

CS1Ah Lecture Note 8

Anatomy of Java programs

We have now seen examples of expressions (Lecture Note 5) and statements (Lecture Note 6) which occur in Java programs. In this note we consider complete programs again, and see the use of expressions and statements in context. We also consider Java's support for organising programs using classes, packages, and methods.

8.1 The main method

All Java applications must contain a `main()` method. The name `main` is distinguished in that it is the name of the method which the Java interpreter will execute first, regardless of where it occurs in the program. The `main()` method must be declared to be **public** and **static**. The modifier **public** means that the method is accessible from any other class; the modifier **static** means that the method is associated with the class rather than any object of the class. Static methods are also known as *class methods*.

The `main()` method accepts an *array* of strings of characters which are the command line arguments to the program when it is interpreted. The result of executing the `main()` method is declared to be *void* because it does not return a result such as an integer or a boolean or a string.

```
1  public class One {
2      public static void main(String[] args) {
3          int one;
4          one = 1;
5          System.out.println("Java program " + one);
6      }
7  }
```

Line 3 declares an integer variable `one` which is assigned the value 1 on line 4. Line 5 prints the string `Java program 1` on the screen. The operator `+` is used to mean several different things in Java: this practice is known as *overloading*. We already know that this denotes both integer addition and floating-point addition so there is one overloading already but the plus symbol also denotes *concatenation* (joining together) of strings. The value 1 is retrieved from the variable `one` and then coerced into the string `"1"` before the concatenation is performed.

Because the class `One` is declared `public`, Java requires that it is stored in a file named `One.java` (and this convention is good practice anyway). It is compiled with the command `javac One.java` (the `.java` file name extension *must* be included). This compilation produces a file called `One.class` which is executed with the command `java One` (the `.class` file name extension *must not* be included).

Exercise: Consider what would be the effect of removing any of lines 3 to 5. Check your reasoning by compiling the modified program and inspecting any error messages which occur.

8.2 Constants: variables with final values

In the previous program the variable `one` is given the value 1 by an assignment. The use of this variable is only to hold the value 1, and given the name, it would be misleading for it to hold any other value. A variable whose value cannot vary is termed a *constant* and in Java it is marked as `final` when it is declared. Line 3 of the program below fixes the final value of the variable `one`.

```
1 public class Two {
2     public static void main(String[] args) {
3         final int one = 1;
4         System.out.println("Java program " + (one + one));
5     }
6 }
```

Exercise: Insert the assignment `one = 1;` after line 3 above and find out what the Java compiler does with this. What would be the effect of removing the brackets around `one + one`? Check your reasoning by compiling and running the modified program. The effect might be slightly surprising.

Marking a variable as having received its final value is not limited only to integers. Other types of variables can be declared as `final` also. In the program below, the string variable `three` is marked as `final`. Once again, this means that its value cannot be changed by an assignment. When the program below is executed it prints the text `Java program three` on the screen.

```
1 public class Three {
2     public static void main(String[] args) {
3         final String three = "three";
4         System.out.println("Java program " + three);
5     }
6 }
```

8.3 Structuring in the large: packages

Java's primary structuring construct is the notion of *class*: all programs are organised as a number of classes. Often, several classes will be used together, with some of them defining classes of objects which are built upon by others. If these classes depend so strongly on each other that it is appropriate to think of them as part of a larger entity, then they are brought together into a named collection, called a *package*.

Packages are useful because they map out the structure of a software system in terms of components of significant size. The Java platform itself is built up of many packages. For example, the package `java.lang` contains class definitions for core language classes such as `String`. The package `java.math` contains class definitions which provide mathematical functions such as arbitrary precision arithmetic, in the `BigInteger` class. Classes have *fully qualified* names which identify the package which they come from. The fully qualified name of `String` is `java.lang.String` and the fully qualified name of `BigInteger` is `java.math.BigInteger`.

Package names help to prevent *identifier name clashes* where two parties working independently coin the same name for different classes. This problem does not appear until the whole system is built from its constituent parts; programming languages without a package concept can give unpredictable results when names clash. Package names also help in code reuse. Java developers are encouraged to name their packages using their Internet domain name, preventing name clashes on an Internet scale.

Any class in a package can be referred to using its fully qualified name, but this quickly can become tedious. To allow convenient shorthands, and to act as a signal that some class is being used, Java allows a program to begin with **import** declarations. Notice that Java automatically imports the `java.lang` package: this is why we can write just `String` instead of `java.lang.String`.

Line 1 of the following program imports the `Locale` class from the `java.util` package. This class is used for representing geographical regions which often have associated spoken languages. A new locale can be defined by the use of the `Locale()` constructor, which is given two strings as arguments in line 4. The first is the name of the language spoken and the second is the name of the country.

```
1  import java.util.Locale;
2
3  class Four {
4      public static void main(String[] args) {
5          Locale Edinburgh = new Locale("English", "Scotland");
6          System.out.print(Edinburgh.getDisplayCountry());
7          System.out.println(": a great place to program!");
8      }
9  }
```

One of the methods which a locale provides is `getDisplayCountry()`. The effect of invoking this method on the `Edinburgh` object is to return the string `"SCOTLAND"`. The methods `print()` and `println()` provided by the `System.out` object differ in that the latter takes a new line after printing its string of characters to the screen. Thus the output produced by the program is `SCOTLAND: a great place to program!`.

8.4 Structuring in the small: methods

Where packages are used for the large-scale structure of programs, methods inside classes and objects provide a small-scale structuring mechanism. We will soon study examples of instance methods belonging to objects, used for object-oriented programming. But in some cases, class methods, belonging to classes, are appropriate instead. A class method is declared **static**, and is invoked using a class name instead of an object name. A typical example is `Math.sqrt()`. Mathematical functions which operate on primitive types are often declared static because they do not refer to any object. Notice that Java's naming convention — class names should begin with an uppercase letter — helps to suggest when a method being invoked is static.

Below, we introduce a static method which operates on a `Locale` object. A static method is appropriate here because we are defining some fixed external behaviour which is not intrinsic to the concept of locale (unlike, *e.g.*, `getDisplayCountry()`).

```
1  import java.util.Locale;
2  class Five {
3      final static String Scotland = "Scotland";
4      final static String English = "English";
5
6      static void report(Locale loc) {
7          String country = loc.getDisplayCountry();
8          if (country.equalsIgnoreCase(Scotland)) {
9              System.out.println(Scotland + " the brave!");
10         } else {
11             System.out.print(country);
12             System.out.println(": I'd rather be in Scotland!");
13         }
14     }
15     public static void main (String[] args) {
16         Locale edinburgh = new Locale(English, Scotland);
17         report(edinburgh);
18         report(Locale.UK);
19         report(Locale.US);
20     }
21 }
```

The `report()` method has a locale *formal parameter* named `loc` and a string *local variable* named `country`. Both of these names are visible only within the body of the `report()` method. This is an important advantage of introducing methods into a program: each method can define and use variables which are entirely within their control, and which cannot be interfered with by other methods in the program. Fundamentally, the new method avoids needing to repeat code: the `report()` method is invoked three times from the body of the `main()` method, with a different *actual parameter* each time.

*Note originally by Stephen Gilmore. Revised by David Aspinall.
David Aspinall, 2002/10/23 16:37:45.*