

CS1Ah Lecture Note 9

Java: Classes and objects

Java is an *object-oriented* programming language. It uses *objects* to organise and simplify the storage of items of data which should be stored together. Objects belong to *classes* which define the information which an object *instance* of that class contains. The content of these fields can be modified either directly by assignments to these fields or indirectly by *invoking a method* on the object. Java provides many pre-defined classes (one example is the `String` class whose object instances are strings of characters) and a Java programmer can invent and use their own classes. In this lecture note we will develop a series of classes for representing information about University of Edinburgh students.

The relationship of an object to its class is like the relationship of a value to its type although there is a subtle difference. An object reference can be `null`, whereas a value cannot. The value of an `int` variable must always be an integer whereas a `String` object for example will either be a string or `null`. Every string, even the empty string, is non-null and the same is true for every class of object, the *constructed* object instances are non-null.

9.1 A simple class for student data

An object which recorded information about a University of Edinburgh student might record a surname, a forename and a matriculation number. We would represent this as follows in Java.

```
class Student { // 1
    public String surname; // 2
    public String forename; // 3
    public String matricno; // 4
} // 5
```

This is a *class definition*. The identifier of the class being defined is `Student`. Objects of this class will have three fields; a `surname` field, a `forename` field and a `matricno` field. The purpose of the `Student` class is to allow us to store together these three related data items. It seems appropriate to store these three data items together because they are the surname, forename and matriculation number respectively of one person.

In programming languages which do not have objects or classes such a definition is called a *record* or a *structure* (sometimes 'struct'). An object of this class is treated as a *first-class* object in the sense that it can be passed to a method or returned as its result.

We can picture a particular object of this class as shown below. Our object identifier points to an area in computer memory where our three related items of data can be found.

JoSmith →	surname	Smith
	forename	Jo
	matricno	0123456

An object such as the one shown above would be created through a series of Java statements. The first statement would create a Student object. Next the fields of the object are initialised to the values which we wish then to hold. Three assignment statements are used for this purpose. The four Java statements which are needed are shown below.

```
Student JoSmith = new Student(); // create a new object
JoSmith.surname = "Smith";      // set the surname
JoSmith.forename = "Jo";       // set the forename
JoSmith.matricno = "0123456";  // set the matric. number
```

The first of these four statements requires closer attention. The first statement declares a Student object which it constructs using the *default constructor* for the class. We did not define a *constructor method* for the class but because it is essential to be able to create objects of any class the Java language provides a default constructor for us.

9.2 Data validation

The Student class is useful, but it is limited. We have no control over the values which are entered into the fields of a object which holds student data. One of the fields which we might wish to *validate* is the matriculation number field. A matriculation number should have seven digits, so we could include a validation method to check this. We consider an improved class for student records, Student2.

```
class Student2 { // 1
    public String surname; // 2
    public String forename; // 3
    public String matricno; // 4
    public boolean valid() { // 5
        return matricno.length() == 7; // 6
    } // 7
} // 8
```

The valid() method invokes the length() method on the string which stores the matriculation number and tests if this is equal to seven.

9.3 Encapsulation and accessor methods

The validation of data in the way shown above is rather poor. It allows invalid data to be entered and then requires the programmer to check after every update whether or not the data is still valid. A better approach would be to require data to be validated when it is entered and prevent invalid data from ever being entered at all.

This policy of strictly controlled access is quite significantly different from that of the `Student` class which was our starting point. The `Student` class provides unregulated access to the surname, forename and matriculation number fields. This might have seemed to be the most useful policy to adopt when designing a class, to make all of its data items accessible everywhere. The significant disadvantage in doing this is that any of the fields can be altered at any time throughout the program in which the `Student` class is used. An accidental modification of one of the fields would be difficult to trace, since we would have to read all of the program in order to determine at what point the field was modified.

To enforce our access policy we must *encapsulate* information within the class. That is, we mark it as *private* and control access to it via *accessor methods*. The Java language provides the keyword `private` for marking fields or methods as private. The `setMatricno()` and `getMatricno()` methods are the *accessor methods* for the `matricno` field. Because names such as these are used the accessor methods for a field are sometimes known as the ‘get’ and ‘set’ methods.

```
class Student3 { // 1
    public String surname; // 2
    public String forename; // 3
    private String matricno; // 4
    // 5
    public void setMatricno(String matricno) { // 6
        if (matricno.length() != 7) { // 7
            this.matricno = null; // 8
        } else { // 9
            this.matricno = matricno; // 10
        } // 11
    } // 12
    // 13
    public String getMatricno() { // 14
        return matricno; // 15
    } // 16
} // 17
```

The keyword “`this`” introduced here and used on lines 8 and 10 allows an object to make a reference to one of its own fields. The two assignments on lines 8 and 10 assign a value to the matriculation number field of the object, declared at line 4. The assignment on line 10 reads the particular matriculation number from the formal parameter to the `setMatricno()` method, the `String` object introduced at line 6.

9.4 The difference between public and private

To provide greater control over the use of the student record class, with the aim of ensuring improved data integrity, we can mark the other fields of the class as being **private**. A private field may be accessed from any method *within* the class definition (we saw above that the private `matricno` field was used in the `getMatricno()` method) but it cannot be accessed from any method *outside* the class definition.

When a field is marked as being private we cannot access it using the dot notation for field access which we saw earlier. That is, if the `surname` and `forename` fields are private then we cannot use the notation `JoSmith.surname` or `JoSmith.forename` either to assign to these fields or to inspect the values which they currently hold.

9.5 Constructors and printing

Having taken the decision to make the fields of `surname` and `forename` private we must provide some (controlled) way to set their values. This is typically done through the use of the *constructor* method for the class, which we can program ourselves.

One of the disadvantages of allowing the fields previously to be accessed publicly was that we could create a `Student` object and then forget to initialise the fields (or forget to initialise some of them). If we wish to have all of the fields defined when the student data object is created then we can insist that they are all provided when the object is constructed. In this case the fields need not be made available for assignment and we can provide only *inspector* methods for them.

The three inspector methods are called `getSurname()`, `getForename()` and `getMatricno()`. These are very simple to define: they simply return the value of the corresponding field of the class. However, the fact that these methods do relatively little work should not lead us to conclude that they are not useful. They are the only way in which we can provide one-way access to these fields, allowing their values to be inspected without allowing them to be updated.

The constructor method for a class always has the same name as the class itself. Thus in the following example we have a class `Student4` and a constructor `Student4()`. The constructor returns an object of the class by initialising the fields of the object. Note that there is no return type for the `Student4()` method (in particular it is not *void*). We know that this method has the purpose of constructing a `Student4` object because it has the same name as the class itself. Our constructor for the class will have three *parameters*: one for the `surname`, one for the `forename` and one for the matriculation number. An example use of this constructor method is shown below.

```
Student4 JoSmith = new Student4("Smith", "Jo", "0123456");
```

Any attempt to invoke the constructor for this class with fewer than three parameters or more than three parameters will be rejected by the Java compiler when the program is compiled. Now that we have provided our own specialised constructor Java does not make available a default constructor for the class.

```

class Student4 { // 1
    private String surname; // 2
    private String forename; // 3
    private String matricno; // 4
    // 5
    // the following method is the constructor // 6
    public Student4(String surname, // 7
                   String forename, String matricno) { // 8
        this.surname = surname; // 9
        this.forename = forename; // 10
        if (matricno.length() != 7) { // 11
            this.matricno = null; // 12
        } else { // 13
            this.matricno = matricno; // 14
        } // 15
    } // 16
    // 17
    public String getSurname() { // 18
        return surname; // 19
    } // 20
    // 21
    public String getForename() { // 22
        return forename; // 23
    } // 24
    // 25
    public String getMatricno() { // 26
        return matricno; // 27
    } // 28
    // 29
    public boolean equals (Student4 stu) { // 30
        return (stu.getMatricno().equals(matricno) && // 31
                stu.getForename().equals(forename) && // 32
                stu.getSurname().equals(surname)); // 33
    } // 34
    // 35
    public String toString() { // 36
        return surname.toUpperCase() + ", " + // 37
                forename + " (" + matricno + ")"; // 38
    } // 39
} // 40

```

Since we have taken this opportunity to make a number of improvements to the class we could also go further. There is another important method which we can add to our class, a method named `toString()`. A method of this name is invoked when we need to convert an object to a string (say for printing on the screen). We can format the string representation in any way that we wish, say with the surname first, in upper case and then the forename second after a comma. The matriculation number can

come last, enclosed in brackets. When we invoke the `System.out.println` method the `toString()` method of the class of the object is used to convert it to the string, which is printed on the screen. The Java statement `System.out.println(JoSmith);` would cause the line of text `SMITH, Jo (0123456)` to be printed on the screen.

9.6 When are objects equal?

Consider the following lines from a Java program.

```
Student4 Jo      = new Student4("Smith", "Jo", "0123456");
Student4 Josie  = new Student4("Smith", "Jo", "0123456");
```

Given these declarations, are the objects `Jo` and `Josie` the same? The answer to this question tells us the meaning of the word “**new**” when it appears in a Java program. `Jo` and `Josie` are *different* objects. Because we asked for them to be newly created objects of the `Student4` class they are not the same, even though they have identical contents in their three fields (the same surname, the same forename and the same matriculation number).

If we were to look into the contents of our computer’s memory after creating these two objects we would see that we had two copies of the information. One copy is accessed through the identifier `Jo` and the other copy is accessed through the identifier `Josie`.

Jo →	surname	Smith
	forename	Jo
	matricno	0123456
Josie →	surname	Smith
	forename	Jo
	matricno	0123456

Sometimes we wish to ask if two objects have equal contents, even if they were created at different points in the program. Java provides two ways of testing objects for these two notions of equality, the `==` operator and the `equals()` method. By default, these both test for identity (*i.e.* whether these are the same object in memory). However, many of the standard classes redefine their own `equals()` method, usually to test if the contents of the two objects are the same (but possibly stored as separate copies). We have done so for `Student4`. In the cases of the `Jo` and `Josie` objects above we have this:

```
Jo == Josie           returns false
Jo.equals(Josie)     returns true
```

By assigning one object to another we will make them identical. If we execute this Java assignment

```
Jo = Josie;
```

and we were then to look into the contents of our computer’s memory after executing this assignment statement we would see that we had still had two copies of the object but that one of them was now inaccessible (it is now *garbage*). Both object identifiers are now referring to the same object instance.

	surname	Smith
	forename	Jo
	matricno	0123456
Jo, Josie →	surname	Smith
	forename	Jo
	matricno	0123456

It might seem rather worrying that we could fill up the memory of our computer by making copies of objects which then become inaccessible. As our program ran we would use up more and more memory until eventually we had none left. At this point the program would stop running (if this happens then the program releases all of the memory that it has acquired). However, this problem is unlikely to occur at all when programming in the Java programming language because the Java run-time system periodically runs a *garbage collector* which identifies inaccessible objects and frees up the memory that they were holding. This memory is then returned to the common pool in order that it can be allocated for some other object. In this fashion the memory used by a running Java program is constantly being recycled on our behalf in order that our programs will not run out of memory.

If we now repeat the two tests for the two notions of equality *after* executing the assignment `Jo = Josie` we then find the following.

```
Jo == Josie           returns true
Jo.equals(Josie)    returns true
```

If two objects are identical then they must have identical contents so once the `==` operator has returned true then we know that the `equals()` method will return true also. However, if the `==` operator returns false then the `equals()` method might return true or it might not, we need to evaluate it to find out. In the context of its use with objects, the `==` operator is sometimes called *pointer equality*, because it is comparing pointers into computer memory where the fields of the object are stored. The form of equality testing which is provided by the `equals()` method is said to be testing *content equality*.

As we have seen, classes can define their own notions of object equality. An alternative for use with the `Student4` class is shown below.

```
public boolean equals(Student4 stu) {
    String matricno = stu.getMatricno();
    return this.matricno.equals(matricno);
}
```

That is, two student records would be considered equal if they have the same matriculation numbers. This test captures the right notion of equality for student data records and is also an efficient implementation of the equality test. It is obviously faster to compare just one string for equality than it is to compare three.

9.7 Numeric objects and comparison operators

Java also provides classes which are used to create objects which have numerical values in their fields. Sometimes when manipulating numeric objects we wish to

test them for ordering on their relative values instead of just for equality. For this we use the `compareTo()` method which returns three possible results, `-1`, `0` and `1`. These three values mean respectively 'less than', 'equal' and 'greater than', signifying a negative difference, zero difference and a positive difference. The following program uses the `equals()` and `compareTo()` methods on arbitrary precision integers (the class `BigInteger`).

```
import java.math.BigInteger; // 1
class Comparisons { // 2
    public static void main(String[] args) { // 3
        BigInteger a = new BigInteger("31"); // 4
        BigInteger b = new BigInteger("31"); // 5
        BigInteger c = new BigInteger("45"); // 6
        System.out.println(a == b); // prints false // 7
        System.out.println(a.equals(b)); // prints true // 8
        System.out.println(c.compareTo(b)); // prints 1 // 9
        System.out.println(a.compareTo(b)); // prints 0 // 10
        System.out.println(b.compareTo(c)); // prints -1 // 11
    } // 12
} // 13
```

Murray Cole, 25th October 2002.