.

4.2 The `String` Class

The `String` class is more tightly integrated with the Java language itself than most other classes. Any sequence of characters enclosed within double quotes `"like this"` is interpreted by the Java compiler as a string and is automatically converted into a `String` object. Thus you have already used strings in earlier examples and exercises.

You can use `String` objects more explicitly in order to carry out various operations on text:

```
String greeting = "Hello world!";
int numChars = greeting.length();
String firstWord = greeting.substring(0,5);
```

Another special feature of this class is that the `+` operator can be used to concatenate strings, rather than calling a method of the class. You can even "add" numbers and they will automatically be converted to strings:

```
String greeting = "Hello";
int iUserID = 321;
System.out.println(greeting+" user number "+iUserID);
```

4.3 Data Types as Objects

In Java everything is an object or class except for the built-in data types `byte`, `short`, `int`, `float`, `double`, `boolean` and `char`. In keeping with the object-oriented nature of Java, there are also *object* representations of these built-in types. For example, the `int` type has a corresponding `Integer` class, `double` has `Double` and so on. These are referred to as *wrapper* classes, as the primitive quantity is "wrapped" inside an object.

The primitive types are frequently used, particularly when mathematical calculations are required, because they make the code simpler and often faster. However, the wrapper classes are needed when passing a number to a method that needs an object as input, which is often the case when dealing with *collections* of numbers, which will be dealt with in a later module. These classes also provide functions for such tasks as converting a `String` to a number or vice versa, or converting between data types. They also provide static constants such as `MIN_VALUE` and `MAX_VALUE` giving the lower and upper bounds of the data type.

Converting between primitive types and wrapper classes is handled automatically by the Java compiler in many cases[2]. Converting a primitive into the corresponding wrapper type is known as a *boxing* conversion, and the reverse process is an *unboxing* conversion. ("Boxing" is the same as "wrapping" in this context.)

```
// automatic boxing conversion
// from int to Integer
Integer intA = 1;

// automatic unboxing conversion
int intB = intA;

// type conversion
double d1 = intA.doubleValue();

// convert string to number
// more useful with e.g. a
// string read from a file
int i2 = Integer.parseInt("123");
```

---

[2] This is true for versions of Java from 1.5 onwards; in earlier versions it was necessary to make these conversions explicitly. ↩

---

5. Summary

In Module 1 we introduced the basic ideas behind object-oriented (OO) programming. In this module you have learned how to create your own classes and objects in Java, and to use existing classes from the Java API.