4 Static Methods: Procedures

1 Review of static methods

A method is a piece of code, neatly wrapped up. It consists of a header as in

```
public static void main(string[] args)
```

followed by the code that constitutes the body of the method in chain brackets. All our programs thus far have been comprised of a class with a single method called main(). Method main() is *invoked*, i.e. caused to be executed, by entering an appropriate command at the command line (such as `java MyClass`).

A class may contain several methods. Methods other than main() are invoked not from the command line but from within main(). Indeed the additional methods may in turn invoke still other methods, and so on. Apart from main(), a method may be invoked many times, as we shall see. We like to organise a program as a collection of methods to give it structure, i.e. to break it into mind-sized chunks each of which can be understood and tested more or less as an independent mini-program.

Methods come in two kinds: they may be *static* or *instance* (also called *dynamic*). You can tell a static method from a dynamic method by the presence of the word `static` in the header. For example, the following are headers of static methods:

```
public static void main(string[] args)
static void drawLine(int n)
static double squareRoot(double x)
```

Static methods are much easier to understand that dynamic methods; for the moment we study static methods only.

There are two flavours of methods (whether static or dynamic): *procedures* and *functions*. Procedures *do something* while functions *evaluate something*. You can tell a procedure from a

function by the presence of the word `void` in the header. For example, the following are headers of procedures:

```
public static void main(string[] args)
static void drawLine(int n)
```

Functions, on the other hand, contain a type name in place of the word `void`, as in

```
static double squareRoot(double x)
static String surname(String name)
```

Procedures effect a change in the world. For example, they display something on a screen, or change the values of variables, or change the contents of a file, or delete a file from a disk. Functions on the other hand merely inspect the world without changing it. The result of their inspection is a value of the type mentioned in the header. For example, invoking a function with header

```
static double squareRoot(double x)
```

computes a value of type double. Another way to describe the difference between procedures and functions is to say that procedures are like complex statements while functions are like complex expressions.

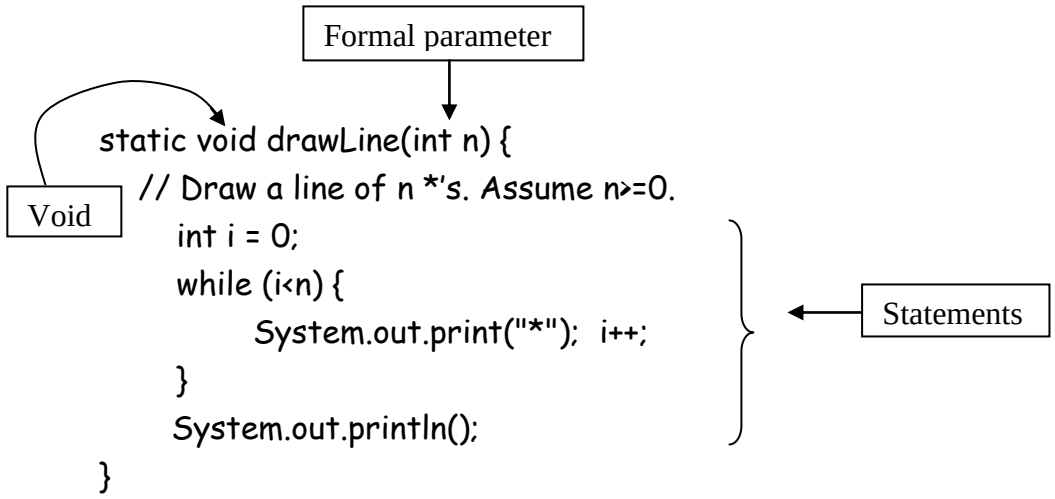
A method may have an access qualifier (one of `private` or `public`) , as in:

```
public static void main(string[] args)
private static double squareRoot(double x)
```

The access qualifier is not important in small programs and you need not worry about it for the moment. Method main() must include the access qualifier, but otherwise your programs will work if you omit all access qualifiers.

2 Procedures

The text of a method is called its *definition* or *declaration*. Below is a small example of a static procedure definition:



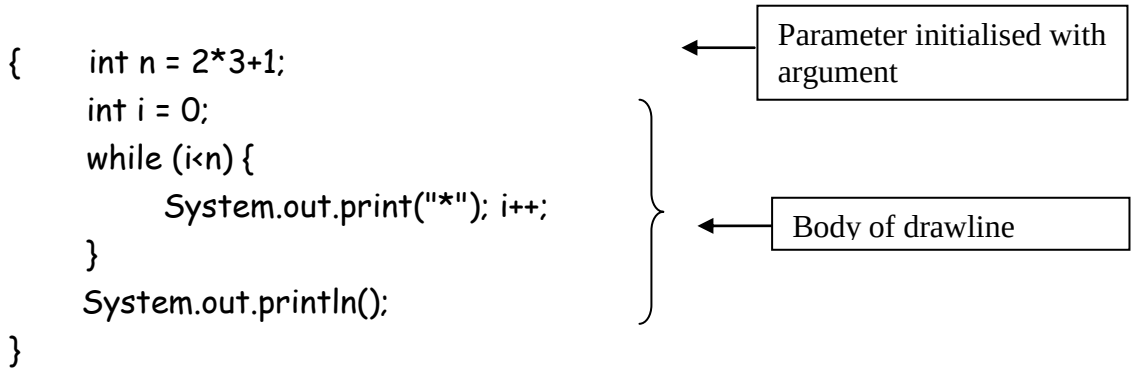
This procedure includes a single "(formal) parameter", here called n. A parameter is a local variable which is declared in the header to indicate that it is to be initialised at the point of invocation. A static procedure is invoked, i.e. its parameters are initialised and its constituent statements are executed, by an invocation statement in another method (except that main() is implicitly invoked from the command line). An invocation statement consists of the procedure name followed by the initial value for the parameters in brackets. A invocation of drawLine above, for example, with an indication that its parameter n is to be initialised to 5, say, is

```
drawLine(5);
```

The initial values for parameters, supplied at invocation, are called "arguments" or "actual parameters". Arguments can be any expression, as long as the expression matches the type of the associated parameter. For example, the following is legitimate:

```
drawLine(2*3+1);
```

The effect of the above statement is exactly that of executing the following code:



... and similarly for any argument. For more than one parameter, each one is initialised with the matching argument in the invocation.

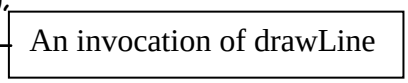
Remember that a procedure invocation is a statement. A procedure may be invoked many times (although it is declared/defined just once).

Below is a simple program which employs more then one method. It reads a succession of (natural) numbers and for each one draws a line of asterisk's of that length.

```
class DrawClass {

        static void drawLine(int n) {
           // Draw a line of n *'s. Assume n>=0.
              int i = 0;
              while (i<n) {
                    System.out.print("*"); i++;
              }
              System.out.println();
        }

        public static void main(String[] args) {
               System.out.println("Enter one number per line ....");
               while (!Console.endOfFile()) {
                      int num = Console.readInt();
                      drawLine(num);
               }
        }
}
```



The order in which methods are declared in a class is not significant.

Actually, the full name of a method includes the class name as a prefix followed by a period, as in DrawClass.drawLine(num). However, the class name is usually omitted when the method is declared in the same class as the invocation, as above.

More on parameters

If you are not comfortable with parameters, study them again in your textbook – they will be much used in the course, and you will find life difficult if you do not come to terms with them early. For the case of zero or multiple parameters, refer to the textbook.

Variables which are declared within a method are said to be "local" to the method. They are created during the execution of the method, and do not survive after the invocation has completed. The same holds true for the method's parameters, of course – the only thing special about parameters is their manner of initialisation.

If a method is invoked repeatedly, its local variables and parameters are created anew for each invocation, and they die when the method has completed its execution for that invocation. The final value of a local variable does not carry over to any following invocation of the method. Note that assignments to a parameter can never affect the associated argument. Consider, for example:

```
static void bump(int n) {
        n = n+1;
}
...
        int k = 3;
        bump(k);
        System.out.print(k);
```

What value appears on the screen – 3 or 4? The answer is 3, because n is a local variable in bump(), and an assignment to it cannot have any effect on the value in k.

Return

Procedures terminate and return control to the point of invocation when the final statement in the procedure is executed. The statement

```
return;
```

terminates execution of a procedure prematurely and returns control to the point of invocation (which will be the command line if return is executed within main()). Do not confuse this return statement, with the return statement of functions – the return statement for functions has an accompanying value, as we shall see.

The parameter-read trap

One common mistake among beginners is to think that parameters of methods must be initialised at the start of the method by reading in values for them from the keyboard. For example, some beginners might erroneously declare drawLine() as follows:

```
static void drawLine(int n) {
// Draw a line of n *'s. Assume n>=0.
       n = Console.readInt();
       for (int i=0; i<n; i++) {
              System.out.print("*");
       }
       System.out.println();
}
```

WRONG! Parameters are initialised at invocation, not by reading.

Remember: parameters are given their values at the point of invocation.