2 The Language C#

C# (pronounced: *see sharp*) is Microsoft's new programming language for the .NET platform. Although .NET can also be programmed in many other languages (for example, Visual Basic .NET, or C++) C# is Microsoft's preferred language; it supports .NET best and is best supported by .NET.

C# is not a revolutionary new language. It is more a combination of Java, C++ and Visual Basic. The aim has been to adopt the best features of each of these languages while avoiding their more complex features. C# has been carefully developed by a small team lead by *Anders Hejlsberg*. Hejlsberg is an experienced language expert. At Borland he was the chief designer of Delphi. He is known to design his languages with the needs of practitioners in mind.

In this chapter we assume that the reader already has some programming experience, preferably in Java or C++. While we are explaining the concepts of C# we will also compare them with Java and C++.

2.1 Overview

Similarities to Java

At first sight C# programs look much like Java programs. Any Java programmer should be able to read them. As well as having almost identical syntax the following concepts have been carried across from Java:

- *Object-orientation*. Like Java, C# is an object-oriented language with single inheritance. Classes can inherit from just one base class but can implement several interfaces.
- *Type safety.* C# is a type-safe language. Programming errors that arise from incompatible types in statements and expressions are detected by the compiler. There is no arbitrary pointer arithmetic and no unchecked type casts as in C++. At run time there are checks to ensure that array indices lie in the appropriate range, that objects are not referenced via uninitialized pointers and that a type cast leads to a well-defined result.

- *Garbage collection.* Dynamically allocated objects are not released by the programmer, but are automatically disposed of by a garbage collector as soon as they are no longer referenced. This eliminates many awkward errors that can occur, for example, in C++ programs.
- *Namespaces.* What Java calls packages C# calls namespaces. A namespace is a collection of type declarations. It allows the same names for classes, structures or interfaces to be used in different contexts.
- *Threads.* C# supports lightweight parallel processes in the form of threads. As in Java, there are mechanisms for synchronization and communication between threads.
- *Reflection.* As in Java, type information about a program can be accessed at run time, classes can be loaded dynamically, and it is even possible to compose executable programs at run time.
- *Libraries.* Many types in the C# library resemble those in the Java library. There are familiar classes such as Object, String, Hashtable or Stream, often even with the same methods as in Java.

Various features are also taken from C++, for example operator overloading, pointer arithmetic in system-level classes (which must be marked as unsafe) as well as some syntactical details, for example in connection with inheritance. From Visual Basic comes the foreach loop, for example.

Differences from Java

Beside these similarities however, C# has several characteristics that go beyond Java. Most of them also apply to the other .NET languages,:

- *Reference parameters.* Parameters can be passed not only *by value* but also *by reference*. Because of this, one can use not only input parameters but also output and transient parameters.
- *Objects on the stack.* Whereas in Java all objects are kept on the heap, in C# an object can also be stored in the method-call stack. Such objects are lightweight, that is, they make no demands of the garbage collector.
- *Block matrices.* For numerical applications the Java storage model for multi-dimensional arrays is too inefficient. C# allows the programmer to choose whether to have a matrix laid out as in Java (that is, as an array of arrays) or as a compact block matrix, as in C, Fortran or Pascal.
- *Enumerations.* As in Pascal or C, there are enumeration types whose values are denoted by names.
- *Goto statement.* The much-maligned goto statement has been reintroduced in C#, but with restrictions that make it scarcely possible to misuse it.

- *Uniform type system.* In C# all types are derived from the type object. In contrast to Java, numbers or character values can also be stored in object variables. The C# mechanism for this is called *boxing*.
- *Versioning.* Classes are given a version number during compilation. Thus a class can be available in several versions at the same time. Each application uses the version of a class with which it was compiled and tested.

Finally there are many features of C# that are convenient to use, although they don't really increase the power of the language. They can be viewed as "*syntactic sugar*", because they allow one to do things that are also possible in other languages, but the way of doing them in C# is simpler and more elegant. Among them are the following:

- *Properties and events.* These features facilitate component technology. Properties are special fields of an object. When they are accessed the system automatically calls getter or setter methods. Events can be declared and triggered by components and handled by other components.
- *Indexers.* An index operation like that for arrays can be declared for custom collections via getter and setter methods.
- *Delegates.* Delegates are essentially the same as *procedure variables* in Pascal or *function pointers* in C. However, they are more powerful. For example, several methods can be stored in a delegate variable at the same time.
- *foreach loop.* This loop statement can be used for iterating through arrays, lists or sets in a convenient manner.
- *Boxing/unboxing.* Values such as numbers or characters can be assigned to variables of type object. To do this they are automatically wrapped into an auxiliary object (*boxing*). On assignment to a number or a character variable they are automatically unwrapped again (*unboxing*). This feature allows the construction of generic container types.
- *Attributes.* The programmer can attach metadata to classes, methods or fields. This information can be accessed at run time by means of reflection. .NET uses this mechanism, for example, for serializing data structures.

Hello World

Now it is time for a first example. The well known Hello World program looks like this in C#:

```
using System;
class Hello {
    public static void Main() {
        Console.WriteLine("Hello World");
    }
}
```

It consists of a class Hello and a method Main (Note that upper and lower case letters are considered to be different in C#). Each program has exactly one Main method, which is called when the program is started. Console.WriteLine("...") is an output statement, where WriteLine is a method of the class Console that comes from the namespace System. In order to make Console known, System must be imported in the first line by writing using System;.

The simplest development environment for .NET is Microsoft's *Software Development Kit* (SDK). It is command-line oriented. In addition to a compiler (csc) it provides several other software tools (such as al or ildasm). These are described in Chapter 8. C# programs are stored in files with the suffix .cs. If we store our Hello World program in a file Hello.cs it can be compiled by typing

```
csc Hello.cs
```

and executed by typing

```
Hello
```

The output appears in the console window.

Under .NET the file name does not have to match the class name, although this is recommended for readability. A file can consist of several classes. In this case it should be named after the principal class it defines.

Structuring Programs

The source text of a C# program can be spread across several files. Each file can consist of one or more namespaces. Each namespace can contain one or more classes or other types. Figure 2.1 shows this structure.

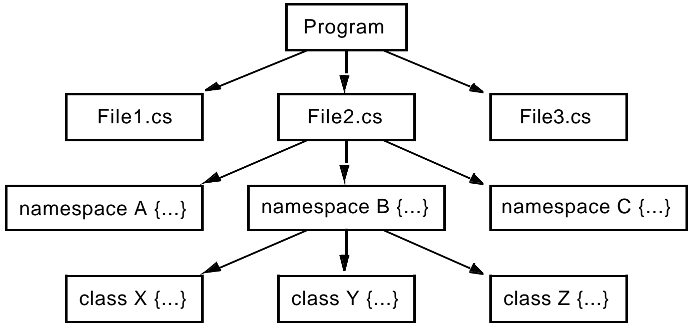

Figure 2.1 *Structure of programs*

Our Hello World program consists of a single file and a single class. No namespace is specified. Thus the class Hello belongs to an anonymous default

namespace created for us by .NET. Namespaces are dealt with in Section 2.5 and 2.13; classes in Section 2.8.

Multi-file Programs

If a program consists of several files, they can be compiled either together or separately. In the first case a single executable is generated. In the second case an executable and one or more DLL (*Dynamic Link Library*) files are created.

Let's consider a class Counter in a file Counter.cs used by a class Prog in a file Prog.cs:

```
public class Counter {        // in file Counter.cs
    int val = 0;
    public void Add(int x) { val = val + x; }
    public int Val() { return val; }
}

using System;                 // in file Prog.cs
public class Prog {
    static void Main() {
        Counter c = new Counter();
        c.Add(3); c.Add(5);
        Console.WriteLine("val = " + c.Val());
    }
}
```

We can compile these two files with a single command

```
csc prog.cs Counter.cs
```

which creates the executable file Prog.exe that contains both classes. Alternatively we can make a library (DLL) from Counter by writing:

```
csc /target:library Counter.cs
```

In this case the compiler creates a file Counter.dll, and we have to specify it as follows when compiling Prog.cs

```
csc /reference:Counter.dll Prog.cs
```

This compilation also creates a file Prog.exe, but this time it contains only the class Prog. The class Counter remains in the file Counter.dll and is dynamically loaded when Prog is invoked. The compilation command and its options are detailed in Section 8.2.

2.2 Symbols

C# programs are made up of names, keywords, numbers, characters, strings, operators and comments.

Names. A name consists of letters, digits and the character "_". The first character must be a letter or a "_". Upper and lower case letters are considered as distinct (for example, red is not the same as Red). Because C# uses Unicode characters [UniC], names can also contain Greek, Arabic or Chinese symbols. However, with Western keyboards these symbols must be input using numeric codes. For example, the code \u03C0 denotes π and the name b\u0061ck means back.

Keywords. C# has 76 keywords; Java has only 47. This already suggests that C# is more complex than Java. Keywords are reserved; that means they cannot be used as names.

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string
struct	switch	this	throw	true	try
typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	while		

Naming conventions. The following rules should be followed when choosing names and deciding whether to use upper or lower case. (They are also followed in the C# class library):

- Names begin with capital letters (e.g., Length, WriteLine) except for local variables and parameters (e.g., i, len) or fields that are not visible outside their class.
- In composite words each word begins with a capital letter (for example, WriteLine). Joining words with "_" is seldom used in C#.
- Methods that do not return a value should begin with a verb (for example, DrawLine). All other names should generally begin with a noun (for example, Size, IndexOf, Collection). Fields or methods of type bool can also begin with an adjective if they express some boolean value (for example, Empty).

Characters and strings. Character constants are written between single quotes (for example, 'x'). String constants are written between double quotes (for example, "John"). In each, any characters can appear except the terminating quote, a line break or the character \, which is used as an *escape character*. The following es-

cape sequences allow the expression of special characters in character or string constants:

```
\     '
\     ""
\\    \
\0    0x0000 (the character with the value 0)
\a    0x0007 (alert)
\b    0x0008 (backspace)
\f    0x000c (form feed)
\n    0x000a (new line)
\r    0x000d (carriage return)
\t    0x0009 (horizontal tab)
\v    0x000b (vertical tab)
```

In order to write, for example

```
file "C:\sample.txt"
```

as a string constant we have to write:

```
"file \"C:\\sample.txt\""
```

As in names, Unicode values (for example, \u0061) can also be used in character or string constants.

If a string constant is preceded by the character @, it may contain line breaks, escape sequences remain unprocessed, and quotes must be doubled. So the above example can also be written:

```
@"file ""C:sample.txt"""
```

Integer constants. They can be expressed in decimal (e.g. 123) or hexadecimal (e.g. 0x007b). Their type is the smallest type from int, uint, long or ulong that their value will fit into. The suffix u or U (e.g. 123u) defines the constant to be of the smallest suitable unsigned type (uint or ulong). The suffix l or L (e.g. 0x007bl) indicates that the value has the smallest type of the set long and ulong.

Floating-point constants. They consist of an integer part, a decimal part and an exponent (e.g., 3.14E0 means $3.14 * 10^0$). Any of these parts can be omitted, but at least one must appear. The numbers 3.14, 314E-2 and .314E1 are valid notations for the same value. The type of a floating-point constant is double. Appending f or F (e.g. 1f) determines the type to be float. Appending m or M (e.g. 12.3m) determines the type to be decimal.

Comments. There are two forms of comments: *Single-line comments* begin with // and extend until the end of the line, for example:

```
// a comment
```

Delimited comments begin with /* and end with */. They can extend over several lines but cannot be nested. Example:

```
/*  a comment
    that takes two lines */
```

Single-line comments are used for short annotations and delimited comments mainly for commenting-out code.

2.3 Types

The data types of C# form a hierarchy, shown in Figure 2.2. There are value types and reference types. ***Value types*** are primitive types such as char, int or float, as well as enumerations and structs. Variables of these types directly contain a value (such as 'x', 123 or 3.14). ***Reference types*** are classes, interfaces, arrays and delegates. Variables of these types hold a reference to an object that is stored in the dynamic storage area (the ***heap***).

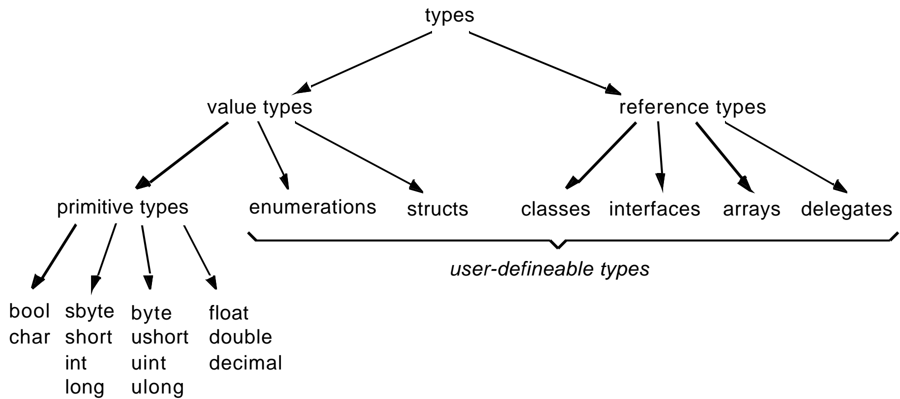

Figure 2.2 *Type hierarchy*

Uniform Type System

C# has a uniform type system, which means that all types, whether value types or reference types, are compatible with the type object (see Section 2.3.7): Values of any type can be assigned to object variables and understand object operations. This makes it easy to design algorithms that can work with any kind of data. Table 2.1 summarizes the differences between value types and reference types.

Table 2.1 Value types and reference types

	value types	*reference types*
Variable contains	a value	a reference to an object
Variable is stored	on the method stack or in the containing object	on the heap
Assignment	copies the value	copies the reference

Example	int i = 17;
	int j = i;
	i [17]
	j [17]
		string s = "Hello";
		string s1 = s;
		s → Hello
		s1 → Hello

The type string, used in Table 2.1, is a predefined class and thus a reference type. Actually string is a keyword that the compiler expands into the class System.String (that is, the class String in the namespace System). Similarly, object is expanded into the class System.Object.

2.3.1 Primitive Types

As with all languages, C# has predefined types for numbers, characters and boolean values. Numeric types are divided into integer types and floating-point types. Within these groups the types differ in range and accuracy. Table 2.2 shows an overview of all primitive types.

Table 2.2 Primitive types

	range of values	*expanded to*
sbyte	-128 .. 127	System.SByte
short	-32768 .. 32767	System.Int16
int	-2147483648 .. 2147483647	System.Int32
long	-2^{63} .. $2^{63}-1$	System.Int64
byte	0 .. 255	System.Byte
ushort	0 .. 65535	System.UInt16
uint	0 .. 4294967295	System.UInt32
ulong	0 .. $2^{64}-1$	System.UInt64
float	±1.4E-45 .. ±3.4E38 (32 Bit, IEEE 754)	System.Single
double	±5E-324 .. ±1.7E308 (64 Bit, IEEE 754)	System.Double
decimal	±1E-28 .. ±7.9E28 (128 Bit)	System.Decimal
bool	true, false	System.Boolean
char	Unicode characters	System.Char

The unsigned types byte, ushort, uint and ulong are mainly used for systems programming and for compatibility with other languages. The type decimal allows the representation of large decimal numbers with high accuracy and is mainly used for financial mathematics.

The compiler maps all primitive types to struct types defined in the namespace System. For example, the type int is mapped to System.Int32. All the operations defined there (including those inherited from System.Object) are thus applicable to int.

There is a compatibility relationship between most of the primitive types. This is shown in Figure 2.3. An arrow between char and ushort means, for example, that char values can be assigned to a ushort variable (ushort includes all char values). The relationship is transitive. That means that char values can also be assigned to int or float variables. An assignment to decimal is however only permitted with an explicit type cast (e.g., decimal d = (decimal) 3;). In assigning values of type long or ulong to float there can be a loss of accuracy if there are insufficient bits in the mantissa to represent the result.

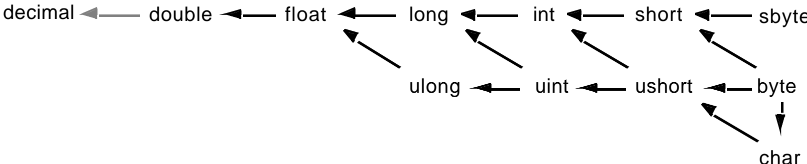

Figure 2.3 *Compatibility relationship between primitive types.*

2.3.2 Enumerations

Enumerations are types whose values are explicitly given by a list of named constants, for example:

```
enum Color { red, blue, green }
```

Variables of type Color can take the values red, blue or green, and the compiler maps these values to the numbers 0, 1 and 2. However, enumerations are not numeric types; they cannot be assigned to numeric variables, and numbers cannot be assigned to Color variables. If desired, the value of an enumeration constant can be specified in the declaration, as in

```
enum Color { red=1, blue=2, green=4 }
enum Direction { left=0, right, up=4, down }   // left=0, right=1, up=4, down=5
```

Enumerations usually occupy four bytes. However, a different type size can be chosen by writing a (numeric) base type after the enumeration type name. For example:

```
enum Access: byte { personal=1, group=2, all=4 }
```

Variables of type Access are thus one byte long. Enumerations can be used as follows:

```
Color c = Color.blue;
Access a = Access.personal | Access.group;
if ((a & Access.personal) != 0) Console.WriteLine("access granted");
```

When using enumeration constants they must be qualified with their type names. If values are chosen to be powers of two (as in the type Access) one can form bit sets using the logical operators &, | and ~. In this way an enumeration variable can hold a set of values. If an operation yields a value for which there is no enumeration constant, that does not bother the compiler. (for example, Access.personal | Access.group yields the value 3). The following operations are allowed with enumerations:

```
==, !=, <, <=, >, >=       if (c == Color.red) ...
                           if (c > Color.red && c <= Color.green) ...
+, -                       c = c + 2;
++, --                     c++;
&                          if ((a & Access.personal) != 0) ...
|                          a = a | Access.group;
~                          a = ~ Access.all;   // one's complement
```

As with the logical operations, an arithmetic operation can yield a value that does not map to any enumeration constant. The compiler accepts this.

Enumerations inherit all the operations of object, such as Equals or ToString (see Section 2.3.7). There is also a class System.Enum that provides special operations on enumerations.

2.3.3 Arrays

Arrays are one- or multi-dimensional vectors of elements. The elements are selected by an index, where the indexing begins at 0.

One-dimensional arrays. One-dimensional arrays are declared by stating their element type followed by empty square brackets:

```
int[] a;                               // declares an array variable a
int[] b = new int[3];                  // initializes b with an array of 3 elements, all 0
int[] c = new int[] {3, 4, 5};         // initializes c with the values 3, 4, 5
int[] d = {3, 4, 5};                   // initializes d with the values 3, 4, 5
SomeClass[] e = new SomeClass[10];     // creates an array of references
SomeStruct[] f = new SomeStruct[10];   // creates an array of values (directly in the array)
```

An array declaration does not allocate storage. Therefore it does not specify an array length. In order to create an array object one has to use the new operator with

the desired element type and length. For example, new int[3] creates an array of three int elements. The values of a newly created array are initialized to 0 (or '\0', false, null as appropriate), except when explicit initial values are specified in curly braces. In the declaration of an array the initialization can also be given directly (without using the new operator), in which case the compiler creates an array of the necessary length.

Note that an array of classes contains *references*, whereas an array of structs holds the *values* directly.

Multi-dimensional arrays. Multi-dimensional arrays can either be jagged or rectangular. Jagged arrays hold references to other arrays, whereas the elements of rectangular arrays form a contiguous block of memory (see Figure 2.4). Rectangular arrays are not only more compact, but also allow a more efficient indexing. Here are some examples of multi-dimensional arrays:

```
// jagged arrays (declared with [][])
int[][] a = new int[2][];          // two rows, the columns of which are still undefined
a[0] = {1, 2, 3};                  // row 0 has three columns
a[1] = {4, 5, 6, 7, 8};            // row 1 has five columns

// rectangular arrays (declared with [,])
int[,] a = new int[2, 3];          // two rows with three columns each
int[,] b = {{1, 2, 3}, {4, 5, 6}}; // initialization of the two rows and three columns
int[,,] c = new int[2, 4, 2];      // two blocks with four rows with two columns each
```

In jagged arrays the rows can have different lengths. For this reason only the length of the first dimension is specified in the new operation and not the length of all dimensions, as with rectangular arrays. Figure 2.4 shows the difference between the two styles of arrays:

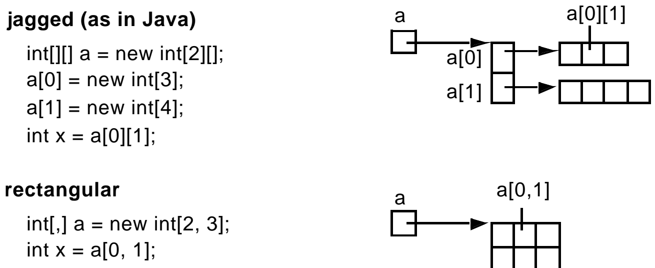

Figure 2.4 *Jagged and rectangular multi-dimensional arrays.*

Array operations. As can be seen from Figure 2.4, arrays variables hold references. Therefore an array assignment is a ***reference assignment***, i.e., the array itself is not copied. Indexing always begins at 0. The length of an array can be determined with the Length operator.

```
int[] a = new int[3];
int[][] b = new int[2][];
b[0] = new int[4];
b[1] = new int[4];
Console.WriteLine(a.Length);          // 3
Console.WriteLine(b.Length);          // 2
Console.WriteLine(b[0].Length);       // 4
```

In rectangular arrays Length gives the total number of elements. In order to get the number of elements in a certain dimension one must use the GetLength method.

```
int[,] a = new int[3, 4];
Console.WriteLine(a.Length);          // 12
Console.WriteLine(a.GetLength(0));    // 3
Console.WriteLine(a.GetLength(1));    // 4
```

The class System.Array contains some useful operations for copying, sorting and searching in arrays.

```
int[] a = new int[2];
int[] b = {7, 2, 4};
Array.Copy(b, a, 2);                  // copies b[0..1] to a
Array.Sort(b);                        // sorts b into ascending order
```

Variable-length arrays. Once an array has been allocated, its length is fixed. However, there is a class System.Collections.ArrayList that implements arrays of *variable* length (see Section 4.1.5). The method Add can be used to add elements of any type to the array. The elements can then be selected by indexing:

```
using System;
using System.Collections;

class Test {
    static void Main() {
        ArrayList a = new ArrayList();       // creates an empty array of variable length
        a.Add("Alice");                      // appends "Alice" to the end of the array
        a.Add("Bob");
        a.Add("Cecil");
        for (int i = 0; i < a.Count; i++)    // a.Count returns the number of elements
            Console.WriteLine(a[i]);         // output: "Alice", "Bob", "Cecil"
    }
}
```

Associative arrays. The class System.Collections.Hashtable allows arrays not only to be indexed by numbers but also, for example, by strings:

```
using System;
using System.Collections;

class Test {
    static void Main() {
        Hashtable phone = new Hashtable();        // creates an empty associative array
        phone["Jones"] = 4362671;
        phone["Miller"] = 2564439;
        phone["Smith"] = 6451162;
        foreach (DictionaryEntry x in phone) {    // foreach: see Section 2.6.9
            Console.Write(x.Key + " = ");          // key, e.g. "Miller"
            Console.WriteLine(x.Value);           // value, e.g. 2564439
        }
    }
}
```

2.3.4 Strings

Character arrays (strings) occur so often that C# provides a special type string for them. The compiler expands this into the class System.String. A string constant or a string variable can be assigned to another string variable:

```
string s = "Hello";
string s2 = s;
```

Strings can be indexed like arrays (e.g., s[i]), but they are not actually arrays. In particular, they cannot be modified. If one needs strings that can be modified one should use the class System.Text.StringBuilder instead:

```
using System;
using System.Text;

class Test {
    static void Main(string[] arg) {
        StringBuilder buffer = new StringBuilder();
        buffer.Append(arg[0]);
        buffer.Insert(0, "myfiles\\");
        buffer.Replace(".cs", ".exe");
        Console.WriteLine(buffer.ToString());
    }
}
```

This example also shows that the Main method can be declared as having a string array parameter to which command-line arguments are passed. If the above program were to be called as

```
Test sample.cs
```

the output would be myfiles\sample.exe.

Strings are reference types, that is, a string variable holds a reference to a string object. String assignments are therefore ***reference assignments***; the value of the string is not copied. However, the operations == and != are, in contrast to Java, value comparisons. The comparison

```
(s+ " World") == "Hello World"
```

returns the value true. The compare operations <, <=, >, >= are not allowed for strings; instead, the method CompareTo must be used (see Table 2.3). Strings can be concatenated with + (e.g., s + " World" gives "Hello World"). This creates a new string object (so s is not changed). The length of a string can be obtained by using s.Length, as for arrays. The class System.String offers many useful operations (see Table 2.3):

Table 2.3 *String operations (extract)*

operation	*returns*
s.CompareTo(s1)	-1, 0 or 1 according to whether s < s1, s == s1 or s > s1
s.IndexOf(s1)	the index of the first occurrence of s1 in s
s.LastIndexOf(s1)	the index of the last occurrence of s1 in s
s.Substring(from, length)	the substring s[from..from+length-1]
s.StartsWith(s1)	true if s starts with s1
s.EndsWith(s1)	true if s ends with s1
s.ToUpper()	a copy of s in upper-case letters
s.ToLower()	a copy of s in lower-case letters
String.Copy(s)	a copy of s

2.3.5 Structs

Structs are user-defined types that hold data and possibly methods. They are declared as follows:

```
struct Point {
    public int x, y;                                             // fields
    public Point(int a, int b) { x = a; y = b; }                 // constructor
    public void MoveTo(int x, int y) { this.x = x; this.y = y; }  // method
}
```

Structs are ***value types***. Therefore variables of type Point hold the values of the fields x and y directly. An assignment between structs is a value assignment and not a reference assignment.

```
Point p;                 // p is so far uninitialized
p.x = 1; p.y = 2;        // field access
Point q = p;             // value assignment (q.x == 1, q.y == 2)
```

Structs can be initialized with a *constructor*. The declaration

```
Point p = new Point(3, 4);
```

declares a new struct object p on the stack and calls the constructor of Point, which initializes the fields to the values 3 and 4. A constructor must always have the same name as the struct type. The method MoveTo is called as follows:

```
p.MoveTo(10, 20);
```

In the code of the called method the object p, on which the method was called, can be referred to by this (the so-called *receiver* of the message MoveTo). Thus this.x denotes the field x of the object p, whereas x denotes the formal parameter of the method MoveTo. When there is no ambiguity this can be omitted from the field access, as above in the constructor of Point.

Structs may not declare parameterless constructors. However, a parameterless constructor may be used on structs, because the compiler creates one for every struct type. The constructor in the declaration

```
Point p = new Point();
```

initializes the fields of p with the value 0. Section 2.8 goes into further detail on structs and constructors.

2.3.6 Classes

Like structs, classes are types consisting of data and methods. In contrast to structs, however, they are *reference types*. That is, a variable of a class type holds a reference to an object that is stored in the heap. Classes are declared as follows:

```
class Rectangle {
   Point origin;   // bottom-left corner
   public int width, height;
   public Rectangle() { origin = new Point(0, 0); width = height = 1; }
   public Rectangle(Point p, int w, int h) { origin = p; width = w; height = h; }
   public void MoveTo(Point p) { origin = p; }
}
```

A Rectangle variable can only be used if a Rectangle object has been installed in it:

```
Rectangle r = new Rectangle(new Point(10, 20), 5, 5);
int area = r.width * r.height;
```

Whenever an object is created with the new operator the appropriate constructor is automatically called. This initializes the fields of the object. The class Rectangle has two constructors that differ in their parameter lists. The parameters of the second constructor match the actual parameters of the new operator in the example above and so this constructor is chosen. The declaration of constructors or meth-

ods with the same name in a class is called *overloading*. This will be discussed further in Section 2.8.

Created objects are never explicitly released in C#. Instead this task is left to the *garbage collector*, which automatically releases objects once they are no longer referenced. This removes the source of many awkward errors that C++ programmers have to struggle with: if objects are released too soon, some references may point into a void. On the other hand, if a C++ programmer forgets to release objects they remain as "memory leaks". Under .NET such errors cannot occur, because the garbage collector takes care of releasing objects.

Because a variable of a class type holds a reference, the assignment

```
Rectangle r1 = r;
```

is a *reference assignment*: afterwards r and r1 point to the same object. Methods are called as for structs:

```
r.MoveTo(new Point(3, 3));
```

In the implementation of the method MoveTo the predefined name this again denotes the object r on which the method was called. Because the field name origin is unambiguous there it does not need to be qualified as this.origin.

Table 2.4 compares classes and structs again. Some of these differences are covered in detail in later sections.

Table 2.4 *Classes versus structs*

classes	*structs*
reference types (variables reference objects on the heap)	value types (variables contain objects)
support inheritance (all classes are derived from object)	do not support inheritance (but are compatible with object)
can implement interfaces	can implement interfaces
parameterless constructor can be declared	parameterless constructor cannot be declared

Structs are lightweight types that are often used for temporary data. Because they are not stored on the heap they do not burden the garbage collector. Classes are mainly used for more complex objects that are often linked into dynamic data structures. Objects of a class can outlive the methods that created them.

2.3.7 object

The type object, which is expanded into System.Object, has a special meaning in C#. It is the root of the entire type hierarchy. That means that all types are compatible with it. So a value of any type can be assigned to an object variable:

```
object obj = new Rectangle();        // assignment of Rectangle to object
Rectangle r = (Rectangle) obj;       // type cast
obj = new int[3];                    // assignment of int[] to object
int[] a = (int[]) obj;               // type cast
```

In particular, object allows the implementation of generic container classes. For example, a stack that stores objects of any type can have a method

```
void Push(object x) {...}
```

that can then be called with parameters of any type:

```
Push(new Rectangle());
Push(new int[3]);
```

The class System.Object (covered in detail in Section 2.9.8) contains several methods that are inherited by all classes and structs (and are mostly overridden). The most important methods are:

```
class Object {
    public virtual bool Equals(object o) {...}  // compares the values of the receiver and o
    public virtual string ToString() {...}      // converts the object into a string
    public virtual int GetHashCode() {...}      // calculates a hash code for the object
    ...
}
```

These methods can be applied to objects of any type and even to constants:

```
string s = 123.ToString();    // s == "123"
```

2.3.8 Boxing and Unboxing

Not only reference types are compatible with object but also value types such as int, structs or enumerations. In the assignment

```
object obj = 3;
```

the value 3 is wrapped up by an object of a temporary class that is then assigned to the variable obj (see Figure 2.5). This is called *boxing*.

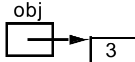

Figure 2.5 *Boxing of an int value*

Assignment in the opposite direction needs a type cast:

```
int x = (int) obj;
```

In this, the value is unwrapped from the temporary object and treated as an int value. This is called ***unboxing***.

Boxing and unboxing are particularly useful with container types because these can then be used not only with elements of reference types, but also with elements of value types. For example, if a class Queue has been declared as follows:

```
class Queue {
    object[] values = new object[10];
    public void Enqueue(object x) {...}
    public object Dequeue() {...}
}
```

it can be used like this:

```
Queue q = new Queue();
q.Enqueue(new Rectangle());            // used with a reference type
q.Enqueue(3);                          // used with a value type (boxing)
...
Rectangle r = (Rectangle) q.Dequeue(); // type cast: object -> Rectangle
int x = (int) q.Dequeue();             // type cast: object -> int (unboxing)
```

Calls to object methods are forwarded to the boxed object. For example, the code fragment:

```
object obj = 123;
string s = obj.ToString();
```

assigns the value "123" to s.

2.4 Expressions

Expressions consist of operands and operators and calculate values. Table 2.5 shows the operators of C# ordered by priority. Operators higher up in the list have priority over operators lower down in the list. Binary operators at the same level are evaluated from left to right in an expression, for example:

```
... a + b - c ...        // ... (a + b) - c ...
```

The unary operators +, -, !, ~, as well as type casts are right-associative, that is, they are evaluated from right to left, for example:

```
... - (int) x ...        // ... - ((int) x) ...
```