6 Statements

The statements of C# do not differ very much from those of other programming languages. In addition to assignments and method calls there are various sorts of selections and loops, as well as branch statements and statements for *exception handling*. We will explain statements using examples. Details of the syntax can be found in Appendix A.3.

6.1 Empty Statement

Every non-structured statement in C# must be terminated by a semicolon. A semicolon by itself is an empty statement that indicates no action and is used, for example, to express an empty loop body.

6.2 Assignment

An assignment evaluates an expression and assigns its value to a variable. The assignment itself is also an expression, so multiple assignments are possible:

```
x = 3 * y + 1;      // x becomes 3 * y + 1
a = b = 0;          // multiple assignment: a and b both get the value 0
```

The type of the expression must be *assignment compatible* with the type of the variable. This means that the two types must be the same or the type of the variable must include the type of the expression as set out in Fig. 3.2 (for example, short values may be assigned to int variables). Assignment compatibility also applies if the type of the expression is a subclass of the variable type (see Chapter 9).

Assignments can be combined with various binary operators. For example:

```
x += y;
```

is a short form for

```
x = x + y
```

These short forms are useful in the case where x is a composite name (e.g. a[i].f). They reduce the typing effort and make it easier for the compiler to optimize an

assignment. Combined assignments are possible with the operators +=, -=, *=, /=, %=, <<=, >>=, &=, |= and ^=.

6.3 Method Call

A method is called through its name and a parameter list. The details of method calls are covered in Section 8.3. Here are some examples of calls with methods of class String:

```
string s = "a,b,c";
string[] parts = s.Split(',');          // calls the non-static method s.Split
s = String.Join(" + ", parts);          // calls the static method String.Join
char[] arr = new char[10];
s.CopyTo(0, arr, 0, s.Length);
```

As Section 8.3 will show, methods can be *static* or *non-static*. A *non-static* method (for example Split) is applied to a specific *object* (for example s). s.Split(',') returns the substrings of s which are delimited by occurrences of ',' (here "a", "b" and "c").

A *static* method (e.g. Join) is not applied to an object but to a *class* (e.g. String). It is comparable with an ordinary function in C. For example, String.Join(" + ", parts) concatenates the strings in the array parts together with " + ". The result here is "a + b + c".

Both Split and Join are *function methods*. They are called as operands in expressions and return a value. However, there are also methods that do *not* deliver a function result. They have the method type void and are called as stand-alone statements. For example, s.CopyTo(from, arr, to, len) copies len characters of the string s starting at the position from into the array arr beginning at the position to.

6.4 if Statement

An if statement has the form

if (*BooleanExpression***)** *Statement* **else** *Statement*

If the Boolean expression is true, the then branch (the first statement) is executed, otherwise the else branch (the second statement). The else branch may be omitted. If the then branch or the else branch consists of several statements then they must be written as a *statement block* in braces. Here are some examples of if statements:

```
if (x > max) max = x;                   // without else branch
if (x > y) max = x; else max = y;       // with else branch
if ('0' <= ch && ch <= '9')
   val = ch - '0';
else if ('A' <= ch && ch <= 'F')        // nested if
```

```
    val = 10 + ch - 'A';
else {                                  // else branch consists of a statement sequence
    val = 0;
    Console.WriteLine("invalid character: " + ch);
}
```

In contrast to C and C++, the if expression must be of type bool. In particular, it is not permissible to interpret the value 0 or null as false.

6.5 switch Statement

The switch statement is a *multi-way selection*. It consists of an expression and several statement sequences each prefixed by case labels. The switch statement branches to the case label that corresponds to the value of the expression. If there is no matching case label it jumps to the default label and if there is none then to the end of the switch statement. Here is an example:

```
switch (country) {
    case "England": case "USA":
        language: = "English";
        break;
    case "Germany": case "Austria": case "Switzerland":
        language = "German";
        break;
    case null:
        Console.WriteLine("no country specified");
        break;
    default:
        Console.WriteLine("don't know language of " + country);
        break;
}
```

The expression in the head of the switch statement must be numeric, an enumeration or of the type char or string. The case labels must be disjoint constant values whose type must be assignment compatible with the type of the expression.

In contrast to most other languages, C# allows the switch expression to be of type string (including case labels with the value null). In this case the switch statement is treated by the compiler as a series of nested if statements, whereas in the other cases it is implemented as a direct jump to the matching case label.

Each statement sequence between the case labels *must* end with a statement to break out. The most common of these is the *break statement* which jumps to the end of the switch statement. Other statements allowed are return, goto and throw. We discuss these later. In contrast to most other languages, C# does not allow a program to fall through a (non-empty) case branch to the next. If this is what we want we must implement it with a *goto* statement.

6.6 while Statement

The while statement is the most common form of loop. It consists of a Boolean expression and a loop body which is repeatedly executed as long as the expression remains true. The expression is tested *before* each execution of the loop body. If the loop body consists of more than one statement then it must be written as a statement block in braces.

```
while (x > y) x = x / 2;        // loop body consists of a single statement
while (i < n) {                 // loop body consists of a statement sequence
    sum += i;
    i++;
}
```

6.7 do-while Statement

The do-while statement differs from the while statement only in that the Boolean expression is tested *after* the loop body. This means that the loop body is executed at least once.

```
do i = 10 * i while (i < n);    // loop body consists of a single statement
do {                            // loop body consists of a statement sequence
    sum += a[i];
    i--;
} while (i >= 0);
```

6.8 for Statement

The for statement is the most flexible but also the most complicated form of loop. It has the form:

for (*Initialization***;** *Condition***;** *Increment***)** *Statement*

Before the first run through the loop the initialization is executed. This usually assigns a value to a loop variable. Before each execution of the loop the condition is tested and at the end of each iteration the incrementation is carried out. The loop body is executed as long as the condition is true. The statement

```
for (int i = 0; i < n; i++)
    sum += i;
```

can be regarded as a short form of the while loop

```
int i = 0;
while (i < n) {
    sum += i;
    i++;
}
```

It is treated by the compiler in exactly the same way as the while loop (except that i is local to the for loop). Both the initialization and the incrementation can comprise more than one statement. These are then separated by a comma instead of being terminated by a semicolon. For example:

```
for (int i = 0, j = n-1; i < n; i++, j--)
   sum += a[i] + b[j];
```

6.9 foreach Statement

The foreach statement offers a convenient way of iterating through an array, a string or some other collection of elements that implements the ICollection interface (see Section 18.2). It has the form:

foreach (*ElementVarDecl* **in** *Collection***)** *Statement*

Here are two examples of foreach statements:

```
int[] a = {3, 17, 4, 8, 2, 29};
sum = 0;
foreach (int x in a) sum += x;

string s = "Hello";
foreach (char ch in s) Console.WriteLine(ch);
```

The first loop sums the elements of the array a. The second prints all the characters of the string s. The following example is also interesting:

```
Queue q = new Queue();
q.Enqueue("John"); q.Enqueue("Alice"); ...
foreach (string s in q) Console.WriteLine(s);
```

The elements of a queue are stored as data of type object. The compiler knows, however, that the loop variable s is of type string. Therefore it generates a checked type cast from object to string when it retrieves the elements from q. The variable of a foreach loop can only be accessed for read operations.

6.10 break and continue Statements

We have already seen the break statement in the context of the switch statement. However, it can also be used to terminate the execution of a loop:

```
for (;;) {
   int x = stream.ReadByte();
   if (x < 0) break;
   sum += x;
}
```

This example shows the use of a for statement as an *endless loop*. If the initialization part, the condition and the incrementation part are omitted then the loop cycles forever—or until, as in this case, it is terminated by executing a break statement. The break statement can also be used in while, do-while and foreach loops. With nested loops, however, it only leaves the innermost loop. In order to break right out of a nested loop structure a goto statement must be used.

The continue statement (written as continue;) may likewise be used in any kind of loop. It indicates that the rest of the loop body should be skipped, any incrementation part should be carried out (only in for loops) and the continuation condition should be tested again before the next run through the loop begins.

6.11 goto Statement

The goto statement jumps to a *label* that is written in front of another statement. The label consists of a name followed by a colon. For example, the do-while statement could also be coded by means of a goto and a label:

```
top:                              //  do {
    sum += i;                     //      sum += i;
    i++;                          //      i++;
    if (i <= n) goto top;         //  } while (i <= n);
```

This is however not recommended because it obscures the structure of the program. A sensible use of a goto statement is, for example, to break out of an inner loop, because this cannot be done with a break statement.

Because the unrestricted use of the goto statement can have a negative impact on the quality of a program there are certain restrictions pertaining to jumps: although it is possible to break out of a block (if there is an error, for example), it is not possible to break into a block. It is also illegal to break out of a finally block (see Chapter 12). Breaking out of a try statement (see Chapter 12) means that the finally block is executed first.

Goto statements can also be used within a switch statement to jump to a case label. This is a sensible use of a goto because it allows the efficient implementation of so-called *finite-state machines*. A finite-state machine consists of states with transitions between them, which can be triggered when defined symbols are read. Fig. 6.1 shows a finite-state machine, with circles representing the states and arrows representing the transitions.

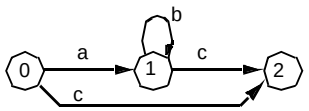

Fig. 6.1 *Finite-state machine*

This state machine can be implemented in the following way by using goto statements:

```
int state = 0;                  // starts in state 0
int ch = Console.Read();        // reads first input symbol
switch (state) {
   case 0:  if (ch == 'a') { ch = Console.Read(); goto case 1; }
            else if (ch == 'c') goto case 2;
            else goto default;
   case 1:  if (ch == 'b') { ch = Console.Read(); goto case 1; }
            else if (ch == 'c') goto case 2;
            else goto default;
   case 2:  Console.WriteLine("input valid!");
            break;
   default: Console.WriteLine("illegal character: " + (char) ch);
            break;
}
```

It would be even shorter in this case to discard the switch statement and jump directly to labels named state0, state1, state2 and illegal.

6.12 return Statement

The return statement allows the (early) termination of methods. It has two forms. *Void* methods *can* be terminated by a return statement without an argument. For example:

```
void P(int x) {
   if (x < 0) return;
   ...
}
```

Of course, the method ends after its last statement even if return is not executed. *Function* methods *must* be dynamically terminated by a return statement that has the return value as its argument. For example:

```
int Max(int a, int b) {
   if (a > b) return a; else return b;
}
```

The type of the return value must be assignment compatible with the return type declared in the method heading. A function method must not reach its end without executing a return statement, because it always has to return a value.

The Main method of a program may also be declared as a function. Its return value will be interpreted as an error code that is stored in a system variable (the errorlevel variable in Windows).

```
class Test {
   static int Main() {
      ...
      if (...) return -1;
      ...
   }
}
```

6.13 Exercises

1. **if statement**. Use an if statement to determine the highest of three numbers x, y and z.

2. **switch statement**. Convert the switch statement in Section 6.5 into an if statement with else branches.

3. **switch statement**. Write a switch statement that calculates the number of days in a given month (ignore leap years). Implement one version in which the month is given as an integer, and another in which it is given as a string. Use ildasm to compare the generated code (see Appendix A.2).

4. **Loop transformations**. Convert the following while loop into a corresponding do-while loop and a corresponding for loop:

   ```
   int i = ...;
   while (i > 0) { Foo(i); i--; }
   ```

5. **Search loop**. Assume we have an array of numbers called tab. Write a search loop that returns the index of the value val in the tab array, or 0, if val does not occur in tab.