PHAS3459: Scientific Programming Using Object Oriented Languages

Module 5: Handling Objects, Arrays and Collections

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

Contents

1. Aims of Module 5

- Understand how object variables differ from variables of primitive data types.
- Know when an object passed as an argument can be modified and how to prevent this.
- Understand the use of the `null` reference.
- Know how to check whether two objects are equal.
- Know how to create a deep or shallow copy of an object.
- Understand how to copy objects and test for the equality of two objects.
- Be able to store and manipulate collections of objects.
- Understand what happens when Java passes a collection as a parameter in a function call.
- Be familiar with the following:
    - *Arrays*, `ArrayLists` and `HashMaps`;
    - `Iterators` and `for` loops for looping over collections;
    - implementing `clone()` and `equals()` methods.

2. Handling Objects

2.1 Object Variables

2.1.1 References

So far we have been using variables to refer both to “primitive” built-in data types and to objects belonging to various classes. However, there is an important difference between these types of variables. A variable is basically a name referring to a particular piece of memory in the computer. In the case of primitive variables, that piece of memory stores a pattern of bits that represents a particular number or character. A variable can be copied, as in the following example, resulting in two variables with the same value, but they are independent: changing one doesn't affect the other.

```
int a = 1;
int b = a;
b = 2;
System.out.println("a = "+a);
```

In the example above, the names `a` and `b` refer to different memory addresses. Initially they both contain the same value, 1. Changing the number in the memory address labelled `b` has no effect on that labelled `a`, as you can see if you run the code. The variables `a` and `b` refer to different “things” that may happen to have the same value.

This may seem fairly obvious, but the situation with object variables is more complicated. The memory address labelled with the name of a particular variable contains not the object itself but a *reference* to the object. This is effectively the memory address where the object itself can be found.

If an object variable is copied, as in the next example, the result is two variables that contain the same pattern of bits and therefore refer to the same memory address and thus the *same object*.

```
package module5;
import module2.SimpleCounter;

public class ReferenceTest {
  public static void main(String[] args) {
    SimpleCounter c = new SimpleCounter(1);
    SimpleCounter d = c;
    d.setCounter(2);
    System.out.println("c = "+c.getCounter());
  }
}
```

There is only one `SimpleCounter` object created in this example, but two variables (`c` and `d`) that refer to it. So using the reference `d` to change the state of the object also affects what happens when we use the reference `c`.

2.1.2 Garbage Collection

In the following example, although two `SimpleCounter` objects are created, each referred to by a separate variable, one of the references is then changed so that both point to the same object:

```
SimpleCounter c = new SimpleCounter(3);
SimpleCounter d = new SimpleCounter(4);
d = c;
```

The second `SimpleCounter` object has now effectively been "forgotten": there is no longer any way to access it as there is no variable containing a reference to it. When this happens, Java will eventually delete the "lost" object and free up the memory it was occupying, in a process called *garbage collection*. As long as there is still at least one reference to an object, that object will not be deleted.

2.1.3 Passing Objects as Arguments

When a variable is passed as an argument to a function, the function actually receives a *copy* of its value, so any changes to the value are not passed back:

```
public static void main(String[] args) {
  int a = 1;
  changeIt(a);
  System.out.println("a is still "+a);
}

public static void changeIt(int i) {
  i = 2;
}
```

This is still true for object variables, but in this case the *value* of the variable (which cannot be changed) is a *reference* to the object in question. The function cannot change this reference to point to a different object.

However, it can use the reference to change the state of the object itself:

```
public static void main(String[] args) {
  SimpleCounter a = new SimpleCounter(1);
  changeIt(a);
  System.out.println("a is now "+a.getCounter());
}

public static void changeIt(SimpleCounter c) {
  c.setCounter(2);
}
```

In most cases you should try to avoid letting this happen. If you want a function to change the value of an object, the function should either be a member of that object or the changed object should be *returned* by the function. Changing an object passed as an argument often leads to unintended side-effects when the object is used again afterwards, and it can be hard to understand the code.

One way of protecting yourself against this behaviour is to create a copy of the object and pass that to the function, keeping the original safe. Alternatively, the function itself can create a copy to be modified for its own purposes, again leaving the original unchanged. Copying objects will be covered in the section on Copying.

2.1.4 Mutable and Immutable Objects

Another useful technique for protecting your objects from unwanted changes is to make them *immutable* (or *read-only*). This means not providing any methods that can change the state of the object after it has been created. Our `SimpleCounter` objects are not immutable because they have a `setCounter` method that changes the internal `counter` variable. In many cases the whole point of a class is that its instances *can* be changed, so it doesn't make sense to make them immutable.

An example of a class whose instances are immutable is `String`. Once you have created a `String` object, you can pass it to any function without worrying that it may have changed when you next look at it. A disadvantage of this is that when you want to concatenate two strings, a third string has to be created and the contents of the original strings copied into it. The following snippet demonstrates this:

```
String x = "Hello";
System.out.println(System.identityHashCode(x));
x = x + " world";  // creates a new object
System.out.println(System.identityHashCode(x));
```

This is a slow process, especially if it has to be repeated many times, so an alternative, *mutable* version is provided for building longer strings out of shorter ones. This is the `StringBuilder` class. In order to avoid creating many intermediate `String` objects, Strings

can be added onto a `StringBuilder` object until it is complete and the `StringBuilder` object is then converted back into a `String`:

```
StringBuilder z = new StringBuilder("Hello");
System.out.println(System.identityHashCode(z));
z.append(" world");  // adds data to existing object
System.out.println(System.identityHashCode(z));
```

2.1.5 The `null` Reference

In some cases an object variable may not actually point to an object at all. This may be because it has been declared but not initialized, which can also be the case for primitive variables:

```
int a;
SimpleCounter b;

// The following lines fail to
// compile because a and b are
// not initialized

System.out.println(a);
int c = b.getCounter();
```

Unlike primitive variables, object variables may be initalized but given the special value `null` to indicate that they do not actually refer to an object. This is sometimes done to reassure the compiler that an object variable has been initialized even though the corresponding object cannot yet be created, as you saw in Module 3:

```
// set object's value to
// a "dummy" value
MyCounter xx = null;

try {
  xx = new MyCounter(150.0);
  xx.incrementCounter();
}
catch (Exception e){
  System.out.println("ERROR:"+e.getMessage());
}
try {
  // If the constructor failed above,
  // xx will still be null, and the
  // next line will cause an exception ...
  xx.print();
}
catch (NullPointerException e){
  // ... which will be caught here.
  System.out.println("ERROR: xx is not defined; it is still null");
}
```

If you try to use a `null` reference then a `NullPointerException` will be thrown, which will cause your program to crash unless it is caught. Therefore in general you should only

use this technique if:

- you are sure that the first `try` block will in fact succeed in assigning a value to the variable rather than throwing an exception, or;
- you check whether the variable is `null` before trying to use it, or;
- you catch the `NullPointerException` and carry out some appropriate alternative action, as in the example.

Some methods may return a `null` reference if the object they would normally return does not exist or cannot be created for some reason. This is true for some of the methods of the collections classes described later in this module. Again, unless you are absolutely sure because of the way the code is written that the result can never actually be `null`, you should always check the return value or catch the `NullPointerException`.

2.2 Testing Object Equality

To test whether two primitive variables have the same value you can use the `==` operator. This checks whether the numbers at the two memory locations referred to are identical. The same operator can be used to test if two object variables are equal, but in this case it will be checking whether the two *references* are identical, which is the case only if they refer to the *same object*:

```
SimpleCounter a = new SimpleCounter(1);
SimpleCounter b = new SimpleCounter(1);
System.out.println(a==b); // false!
SimpleCounter c = a;
System.out.println(a==c); // true
```

The way to check whether two objects are equal in the sense that they are equivalent in some more meaningful way is to use the boolean `equals` method. By default this will do the same as the `==` operator and return `true` only if the two variables refer to the same object. However, you can override this behaviour by adding your own `equals` method to the class to carry out a more sensible test. Usually this means checking whether the relevant member variables are equal. For example two 3-vectors are equal if their x, y and z components match. In the case of the `SimpleCounter` class used as an example in Module 2, we only need to check whether the `counter` variables of the two objects are equal.

While this is simple in principle, there are a few complications that arise from the fact that the `equals` method takes as its argument an `Object`, i.e. something that can in principle be any type of object, not necessarily an object of the type it is being compared to. The reasons for this will hopefully be a little clearer after the module on inheritance. If the types of the objects are different, the method should normally return false[1]. The `equals` method also has to cope with the possibility that it will be passed a `null` reference rather than a reference to a real object, in which case the method should always return `false`.

Fortunately, as this is such a common task, Eclipse provides a tool to do all this work for you. While editing the class that needs an `equals` method, go to the *Source* menu and select *Generate hashCode() and equals()....* You will be presented with a list of member variables of the class and allowed to select which ones should be used in comparing

objects. Normally this will be all of them, but there may be occasions when one member variable can be different but the objects are still regarded as equal. You can also choose where in the source code to insert the new methods. Then there are two checkboxes. You should not tick "*Use 'instanceof' to compare types*" and for now you can also leave out "*Generate method comments*". You will end up with two additional methods in your class. The `equals` method should look something like this:

```
public boolean equals(Object obj) {

  if (this == obj) {
    return true;
  } else if (obj == null) {
    return false;
  } else if (getClass() != obj.getClass()) {
    return false;
  }

  final SimpleCounter other = (SimpleCounter) obj;
  if (counter != other.counter) {
    return false;
  }
  return true;
}
```

First the method checks to see whether the argument is actually a reference to the object whose `equals` method is being called. This is not strictly necessary but can save time that would otherwise be spent doing the other checks. Next the argument is checked to see if it is `null`, in which case the method must return `false`. Then there is a test to see if the two objects do in fact belong to the same class. (You don't need to worry about how this works.) If not, they are not equal. Lastly, the member variables are compared if the two objects do belong to the same class, but are not the same object.

The other method, `hashCode()`, is used in other ways of comparing objects, for example when objects have to be sorted into a reproducible order (e.g. when storing objects in a `HashMap` as described later in this module). It is important that if two objects are equal, according to the `equals` method, they should also return the same result from `hashCode`. The default version of `hashCode` returns a value based on the memory address of the object, which will of course be different even for separate objects that are equal. So if you provide your own `equals` method for a class you should also provide a `hashCode` method. Since Eclipse can do the work for us we won't go into further detail here.

2.3 Copying Objects

Copying objects is a frequent requirement. We may need to start with several identical objects before modifying them in various ways, or we might want to make a copy of an object before it is modified and then compare the "before" and "after" versions to see if a method has changed it.

Copying primitive variables is simple but, as we saw in the section on References, you can't copy an object simply using a statement like `a = b`. This will only copy the reference, not the object itself.

Suppose we have a `SimpleCounter` object `c1` and want to create an independent copy `c2`. There are several ways we can go about this:

- extract the information held in `c1` and instantiate `c2` with this information;
- use the built-in default `clone` method that is available to all Java objects;
- provide our own `clone` method;
- provide a *copy constructor*.

The first approach is illustrated here, but rapidly becomes messy as the objects involved become more complicated:

```
SimpleCounter c1 = new SimpleCounter(1);
SimpleCounter c2 = new SimpleCounter(c1.getCounter());
```

2.3.1 The `clone` Method

In less simple cases it is easier and more efficient to use the `clone` method, although there are a few things you need to be aware of. First let's try using the built-in default version:

```
try {
  SimpleCounter c2 = (SimpleCounter) c1.clone();
  System.out.println("Cloned object");
}
catch (CloneNotSupportedException e) {
  System.out.println("Can't clone object!");
}
```

The first complication is that this will not work (a `CloneNotSupportedException` will be thrown) unless you modify the `SimpleCounter` class by adding the declaration `implements Cloneable` to the class as follows:

```
public class SimpleCounter implements Cloneable {
  // The rest of the class
  // should still be in here.
}
```

What this means will become clearer in the module on inheritance and interfaces. Another feature, as you see in the example, is that the call to the `clone` method has to be put in a `try` block as it may throw a `CloneNotSupportedException`.

The `clone` method returns an object without knowing what the actual type of the object is, as a reference of type `Object`. That's why in the example we have to *cast* the result to what we know to be the correct type, in this case `SimpleCounter`.

If you try running this code you may also notice that it works if it is included somewhere in the `SimpleCounter` class but not if it is in another class such as `TestSimpleCounter`. This is because the default `clone` method is declared `protected`, yet another feature that will be explained later!

This last problem at least can easily be solved by giving the class its own `clone` method. Once more Eclipse gives an easy way to do this. From the *Source* menu, choose *Override/Implement Methods*.... Select `clone` and make sure the other boxes are not ticked, then click *OK*. The result should be something like this:

```
protected Object clone() throws CloneNotSupportedException {
  return super.clone();
}
```

(There may be a couple of extra lines containing an annotation and a comment, but you can ignore them.) This simple `clone` method simply calls the default version, but makes it accessible to other classes in the same package.

2.3.2 Deep and Shallow Copies

The default `clone` method may in fact be sufficient for copying simple classes, but to know whether it meets your needs you must understand the distinction between *deep* and *shallow* copies. By default `clone` carries out a *bitwise* copy: the result is an exact copy of the original object, including any member variables even if they are object references. This means that both copies contain references to the *same* member objects. This is referred to as a *shallow* copy. However, often what we really want is for any member objects to themselves be duplicated to create two completely independent entities. This is a *deep* copy.

For the sake of a simple example, suppose we have a class `SimpleCounterPair` that contains two `SimpleCounter` objects as members:

```
public class SimpleCounterPair implements Cloneable {
  public SimpleCounter first = new SimpleCounter(1);
  public SimpleCounter second = new SimpleCounter(1);

  protected Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
}
```

Now let's try creating an instance of this class and cloning it:

```
SimpleCounterPair p1 = new SimpleCounterPair();
try {
  SimpleCounterPair p2 = (SimpleCounterPair) p1.clone();
  p2.first.setCounter(2);
  System.out.println(p1.first.getCounter());
}
catch (CloneNotSupportedException e) {
  System.out.println(e);
}
```

Although the variables `p1` and `p2` refer to separate `SimpleCounterPair` objects, each of those objects contains references to the same two `SimpleCounter` objects. We have created

a *shallow* copy of the first `SimpleCounterPair`.

To create a *deep* copy we need to modify our `clone` method to create separate copies of the member objects. This is done by calling their `clone` methods in turn:

```
protected Object clone() throws CloneNotSupportedException {
  SimpleCounter copy1 = (SimpleCounter) this.first.clone();
  SimpleCounter copy2 = (SimpleCounter) this.second.clone();
  SimpleCounterPair copy = new SimpleCounterPair();
  copy.first = copy1;
  copy.second = copy2;
  return (Object) copy;
}
```

This is not a very elegant example but it serves to illustrate the technique. In this simple case it would of course be simpler to create new `SimpleCounter` objects within the `clone` method of `SimpleCounterPair` by using `getCounter` to read the integer numbers within them, then using the `SimpleCounter` constructor.

This technique can be very tricky to implement for less simple classes. We need to ensure that every object contained within the object to be cloned has its own sensible `clone` method. We can also get into trouble if we have a circular dependency, i.e. an object *A* containing a reference to an object *B* that in turn refers to object *A*. Another approach is to use *serialization* to write the contents of an object into an array of memory locations in the computer and then read this data into a new object. You do not need to know about serialization in this course and the following example is included for information only:

```
public class ObjectCloner {
  public static Object deepCopy(Object old) throws IOException {
    ObjectOutputStream oos = null;
    ObjectInputStream ois = null;
    try {
      // Write the object to a byte array
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      oos = new ObjectOutputStream(bos);
      oos.writeObject(old);
      oos.flush();

      // Now read it back into a new object
      ByteArrayInputStream bis =
                  new ByteArrayInputStream(bos.toByteArray());
      ois = new ObjectInputStream(bis);
      return ois.readObject();
    }
    catch (ClassNotFoundException e) {return null;}
    finally {
      if (oos!=null) oos.close();
      if (ois!=null) ois.close();
    }
  }
}
```

2.3.3 Copy Constructors

Finally, another way of copying objects, which may be more suitable than using `clone` in some cases, is to provide a *copy constructor* in your class. This is a constructor that takes as its argument an object (of the same class) to copy into the newly created object. For the `SimpleCounter` class it might look like this:

```
public SimpleCounter(SimpleCounter c) {
  counter = c.counter;
}
```

Now we can create a copy of an existing `SimpleCounter` object like this:

```
SimpleCounter c2 = new SimpleCounter(c1);
```

---

[1] There are exceptions to this, but we will not discuss them here. ↩

---

3. Collections

3.1 Introduction to Collections

The designer of a computer program intended to deal with significant amounts of data will need to carefully consider how to organise the data in a logical and efficient way. It is often the case in scientific programming applications that substantial amounts of data — either real or simulated — need to be managed. For example, a high energy physics application might need to store large numbers of particle trajectories resulting from a collision, or an astronomical analysis program may need to store lists of spectroscopic data resulting from a sky survey, etc.

There are two ways of organising collections of data: using primitive **Arrays**, or using far more sophisticated **Collection Objects**.

3.2 Arrays

An array is the simplest construct for storing a list of values. It is a contiguous block of memory divided into “slots”, where each slot contains the requisite storage space for each individual object. So for example for an array of `double` floating point numbers, each slot is 8-bytes wide. The basic array syntax is illustrated in the following:

```
// General syntax to declare an array is:
type[] arrayName = new type[length]

// An array storing two integer values:
int[]ks = new int[2];
ks[0] = 45; ks[1] = 46;

// An array storing 10 Point objects:
Point[] mypoints = new Point[10];
mypoints[0] = new Point(10,11);
mypoints[0].x = 12;
// ... etc.
```

A few things to remember:

- The array size is defined at the point at declaration and once defined is fixed.
- An array of length n has members from [0] to [$n - 1$]. It's easy to forget this and try to access element [n], which will produce an error.
- The length of an array is stored in its data member "`length`".

These points are illustrated in the second code fragment:

```
// Declare an array to store
// six integers:

int[] values = new int[6];

// The following is fine if you
// know the array's length:

for (int i = 0; i <=5; i++) {
  values[i] = i*i;
}

// But it's better to use the
// array's "length" variable:

for (int k = 0; k < values.length; k++) {
  values[k] = values[k]*values[k];
}

// The following line will
// produce an error:

int ix = values[6];
```

We can use arrays as the return type and in the parameter list for function calls, for example:

```
public class myList {

  private int[] data;

  public myList(int[] values) { data = values ;}

  public int[] negativeList() {
    int[] nData = new int[data.length];
    for (int i = 0; i < data.length; ++i){
      nData[i] = -1*data[i];
    }
  return nData;
}

// In some other code we might then do

int[] data = {0,5,6,7,9,10};
myList myL = new myList(data);
int[] nData = myL.negativeList();

// ... etc.
```

Recall that arrays, as objects, have a different behaviour from primitive types when passed as function arguments, as discussed in the section on Passing Objects as Arguments above. As good programming practice, the method "`negativeList`" returns a new array rather than modifying the array passed to it. Note also the alternative method of array initialisation employed here:

```
// automatically create an integer array
// of length 6 and set the values:
int[] values = {0,1,4,9,16,25};
```

3.2.1 Multi-Dimensional Arrays

We can define multi-dimensional arrays, the elements of which are indexed by two or more indices. In the case of a 2-D array or matrix which, for example, we may use to represent values of a function on a grid of (x, y) coordinates, the syntax is as follows:

```
int[][] grid = new int[4][5];
grid[0][0] = 1;  // first element
// ....
grid[3][4] = 1;  // last element

// or we can initialise as follows:
int[][] grid2D = { {1,2,3,4},       // row-1
                   {11,12,13,14},   // row-2
                   {11,12,13,14} }; // row-3

// Size (#rows, # columns) of a
// 2D array is determined as follows:

int nRows = grid2D.length;
int nCols = Array.getLength(grid2D[0]);

// The Array class must be imported
// in your code using
// "import java.lang.reflect.Array;"
```

This syntax extends in a natural way to higher dimensions. However if you find yourself needing to use N-dimensional arrays where ($N > 3$) then you should consider whether there is an alternative way of achieving your programming goals since manipulation of higher-dimensional arrays is not particularly convenient and is somewhat error prone. For example, rather than storing the values of some function in a 3-D array, it might be better to define your own custom class "spacePoint" containing the pertinent information (e.g. x, y, z coordinates and the value of the function at that point) and store these objects in a suitable Java collection class.

3.3 Java Collection Classes

While arrays are relatively easy to use it's clear what's going on, they have severe limitations:

- once the array is created you cannot extend it, or otherwise modify its size;
- built-in arrays are relatively "dumb" objects with no functionality to sort them, find the maximum values stored in them, etc.

There is clearly therefore a need for more sophisticated higher-level collection classes. The Java collection classes fall into four broad types:

- **set**: a mathematical set of elements (no duplicates allowed). The available classes are `HashSet` and `TreeSet`.
- **list**: an ordered sequence of elements (duplicates allowed). The available classes are `ArrayList`, `Vector` and `LinkedList`.
- **queue**: a collection of elements held prior to processing, providing various related methods.
- **map**: a key-value cross reference. The available classes are `HashMap`, `TreeMap` and `Hashtable`.

The choice of which class to use is dictated by the job to be done. There are certain trade-offs to be made in terms of simplicity versus efficiency (either in terms of storage or

speed of execution). We will concentrate on the `ArrayList` (list) and `HashMap` (map) classes, but once you are familiar with these, it will be relatively straightforward to employ any of the other collection classes in your code.

The collection classes form part of the package `java.util` so you will need an appropriate `import` statement in any class where you want to use them.

3.3.1 Generics

In order to use the collection classes you will need to be familiar with the concept of *generics*. A collection should be able to hold objects of any type: we may want a `List` of integer numbers or of `ParticleDetector` objects for example. One way of doing this would be to use variables of type `Object` to refer to every element that we want to store in the `List`: this is possible since all other classes extend `Object`[2]. The problem with this is that we then have to be careful to keep track of what type of object is actually stored in a given collection. Otherwise we run the risk of having a program that compiles but may crash when run, producing a `ClassCastException` when we try to convert an `Integer` into a `ParticleDetector` or vice versa.

Generics help us catch these problems earlier, when compiling the code, by stating explicitly when we create a collection object what type of elements we intend it to contain. The collection classes are *generic types* and the syntax for using them will be introduced in the following section.

3.3.2 Using Lists: The `ArrayList` Class

An `ArrayList` is basically a dynamic 1-D array for storing objects, with some additional features that are not provided by a Java array.

The syntax for creating and filling ArrayLists is illustrated here:

```
// Create ArrayList objects
ArrayList<Point> mypoints = new ArrayList<Point>();
ArrayList<Integer> mynumbers = new ArrayList<Integer>();

for (int x = 0; x < 10; x++){
  for (int y = 0; y < 10; y++) {
    Point p = new Point(x,y);

    // Add new objects
    mypoints.add(p);
    mynumbers.add(x*y);
  }
}
```

Here we see the syntax for creating an instance of a generic type:

```
new ArrayList<Point>()
```

creates a `ArrayList` that can contain objects of type `Point`. The type `ArrayList<Point>` is

called a *parameterized type* and `Point` is the *type parameter*.

A collection can only hold objects, not primitive variables, which is why we have to use `ArrayList<Integer>` and not `ArrayList<int>`. When we add `x*y` to the vector, this value, of type `int`, is automatically converted into the corresponding `Integer` object. This is known as a *boxing* conversion, as mentioned in the notes for Module 2.

Retrieving the contents of a collection is done as follows:

```
Point p = mypoints.get(2);
int val = mynumbers.get(1);
```

The *unboxing* of the `Integer` to the primitive `int` type is also done automatically.

Some other commonly used method of the `List` interface are illustrated here:

```
ArrayList<String> list = new ArrayList<String>();
list.add("hello");     // add to end
list.add("and");       // add to end
list.add("welcome");   // add to end
list.add(2,"another"); // add 3 at position 2
list.set(3,"word");
String o = list.get(1);
list.remove(2);
list.clear();
boolean contains_and = list.contains("and");
int location = list.indexOf("and");
```

You can see that there are variety of intuitively named fill, retrieve and search methods. As always, consult the online Java documentation for the full details.

3.3.3 Looping Over Collections

The collection classes provide versatile methods for looping over their contents. The first technique we will use involves an *iterator*, which provides consistent methods for accessing elements of any collection, including one to remove a given element:

```
Iterator<String> it = list.iterator();
while (it.hasNext()) {
  String s = it.next();
  it.remove(); // removes current element
}
```

The second technique is less flexible and does not allow us to delete elements, but provides a very easy way to loop over a collection or an array:

```
for (String word : list) {
  System.out.println(word);
}
```

3.3.4 Sorting Collections

The `Collections` class has a number of static methods that operate on lists and other collections, for example to sort the elements of a list:

```
ArrayList<Integer> nums = new ArrayList<Integer>();
for (int i = 10; i > 0; i—) {
  nums.add(i);
}
Collections.sort(nums);
```

In the case of numeric types, it's obvious that "`sort`" will arrange the elements in order of numeric value, in this case in ascending order. But Java cannot know how to sort complex user-defined types such as "`Electron`" or "`Oscilloscope`". The programmer must tell the collection how to sort its contents by implementing a comparator class:

```
public class SortIntByAbsoluteValue implements Comparator<Integer> {
  public int compare(Integer arg0, Integer arg1) {

    // The compare method must return:
    // 0       : if arg0 is equal to arg1
    // +ve int : if arg0 > arg1
    // -ve int : if arg0 < arg1

    return (Math.abs(arg0) - Math.abs(arg1));
  }
}

// ...
Collections.sort(nums, new SortIntByAbsoluteValue());
```

Note that in this example we have in fact defined a new type of ordering for the `Integer` type which we could use if we needed to sort by absolute value rather than signed value. If we want to define an ordering for one of our own classes, we also have to override the default `hashCode` method with a version that will always return the same value for objects that are equal. It does not need to always return different values for unequal objects, although it is generally faster to sort objects if this is in fact the case.

3.3.5 Using Maps: The `HashMap` Class

A map is a kind of look-up-table. A look-up table consists of *keys* which point to, or uniquely identify, corresponding *values*. An example look-up table is a dictionary for which the word (the key) is used to look-up the corresponding description or definition (the value). Another example is a unique account number being used to point to a bank record in a piece of financial software.

Note that in Java both the key and the value in each key-value pair must be objects; primitive types are not allowed. Moreover each key must be unique, as illustrated in the following example for inserting pairs into a `HashMap`:

```
HashMap<String, GregorianCalendar> myDates =
            new HashMap<String, GregorianCalendar>();

// Insert using put(<KEY>,<VALUE>):
myDates.put("christmas",
            new GregorianCalendar(2007,Calendar.DECEMBER,25));
myDates.put("birthday-john",
            new GregorianCalendar(2006,Calendar.AUGUST,22));
myDates.put("birthday-jane",
            new GregorianCalendar(2006,Calendar.AUGUST,22));
myDates.put("birthday-john",
            new GregorianCalendar(2006,Calendar.AUGUST,24));

// DUPLICATE KEYS ARE NOT ALLOWED
// Therefore in this example only
// 3 items remain since the same key
// "birthday-john" was used twice:

System.out.println(myDates.size());

// The last pair to use the duplicate
// key is the only one that remains
// in the HashMap.
```

If you know a key value, then retrieving the corresponding value is trivial:

```
GregorianCalendar johnBirthday = myDates.get("birthday-john");

// Note that the value obtained will be
// "null" if the look-up failed, i.e. the
// specified key did not exist in the
// HashMap. An alternative is to check
// first if the required key exists:

boolean fredBirthdayExists = myDates.containsKey("birthday-fred");
if (fredBirthdayExists){
  GregorianCalendar fredBirthday = myDates.get("birthday-fred");
}

// The method "containsValue" can be
// used analogously.
```

The map classes are not collections in the same sense as sets, lists and queues. Looping over the contents of a map can be done in various ways depending on what the programmer is trying to do. There are three methods to provide ways of viewing a map as a collection:

- `keySet()` returns a set of the keys contained in the map;
- `values()` returns a collection of the values contained in the map;
- `entrySet()` returns a set of key-value pairs, represented using the type `Map.Value`.

The same methods described in the section on References can be used to loop over these keys, values or key-value pairs:

```
for (String event : myDates.keySet()) {
  GregorianCalendar date = myDates.get(event);
  System.out.println(event+" date: "+
          date.get(Calendar.DAY_OF_MONTH)+"/"+
          date.get(Calendar.MONTH)+"/"+
          date.get(Calendar.YEAR));
}
```

---

[2] See Module 6 for an explanation, but basically this means that an object of any type, say `String` or `ParticleDetector` can be treated in some ways as an instance of type `Object`. ↩

---

4. Summary

This module has introduced a range of concepts and techniques dealing with objects and collections of objects.

One of the most important ideas to remember is that object variables are *references*, not objects themselves. A variable may not refer to an object at all (if it is `null`) and more than one variable may refer to the same object. You should also now be able to copy and compare objects, and store and retrieve them using arrays and the Java collections framework.

While you may not immediately feel at home with everything described here, you will probably find yourself frequently referring back to the examples from this module and your solutions to the exercises.