15 Conditional Statements

Read and study this section with care. It is fundamental to programming and contains new ideas and some complex syntax.

15.1 Introduction

Up to now we have been dealing with programs that read numbers, do fixed calculations in a pre-specified order and output results. One of the main powers of computing is conditional control over which statements (parts of the program) are executed, in which order, and how many times. This is generally called *Flow Control* and will be considered in two sections, the first dealing with *Conditional Statements* and the second dealing with *Loops*, which you will deal with *after* the next checkpoint.

15.2 The `boolean` Data Type

Fundamental to flow control is the `boolean` data type which can take on the two possible values:

`true` or `false`

This can be set by the "comparator" operators,

`>`	Greater Than
`<`	Less Than
`==`	Is equal to (Note: double == sign)
`>=`	Greater than or equal to
`<=`	Less than or equal to
`!=`	Not equal to

so for example we can write;

```
boolean ask;
int iValue = <some expression>;

ask = iValue < 5;
```

which will set `ask` to `true` or `false` depending on the value of `iValue`.

There comparator operators can be used to compare any basic data types, `ints` or `doubles` in this course.

15.3 Conditional Statements

In Java conditional execution is mainly accomplished by the `if` statement which in its simplest form is:

```
if (boolean) {
     <-- First line of optional code block -->;
```

```
           <-- Second line of optional code block -->;
           <-- .... -->;
           <-- nth line of optional code block -->;
      }
```

When the “`boolean`” is `true` then the *optional code block* is executed, else it is skipped over. Note the following points about the syntax,

1. There is **no** “`;`” after the `if ()` statement. *This is a very common source of programming bugs!*
2. The optional code block is enclosed in {} brackets that *must* match. The emacs editor will help here.
3. The `<-- text -->` line means “replace with valid Java statement”.
4. Each line of optional code ends with a “`;`”.

so for a simple example we have:

```
      double xValue;
      <-- code to set the value of xValue -->
      if (xValue > 5.0) {
         System.out.println("xValue is greater than 5.0’’);
      }
```

which will print out the message if `xValue` > 5.
A few points to note are:

1. The “indentation” is *not* part of the program, but laying the code out with “indented” `if()` will make it much easier to read. The `emacs` editor will do most of this for you if you insert a `<TAB>` before each line.
2. The “equal to” (==) operator and the “not equal to” (`!=`) operators should not be used with `double` variables since they compare every *bit* so are highly dependent on how the number is stored and are very sensitive to rounding errors. (See more on this later.)

More on Logical Statements

To form more complex comparison statements the comparator operators can be combined with the three *logical operators*

`||`	OR
&&	AND
!	NOT

Note: The logical AND and OR operators are **double**[1] characters.

These operators are evaluated after the comparison operators, but it is good programming practice to put in brackets to make the order of evaluation clear (to you). For example a condition of `xValue < 5` *OR* `xValue > 10` can be written as:

[1]There *are* single character operators `|` and & which are the “bitwise operators” will not be used in this course.

```
if((xValue < 5) || (xValue > 10)) {
    <-- First conditional statement  -->;
    <-- Second conditional statement -->;
               .....
}
```

These operators can be combined to form very complex statements.

15.4 Double Conditionals

The extended syntax of the `if()` statement is the very useful `if () {} else {}` construct:

```
if (boolean) {
    <--  First line of optional true code block -->;
    <--  Second line of optional true code block -->;
    <-- ....-->;
    <--  nth line of optional true code block -->;
} else {
    <--  First line of optional false code block -->;
    <--  Second line of optional false code block -->;
    <--  .... -->;
    <--  nth line of optional false code block -->;
}
```

If the `boolean` value is `true` then the first code block is executed; *else* the second code block is executed.

The use of the `if() {} else {}` gives good “block” structured code that is easy to read, and thus is *more* likely to be correct.

15.5 Multiple Conditional

The full syntax of the `if()` includes the `else if()` giving the rather complex structure of:

```
if (boolean_1) {
  <-- optional code if boolean_1 is  "true" -->;
} else if (boolean_2) {
  <-- optional code if boolean_2 is "true"  -->;
} else if ...
  .
  .
} else if (boolean_n) {
  <-- optional code if boolean_n is "true"  -->;
} else  {
  <-- optional code if all booleans are "false"  -->
}
```

which allows a whole "chain" of logical statements to be "tried-out" with the correct code executed. The logic of such `else if` "chains" is very difficult to get right and even more difficult to Debug. If you do need to use this structure then you should "draw" the structure out on paper first before you try and code it.

15.6 The `System.exit()` Method

The `System.exit();` method basically "stops" execution of the program exactly as if the program had completed. Up to now you have been using this at the end of your program but it can also be used to conditionally exit the program.

The syntax is simple being

```
System.exit(int status);
```

where `status` is an integer value that is returned to the operating system.

This is useful for complex programs than interact with the actual system.

The typical use of `exit()` is to stop your program if an error has occurred, for example:

```
double xValue, yValue;
<-- code to calculate value of xValue -->;
if ( xValue < 0){
  System.out.println("Value of xValue < 0. Fatal Error");
  // An exit value of 1 usually indicates an error to the system
  System.exit(1);
}
else {
  yValue = Math.sqrt(xValue};
}
```

which will stop the program (with a sensible message) if `xValue` is negative *before* it tries to take the square root of it.

15.7 The `switch` Construct

`switch` is a complex and very useful dispatch construct that can be used to replace complex `if else()` chains. You are *strongly* advised to learn about this, see textbooks, but *after* you have mastered the `if()` structure.

Examples

The following on-line source examples are available:

- Simple square root calcation trapping negative numbers SquareRoot.java
- Determining if a double is $+/-/0.0$ with a `if, else if, else` chain in NumberSign.java
- More complex mark processor program with multiple conditionals PassFail.java

What Next?

You have now completed sufficient Java to attempt *Checkpoint 4*.