2 Static Methods and Strings

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, the 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.

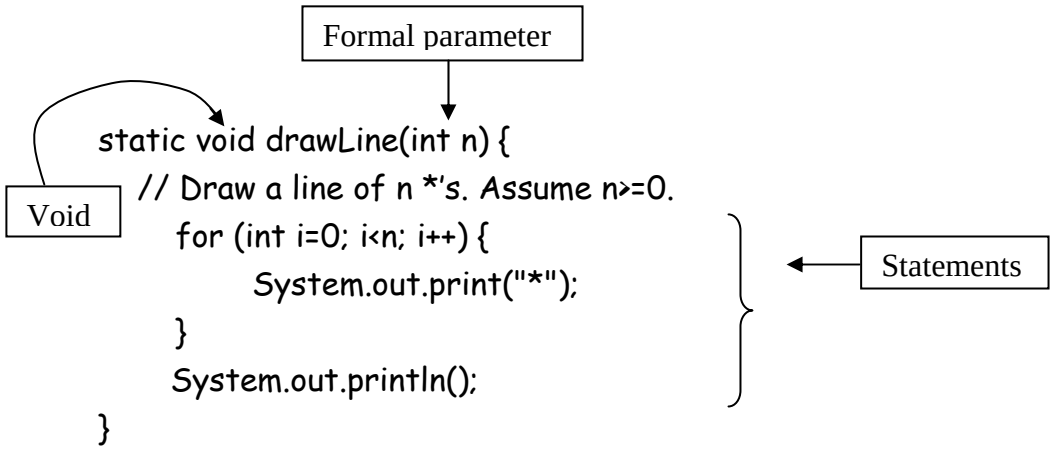
A method may have an access qualifier (one of `private` or `public`) , as in:

```
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);
```

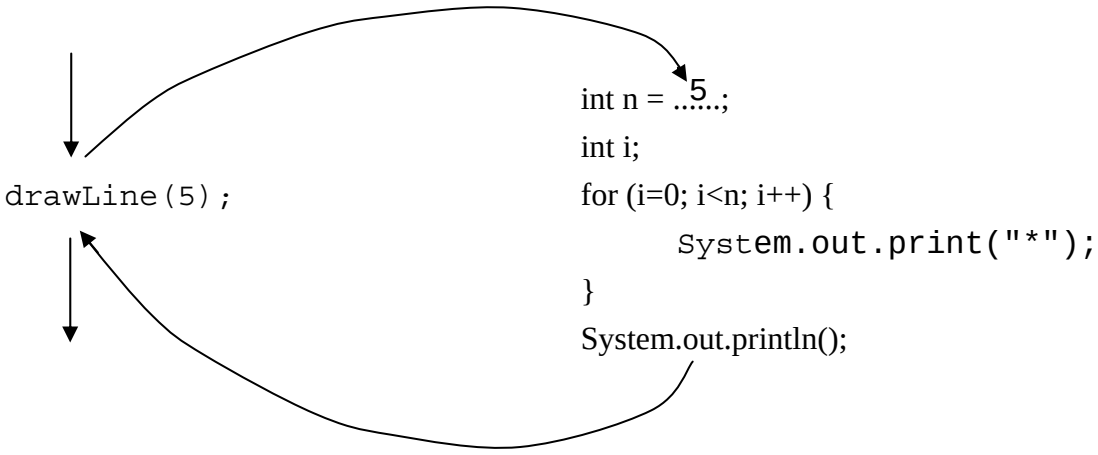
The initial values for parameters, supplied at invocation, are called "arguments". (Arguments are also called "actual parameters", in contrast with the "formal parameters" that occur in the method header.) Arguments can be any expression, as long as they match the type of the associated parameter. For example, the following is legitimate:

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

When an invocation statement is encountered by the system

1. It suspends execution of the current method.
2. It executes the invoked procedure, with its parameters initialised with the supplied arguments.
3. When the invoked procedure completes, the execution of the invoking method is continued.

The effect of invoking drawLine(5) can be depicted thus:



Remember that a procedure invocation is a statement. A procedure may be invoked many times (although it is declared 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.
              for (int i=0; i<n; i++) {
                    System.out.print("*");
              }
              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);
               }
        }
}
```

An invocation of drawLine

The order in which methods are declared in a class is not significant. In fact static methods can be declared in separate classes. However, to invoke a static method declared in another class, the name of the method must be prefixed with the name of the class in which it is defined, followed by a period, as in the following version of the preceding program.

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

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

Note class name prefixed

However, in this case there is nothing to be gained by placing drawLine in a separate class – the first version is simpler.

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 would 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.

3 Functions

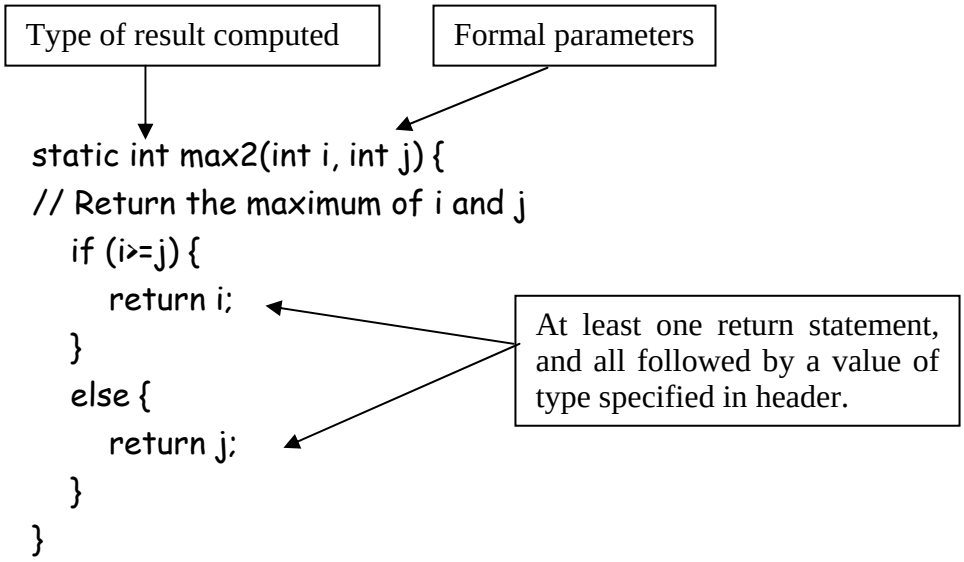
A function does not carry out an action (such as printing) but computes a value of a particular type. The type of the value returned is called the *return type* of the function and is included in the header in place of the `void` of procedures. For example, the return type of the following function is `double`:

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

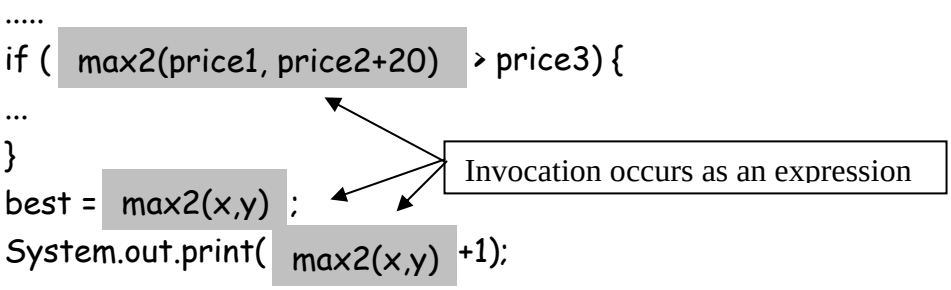
The value is returned to the point of invocation by the execution of a return statement in the body of the function, where the return statement includes an expression of the return type. For example, if the return type is int, then the function will contain at least one statement of the form

```
return 3*4;
```

(of course, 3*4 here can be replaced with any integer expression). Every function must have at least one return statement ending in an expression, but it may have more than one. In other respects, static functions are declared like static procedures:



A static function is invoked by using its name (plus arguments in brackets) as an expression of the type stated in the header:



Note the contrast with procedures: procedures are invoked using an invocation statement, whereas functions are invoked by the appearance of the name as an expression. When a function invocation is encountered by the system:

1. It suspends execution of the current method.
2. It executes the invoked function, with its parameters initialised with the supplied arguments.
3. In the invoking method, it substitutes the value computed by the function for the function invocation.
4. It continues with the execution of the invoking method.

If you do not understand this, study it further in the textbook.

If a static function name is invoked from outside the class where it is declared, it must be prefixed with its class name and a period, just as with procedures. For example, the function round() which we met previously is a static function defined in a class called Math, and so a typical invocation is Math.round(3.14).

The return type of a function can be any type at all. For example, the function endOfFile() in class Console is a static function whose return type is boolean. If the return type is integer then we say that the function is "integer-valued"; if the return type is boolean then the function is boolean-valued, and so on.

The result-write trap

As with procedures, a common mistake among beginners is to think that parameters of a function must be initialised at the start of the function by reading in values for them from the keyboard. This is not so: parameters acquire their initial values from the corresponding arguments at each invocation. Another error made by beginners is to believe that the result computed by a function must be printed. This is not so: a function computes a value which it returns it to the caller:

```
static int max2(int i, int j) {
// Return the maximum of i and j
   if (i>=j)
      System.out.print(i);
   else
      System.out.print(j)
}
```

WRONG! This is a function, & so results not printed, but returned!

Example

To illustrate functions in a complete program, we write a program to read the lengths (in centimetres, say) of three rods, and determine whether or not the rods can be used to make a triangle. Three rods form a triangle if the longest length is less than the sum of the other two. An example of input/output for the program is

```
Enter three positive lengths: 30 10 15
You don't have a triangle
```

Another is:

```
Enter three positive lengths: 3 4 5
You have a triangle
```

In the program, we employ three functions, one to compute the maximum of three integers, one to find the sum of the two smallest integers from three given integers, and a boolean-valued function to determine whether three lines constitute a triangle. The invocations of the functions are highlighted with boxes. Note that one of the function max() is invoked from two locations. Of course, we could have written the program using just method main(), but we deliberately employ functions for illustrative purposes, and in any event the use of auxiliary functions gives the program a more understandable structure as a collection of small parts.

```
class Triangle {

   static int max(int i, int j, int k) {
   // Return the maximum of i, j, and k
      if (i>=j && i>=k)  // i is the maximum
         return i;
      else if (j>=i && j>=k)  // j is the maximum
         return j;
      else  // k is the maximum
         return k;
```

Invoked by sumSmall() & isTriangle()

```
   }

   static int sumSmall(int i, int j, int k) {
   // Return the sum of the smaller two among i, j, and k
      return i+j+k- max(i,j,k) ;
   }

   static boolean isTriangle(int i, int j, int k) {
   // Do positives i, j, k constitute a triangle?
      return max(i, j, k)  <  sumSmall(i, j, k)  ;  // see note below
   }

   public static void main(String[] args) {
      int b,c,d;
      System.out.print("Enter three positive lengths: ");
      b = Console.readInt(); c = Console.readInt(); d = Console.readInt();
      if ( isTriangle(b, c, d) )
         System.out.println("You have a triangle");
       else
         System.out.println("You don't have a triangle");
   }
}
```

Invoked by isTriangle()

Invoked by main()

Invoked by command line

Some students would be inclined to write the body of isTriangle() as

```
      if (max(i, j, k) < sumSmall(i, j, k))  return true;
      else return false;
```

This is not incorrect, but it is needlessly clumsy.

Hybrid procedure/functions

Occasionally, a procedure returns a result as well as carrying out an action. The most common example of such hybrids is when a procedure wishes to indicate to the invoker whether or not it carried out its action successfully, and does so by returning a boolean. Below is a small example. It is another version of drawLine() which checks whether the length of the line to be drawn is not too small or too big.

```
      static boolean drawLine(int n) {
         // Draw a line of n *'s
```

```
    if (n<0 || n>80)
        return false; // indicate failure
    else {  // draw the line
       for (int i=0; i<n; i++)  {
          System.out.print("*");
       }
       System.out.println();
       return true; // indicate success
    }
}
```

A typical use of this is:

```
public static void main(String[] args) {
        .............
        int num = Console.readInt();
        boolean success = drawLine(n);
        if (!success) System.out.print("No can do");
        .............
}
```

Invocation is an expression....

An invoker who is not interested in the value returned can treat the hybrid just as a normal procedure:

```
int num = Console.readInt();
drawLine(num);
```

Or invocation is a statement....

-- the boolean value returned is ignored in this case.

Very occasionally, a function may effect a change as well as computing a value. The most common example of such hybrids are methods such as Console.readString(), Console.readInt() etc. which read a value from the keyboard and return it, but also display the value entered on the screen. Again, such hybrids can be used as procedures by ignoring the value returned. For example, the *statement*

```
Console.readString();
```

carries out the useful role of pausing the program until the user presses the return key – any text entered on the line is ignored.

4 Example: Printing a calendar

We tackle a modestly large programming problem, of a size and complexity that requires us to structure it as a collection of methods. We will write a program which prints a calendar for any year. We will restrict the year to the range 1900 to 2099, just to simplify the leap year calculations. The following is an outline of the program:

```
class Calendar {

  static void yearHeader(int y) {
  // Display calendar header for year y, such as
  //          Calendar 2001
     ......................
  }

  static void monthHeader(int m) {
  // Display month header for month M, such as
  //           February
 //  Sun   Mon   Tue   Wed   Thu   Fri   Sat
     ......................
  }

  static int firstDay(int y) {
  // Calculate day of week on which 1/1/y falls
  // Sunday = 0, Monday = 1, etc.
     ......................
  }

  static int daysInMonth(int m, int y) {
  // Calculate number of days in month m in year y.
  // (Note year is only significant for month 2 (i.e. February))
     ......................
  }

  static void putMonth(int d, int s) {
  // Display the calendar for a month containing s days, where
  // the first of the month falls on day d (Sunday = 0, etc.).
  // E.g. for a month of 28 days beginning on a Thursday:
 //
  //                       1    2    3
```

```
  //      4   5   6   7   8   9  10
  //     11  12  13  14  15  16  17
  //     18  19  20  21  22  23  24
  //     25  26  27  28
     ......................
  }

  static void putCalendar(int y) {
  // Display a calendar for year y, y>=1900.
       print the year header (use yearHeader());
       calculate day of week d of first day (use firstDay());
       for each m in range 1 to 12 {
            print header for month m (use monthHeader());
            compute number of days s in m (use daysInMonth());
            print a month of s days starting on day d (use putMonth());
            add s to d modulo 7; // (j modulo k is remainder on dividing j by k)
       }
  }

  public static void main(String[] args) {
  // Read a year (from 1900) and print its calendar
       use putCalendar()
  }

}
```

The complete program is presented below.

```
class Calendar {

  static void yearHeader(int y) {
  // Display calendar header for year y
     System.out.print("          CALENDAR " + y);
     System.out.println(); System.out.println();
  }

  static void monthHeader(int m) {
  // Display month header for month m
     String month;
     // Can be done more succinctly using arrays
```

```
    if (m==1) month = "January  ";
    else if (m==2) month = "February";
    else if (m==3) month = "March";
    else if (m==4) month = "April";
    else if (m==5) month = "May";
    else if (m==6) month = "June";
    else if (m==7) month = "July";
    else if (m==8) month = "August";
    else if (m==9) month = "September";
    else if (m==10) month = "October";
    else if (m==11) month = "November";
    else month = "December";
    System.out.println("          " + month);
    System.out.println("  Sun Mon Tue Wed Thu Fri Sat");
}

static int firstDay(int y) {
// Day of week on which 1/1/y falls (Sunday = 0, etc.).
    return (((y-1900)*365 // elapsed days since 1/1/1900
          + (y-1901)/4  // not forgetting leap days
          + 1           // 1/1/1900 fell on a Monday
       ) % 7);          // 7 days in a week
}

static int daysInMonth(int m, int y) {
// Number of days in month m in year y
        if (m==9 || m==4 || m==6 || m==11)  return(30);
        else if (m==2) { // catch leap year
                if (y%4==0 && y!=1900) return(29);
                else return(28);
        }
        else return(31);
}

static void putMonth(int d, int s) {
// Display a month of s days, the first of the month
// falling on day d (Sunday = 0, etc.).
    int date, day, k;
    // indent first line of month by d positions
    for (k=1; k<=d; k++) System.out.print("    ");
```

```
      day = d;    // day of week
      for (date=1; date<=s; date++) {
         if (date<10) System.out.print("   " + date); // extra space
         else System.out.print("  " + date);
         day = (day+1) % 7;
         if (day==0 && date<s) // another line of days to come
             System.out.println();
      }
      System.out.println(); System.out.println();
   }

   static void putCalendar(int y) {
   // Display a calendar for year y, y>=1900.
      int m, d, s;
      yearHeader(y);
      d = firstDay(y);
      for (m=1; m<=12; m++) {
         monthHeader(m);
         s = daysInMonth(m, y);
         putMonth(d, s);
         d = (d+s) % 7;
      }
   }

   public static void main(String[] args) {
   // Read a year (from 1900) and print its calendar
     int y;
     System.out.print("Enter a year in range 1900..: ");
     y = Console.readInt();
     putCalendar(y);
   }

}
```

5 Local and global variables

A variable that is declared within a method is said to be *local* to the method. It cannot be accessed by code in a different method. Consider:

```
      class MyClass {
```

```
    public static void main(String[] args) {
        int x;
        x = 0; // this is ok
        .....
    }

    static void p() {
        x++; // this is illegal – x belongs to a different method
        .....
    }
}
```

Local variables are created each time the method is invoked, and destroyed each time it terminates. They do not carry over their value from one invocation to the next. Consider:

```
class MyClass {

    public static void main(String[] args) {
        silly(); silly(); silly();
    }

    static void silly() {
        int count = 0;
        count++;
        System.out.print(count + " ");
    }
}
```

The above program does not cause 1 2 3 to be printed, but 1 1 1.

Variables can be declared in a class rather than locally in a method; such variables are said to be *global* to the methods in the class. Variable countCalls below is global to both methods in the class, i.e. it is accessible from both methods, and it carries over its value between method invocations:

```
class MyClass {

    static int numCalls = 0; // for number of method invocations
```

```
        static void p() {
                numCalls++; // record another invocation
                .....
        }

        static void q() {
                numCalls++; // record another invocation
                .....
        }

        public static void main(String[] args) {
                ..... p(); .....; q(); ..... p(); ....
                ..... q(); .....; q(); ..... p(); ....
                ......
                System.out.print("Number of invocations = ", numCalls);
        }

}
```

Note that numCalls is defined to be static. Global variables can also be dynamic (also called "instance variables") – and indeed this is more common; we'll meet them later.

6 Review of characters and strings

Characters

Characters includes the letters of the alphabet in both upper and lower case, the decimal digits, punctuation characters, other common and not so common symbols (such as @ or #), and various control characters (such as the new line character, written \n, which when printed causes the cursor to move to the start of a new line). The character type in Java is written `char`. Character values are bracketed by single quote marks, as in `char ch = 'X';`.
The following are some of the more useful operations on characters. Each takes a character argument (called ch below) and yields a boolean value as indicated by the name (e.g. Character.isDigit(ch) yields true if ch is a digit).

Character.isDigit(ch)
Character.isLetter(ch)
Character.isLetterOrDigit(ch)
Character.isLowerCase(ch)

Character.isUpperCase(ch)
Character.isWhitespace(ch)

For example, Character.isLetter('b') yields true, whereas Character.isLetter('1') yields false. Note that Character.isDigit(1) is incorrect – you must write Character.isDigit('1'). The following operations also take a character argument, and yield a character as result.

Character.toLowerCase(ch)
Character.toUpperCase(ch)

For example, Character.toLowerCase('B') yields 'b', Character.toLowerCase('b') yields 'b', and Character.toLowerCase('1') yields '1'. Note that for ch a character variable, Character.toLowerCase(ch) is an *expression* that generates a new character without changing the contents of ch. If you really want to replace the contents of ch with a lower case version you write an assignment statement: ch = Character.toLowerCase(ch);.

The following piece of code shows how to read a single character from the keyboard and determine if it is a letter or not:

```
char ch = Console.readChar();
if (Character.isLetter(ch))
        System.out.print("That's a letter!");
else
        System.out.print("That's not a letter!");
```

If you need further understanding of the behaviour of the character operations, look them up in a text book.

Strings

Java provides the type String, whose values consist of all strings. A string is a sequence of characters, whether letters, digits, punctuation marks, etc. In Java, strings are delimited by double-quote marks, as in "This is a string", but the quote marks are not part of the string and do not get printed. Strings mostly occur as arguments of print() and println(). However, strings are values in their own right, and can be the subject of operations. The most common operation on strings is concatenation, denoted by +. For example, execution of

```
String s;
s = "Fee Fi";
s = s + "Fo Fum";
```

```
System.out.print(s);
```

causes `Fee FiFo Fum` to be displayed. The concatenation operation also supports concatenation of single characters; for example, "cat" + 's' yields "cats" (as does "cat" + "s").

It is important that you understand the distinction between writing variables surrounded by quotes, on the one hand, and without quotes on the other hand. The following program illustrates this; it outputs `j = 17`

```
class What {
   public static void main(String[] args) {
      int j = 17;
      System.out.print("j = " + j);
   }
}
```

Java supplies a large collection of string operations, some of the more important ones are listed below. Many of the operations rely on the fact that each character in a string has a position, starting at 0. For example, the index of 'D' in "Dublin" is 0, the index of 'u' is 1, and so on. In the following table of examples, we presume that variable s is initialised as follows:

```
String s = "Dublin";
```

Expression	*Value*	*Explanation*
s.length()	6	number of characters in string s
s.startsWith("Dub");	true	does s begin with "Dub"?
s.startsWith("dub");	false	does s begin with "dub"?
s.endsWith("in");	true	does s end with "in"?
s.indexOf("bli");	2	index of first "bli" in s?
s.indexOf("cat");	-1	index of first "cat" in s (-1 as no occurrence)
s.substring(2,5);	“bli”	that part of s indexed from 2 to (but excluding) 5
s.substring(2);	“blin”	the tail of s starting from index 2
s.charAt(2);	'b'	the character at index 2 in s
s.toUpperCase();	“DUBLIN”	the upper case equivalent of s
s.toLowerCase();	“dublin”	the lower case equivalent of s
“ hi there ”.trim()	“hi there”	“ hi there ” without leading or trailing blanks

The string preceding the dot in these operations can be any string expression, and not necessarily a string variable. For example, `(s.substring(2,5)).trim()` is a valid string expression. Note that these operations do not change the string – they generate a new string which can be printed or assigned to a variable. For example, if you want to change all the lower case letters in a string s to uppercase, write the assignment

```
s = s.toUpperCase();
```

Example

The following program illustrates some string operations. It reads a person's name in the form of a forename and a surname, possibly with spaces surrounding each word. It displays the name as upper case surname followed by the initial letter of the forename. An example of input/output is

```
Enter your name: Albert Einstein
EINSTEIN, A.
```

The program is

```
class Names {

    public static void main(String[] args) {

        // Read a name and remove any leading and trailing spaces
        System.out.print("Enter your name: ");
        String name = Console.readString();
        name = name.trim();

        // Locate the position of the first letter of the surname
        int i = name.indexOf(' '); // first space at position i
        while (name.charAt(i) == ' ') {  // passing over spaces
            i++;
        }

        // Extract the surname and convert to upper case
        String surname = (name.substring(i)).toUpperCase();

        // Extract the initial
        char initial = name.charAt(0); // strings are indexed from 0
```

```
            // Display output
            System.out.println(surname + ", " + initial + ".");
    }
}
```

Observe that the assignment to `surname` carries out the extraction of the surname, and then the conversion to upper case. Make sure you understand how this works – it could also be written as

```
String surname = name.substring(i);
surname = surname.toUpperCase();
```

(As an exercise, replace the two assignments to `name` with a single assignment which both reads the name and removes the leading and trailing spaces).

String comparisons

Never use the relational operators (==, !=, <, etc.) to compare strings. Instead of == use equals() (or equalsIgnoreCase() if the case of the strings arguments is not significant). For example, to compare string variables s and t for equality write one of

```
s.equals(t)
t.equals(s)
```

-- do not write `s==t` or `t==s`. For more general comparisons use compareTo(), as in

```
s.compareTo(t)
```

which yields a negative integer if s comes before t in the usual lexicographic (i.e. dictionary or alphabetic) ordering, a positive integer if t comes before s, and zero if s and t are equal. Do not write s<t or s<=t. If you are not familiar with this, look it up in your textbook.

Banana Skin: equals() and compareTo() are used to compare strings, but characters are compared using ==, < , <== etc.

Example

The following program illustrates string comparisons. It reads two names (in the same format as the preceding example), and displays that name which comes first in phonebook ordering. Phonebook ordering is alphabetically by surnames, with ties being resolved by alphabetic ordering of forenames . An example of i/o is:

```
Enter first name: Albert Einstein
Enter second name: Stephen Hawking
Albert Einstein
```

```
class LeastName {

        public static void main(String[] args) {
           String forename1, surname1, forename2, surname2;

           // Read first name and break into component parts
           System.out.print("Enter first name: ");
           forename1 = Console.readToken();
           surname1 = Console.readToken();

           // Read second name and break into component parts
           System.out.print("Enter second name: ");
           forename2 = Console.readToken();
           surname2 = Console.readToken();

           // Compare names alphabetically
           if (surname1.compareTo(surname2)<0 ||
              (surname1.equals(surname2) && forename1.compareTo(forename2)<0))
           {
              System.out.println(forename1 + " " + surname1);
           }
           else {
              System.out.println(forename2 + " " + surname2);
           }
        }
}
```