CS1Ah Lecture Note 6

Control structures in Java

We have mentioned that Java programs contain *classes* and that these classes contain *methods* which contain *statements* that are executed by the computer. Now we look at ways of putting statements together using *control structures* to organise the execution of the program. A control structure might cause a statement to be executed once, several times, or not at all. Control structures make up some of the statements of the Java language. Statements are said to be *sequentially composed* when they are written one after the other. Sequentially composed statements can be grouped together into a *block* by using the left brace and right brace symbols (“`{`” and “`}`”) to bracket them.

6.1 Assignments

An *assignment* statement in Java has two parts. It has a *variable* on the left-hand side and an *expression* on the right. The expression is first *evaluated* to yield a value. That value is then used as the new value of the variable. Here are some examples.

Statement	**Effect**	`x` before	`x` after
`x = 0;`	sets `x` to zero	5	0
`x = -x;`	negates `x`	5	–5
`x = x + 2;`	adds two to `x`	5	7

The equals sign is an *assignment operator* in Java, but it is not the only one. The operators `*=`, `/=`, `%=`, `+=` and `-=` are also assignment operators, and there are still others. These operators combine the use of an operator and the execution of an assignment into a single statement. These assignment operators provide a shorthand way of expressing assignment statements where the expression is used to modify the value of a variable by making use of its previous value in order to calculate the new value. Here are some examples of these kinds of assignment statements.

Statement	**Equivalent**	**Effect**	`x` before	`x` after
`x *= 2;`	`x = x * 2;`	multiplies `x` by two	5	10
`x /= 2;`	`x = x / 2;`	divides `x` by two	5	2
`x %= 2;`	`x = x % 2;`	sets `x` to `x` remainder two	5	1
`x += 2;`	`x = x + 2;`	adds two to `x`	5	7
`x -= 2;`	`x = x - 2;`	subtracts two from `x`	5	3

Other kinds of updates occur so frequently that the Java language provides shorthand versions of them. The operations of adding 1 to a variable and subtracting 1 from a variable are abbreviated as shown below.

Statement	**Equivalent**	**Effect**	x before	x after
`x++;`	`x = x + 1;`	add one to x	5	6
`x--;`	`x = x - 1;`	subtract one from x	5	4

6.2 Conditional statements

A conditional statement allows a choice from a selection of statements. It first evaluates an expression to decide between the possibilities. If there are only two then a *boolean-valued* expression (also called a *logical* expression) will be enough to allow us to choose. These can be formed using the == operator (equal to) or the != operator (not equal to). When comparing numeric values we could use the relational operators < (less than), > (greater than), <= (less than or equal to) or >= (greater than or equal to).

A conditional statement uses the keywords **if** and **else** to mark the beginning of the conditional statement and to separate the two sub-statements. The first sub-statement is to be executed if the expression in the condition evaluates to true. The second sub-statement is to be executed if the expression in the condition evaluates to false. These two sub-statements are referred to as the *then-statement* and the *else-statement* respectively.

It is possible for a conditional statement not to have an else-clause. Java has both an if-then statement and an if-then-else statement. We will consider some examples.

Statement	**Effect**

`if (x == 0)`
`    System.out.println("zero");`
	Prints "zero" if x has the value 0.

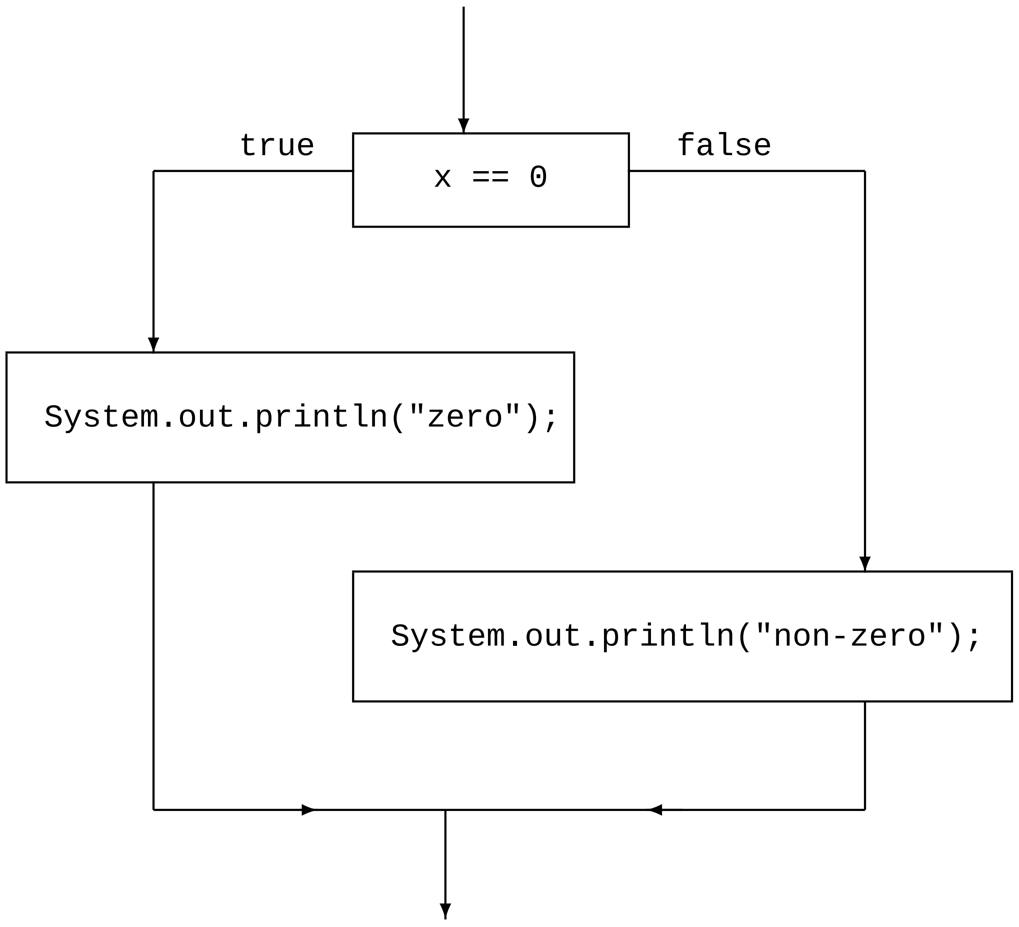
`if (x == 0)`
`    System.out.println("zero");`
`if (x != 0)`
`    System.out.println("non-zero");`
	Prints "zero" if x has the value 0, prints "non-zero" otherwise.

`if (x == 0)`
`    System.out.println("zero");`
`else`
`    System.out.println("non-zero");`
	Prints "zero" if x has the value 0, prints "non-zero" otherwise

`if (x == 0)`
`    System.out.println("zero");`
`else`
`    if (x > 0)`
`        System.out.println("positive");`
`    else`
`        System.out.println("negative");`
	Prints "zero" if x has the value 0, prints "positive" if x is positive, prints "negative" if x is negative.

`if (x != 0)`
`    if (x > 0)`
`        System.out.println("positive");`
`    else`
`        System.out.println("negative");`
	Prints "positive" if x is positive, and non-zero. Prints "negative" if x is negative.

We can picture a conditional statement graphically. We have a condition box with two exit paths, one labelled `true` and the other labelled `false`. The path labelled `true` leads to the "then"-statement. The path labelled `false` leads to the "else"-statement. After these two statements the paths meet up so that control can pass to the next statement in the program.



All of the conditional statements which we have seen have just a single statement as the then-statement or the else-statement. When we need to perform two actions in some case then we need to bracket them together with left and right braces. Without these the effect is quite different.

Statement	**Effect**

`if (x < 0) {`
`    System.out.println("negative");`
`    x *= -1;`
`}`
	Changes the sign of `x` and prints `negative` if `x` is negative.

`if (x < 0)`
`    System.out.println("negative");`
`x *= -1;`
	Prints `negative` if `x` is negative. Changes the sign of `x` whether it was negative or not.

When formatting computer programs we will normally use blank-space indentation to convey hints to the reader about statement structure. However, as far as the compiler for the Java language is concerned, one blank space is as good as ten so the different effect of the two statements is achieved by the use of the left and right braces, and not by the blank-space indentation at the start of the line.

6.2.1 Reserved words

The words “if” and “else” are *reserved words* in Java but the word “then” is not. A reserved word is used to mark out a particular kind of statement and cannot be used as an program identifer (this is the sense in which it is reserved). Throughout these notes we will format reserved words in bold typewriter font.

6.3 The switch statement

A conditional statement contains a boolean-valued expression. Sometimes the expression which we need to examine is an integer or a character. In this circumstance we can use a switch statement. The switch statement introduces four new keywords, **`switch`**, **`case`**, **`default`** and **`break`**. The following example is very similar to the second and third examples of conditional statements which we saw.

Statement	**Effect**
`switch` `(x) {`	
`case` `0:    System.out.println ("zero");`	Prints “`zero`” if `x` has the value 0, prints “`non-zero`” otherwise.
`              `**`break`**`;`	
`default``: System.out.println ("non-zero");`	
`              `**`break`**`;`	
`}`	

Without the break statements the flow of control *falls through* the statement so that the first matching statement is executed and then all of the statements which follow it. Since this is rarely useful, a switch statement almost always has a break statement at the end of each case.

In the example shown above we have only one case treated specially in the switch. We could have more. The if-then-else statement allows to choose between two possible sub-statements but the switch allows us to choose between any number of them. The other cases also appear in the body of the switch statement with the default case coming at the end.

6.3.1 Contrasting “switch” with “if-then-else”

The switch and conditional statements perform related tasks, but they are not interchangeable. One reason for this is that Java does not allow the programmer to write boolean literals as expressions on the limbs of the switch. This means that although we can think of a conditional statement as being like a switch on the value of a boolean expression, we cannot express this in Java.

Supporting this separation between “switch” and “if-then-else” is the fact that Java treats boolean values as different from integer values. A boolean variable in Java can hold only the value `true` or the value `false`. An integer variable can hold negative or positive integers.

6.4 Iteration

The statements which we have seen so far allow *selective* execution of statements. To express *repetitive* execution of statements we use another construct, Java's **`while`** statement. A while statement is often called a *loop*. A while loop contains a boolean-valued expression which is known as its *test* (or sometimes its *condition* or sometimes its *guard*). It also contains a statement which is the *body* of the loop.

The first thing to happen is that the boolean expression is evaluated. If it evaluates to `false` then the body of the loop is not executed at all and the flow of control can pass to the next statement in the program. If it evaluates to `true` then the loop body is executed, and then the test is redone to see if the loop body is to be executed again (this is the "looping" part), and this continues in this way until the test eventually evaluates to `false` and the flow of control can then finally pass on to the next statement in the program.

Here is a simple program fragment which prints out messages while decreasing the value of `x`. After it has finished it prints a message to report this.

```
while (x > 0) {
   System.out.println("decreasing");
   x--;
}
System.out.println ("finished");
```

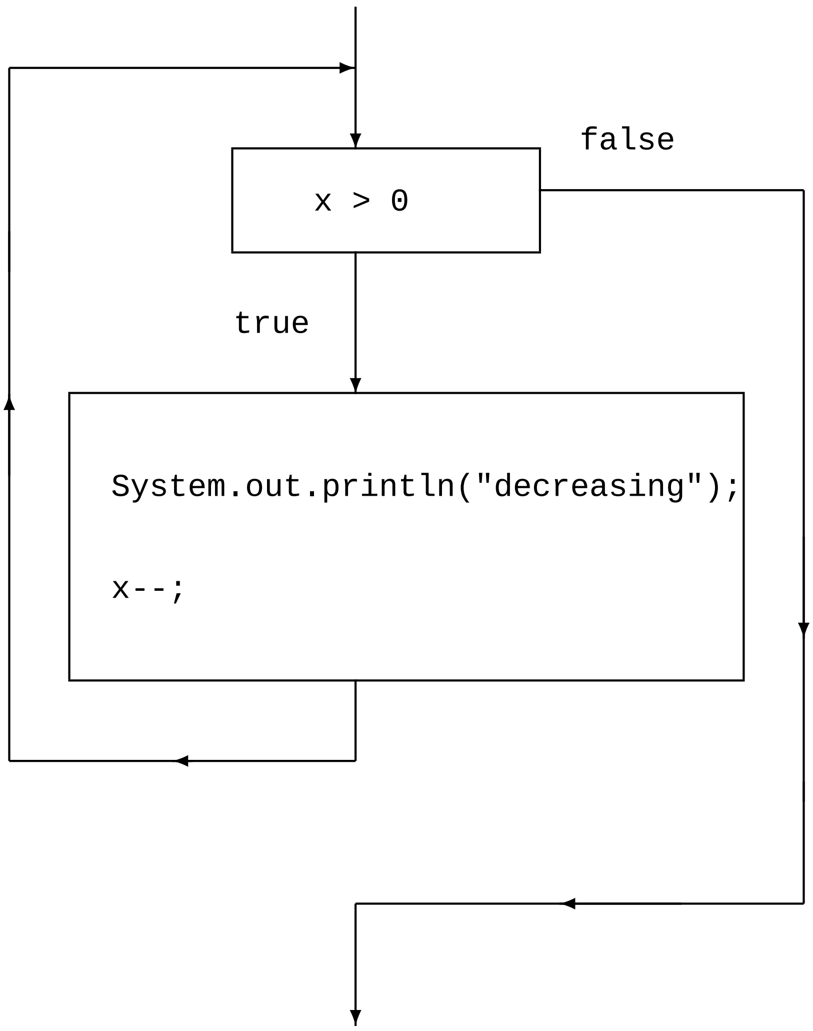
If we executed such a code fragment when the variable `x` held the value 3 then we would see the following results:

```
decreasing
decreasing
decreasing
finished
```

The two statements which come between the open brace and the close brace are the body of the loop. The body is repeatedly executed while the loop condition (`x > 0`) is true.

There is a possibility that the loop body will never be executed at all. If the value of `x` was zero or less upon reaching the **`while`** statement then the loop body will not be executed. In general with a **`while`** loop the body is executed zero or more times.

As with the conditional statement, a while loop can be pictured in a diagram. Similarly to the conditional statement again, a while loop has a boolean-valued condition at the top. The result of this expression determines which statement will be executed next. The picture is slightly more complicated because of the need to loop back to repeat the test after the loop body has been completed.



6.4.1 An example: computing prime factors

A more intricate use of the while loop is shown below. The program fragment below prints the *prime factors* of `x`. The prime factors of a number are those primes which can be multiplied together to give the number. For example, 3 and 5 are prime numbers and 3, 3, 3 and 5 are the prime factors of 135.

```
    int d = 2;                      // set d to 2
    while (x != 1) {                // stop when we reach 1
        while (x % d == 0) {        // while d divides x
            System.out.println(d);  //   report d
            x /= d;                 //   divide by d
        }                           //
        d++;                        // move on
    }
```

Compared to the program fragments which we have seen so far in these lecture notes, this is a very complicated example. We have a **`while`** loop with a **`while`** loop inside it. We sometimes describe this as a *nested* loop: the inner loop nested in the outer one.

Note by Stephen Gilmore.
David Aspinall, 2002/10/24 11:54:41.