Control structures: switch

As explained previously, one can use the `if` control structure to conditionally perform a computation. If there are many different conditions to handle, the program quickly becomes complex.

For example, consider the task of writing a function which, for a given month number returns the number of days in this month, or -1 if the month number is invalid. For this we can use the following code:

```
int daysInMonth(int month)
{
   int result;

   if (month == 1)
      result = 31;
   else if (month == 2)
      result = 28;
   else if (month == 3)
      result = 31;
   else if (month == 4)
      result = 30;
   else if (month == 5)
      result = 31;
   else if (month == 6)
      result = 30;
   else if (month == 7)
      result = 31;
   else if (month == 8)
      result = 31;
   else if (month == 9)
      result = 30;
   else if (month == 10)
      result = 31;
   else if (month == 11)
      result = 30;
   else if (month == 12)
      result = 31;
   else
      result = -1;

   return result;
}
```

We can try to "compress" this function using compound tests, as follows:

```
int daysInMonth(int month)
{
   int result;

   if (month == 1 || month == 3 || month == 5
     || month == 7 || month == 8 || month == 10
     || month == 12)
      result = 31;
   else if (month == 4 || month == 6 || month == 9
```

```
            || month == 11)
        result = 30;
    else if (month == 2)
        result = 28;
    else
        result = -1;
}
```

Yet although this code is shorter, it is not particularly more readable.

To simplify this situation, different languages offer different solutions; however they are all based on the same general idea, called a *multi-way branch*:

- at the top, you write only once the variable containing the value to test;
- on subsequent lines, you write the different values that are recognized, and next to this what needs to be done in each case.

The particular multi-way branch construct offered by Java is inherited from the C language; it is called the *switch statement*. It is defined as follows:

Switch statement:

Syntax:

```
switch ( <expression> ) {
    [
        [ case <constant value> : ]*
            [ <statement> ]*
    ]*
        [ default : ]?
            [ <statement> ]*
}
```

(the "`switch`" keyword, followed by an expression between parentheses, followed by an opening brace, followed by zero or more occurrences of zero or more "`case`" and zero or more statements, followed by an optional `default:` followed by zero or more statements, followed by a closing brace).

Semantics: the expresssion at the top is evaluated. Then the flow of control is transferred to the *case label* that matches the value of the expression; or to the `default` label if no label matches the expression.

Note
A "case label" is a construct of the form "`case` <value>".

For example:

```
int daysInMonth(int month)
{
    switch (month)
    {
       case 1: case 3: case 5:
       case 7: case 8: case 10:
       case 12:
          result = 31;
          break;
       case 4: case 6: case 9:
       case 11:
          result = 30;
          break;
       case 2:
          result = 28;
          break;
       default:
          result = -1;
    }
}
```

This example uses one `switch` with 12 case labels and a `default` label.

Note how each case uses the "`break`" statement. To understand why, read the "semantics" definition above: control is transferred to the label that matches, but then *the flow of control continues* within the `switch` statement, *even into the next cases*.

For example with the following code:

```
int n = 3;
switch (n) {
   case 3:
      out.println("three");
   case 5:
      out.println("four");
}
```

this code would print "three" but also "four": once the flow of control jumps to the case label "`case 3`", it continues uninterrupted until the end of the `switch` statement. In order to ensure only the code for "`case 3`" runs, you need to add `break` as follows:

```
int n = 3;
switch (n) {
   case 3:
      out.println("three");
      break;
   case 5:
      out.println("four");
}
```

Limitations of `switch` in Java

For historical reasons, `switch` in Java can only be used with very few data types in the condition. The full list of types that can be used with `switch` is:

- `int`;
- `String`;

- `byte`, `short`, `char`;
- enumerated types (we will cover this later);
- a few special object classes that wrap around primitive types: `Character`, `Byte`, `Short`, and `Integer`.

In particular, `switch` *cannot match approximate real values* of type `float` or `double`. If you must write a complex condition using values of these types, you are limited to using `if`.

> **Note**
> This limitation is a restriction historically inherited from C. Many other programming languages have more comprehensive multi-way branches that can match on many more types. **When learning a new programming language, pay special attention to which multi-way branches it supports.** This is important as it will greatly enhance your productivity when writing your own code.

Important concepts

- *multi-way branching*;
- the *switch statement*;
- *case labels*;
- why `break` is necessary;
- which data types are compatible with `switch`.

Further reading

- Absolute Java, chapter 3 (pp. 107-112)
- Introduction to Programming, section 3.6 (pp. 104-109)

Copyright and licensing