This book was set in Times by the authors using LaTeX.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Sestoft, Peter.
C# precisely / Peter Sestoft and Henrik I. Hansen.
p. cm.
Includes bibliographic references and index.
ISBN 0-262-69317-8 (pbk.: alk. paper)
1. C# (Computer program language) I. Hansen, Henrik I. II. Title.
QA76.73.C154S47 2004
005.13′3—dc22
2004048288

10 9 8 7 6 5 4 3 2

Preface

This book describes the programming language C# (pronounced "c sharp"), version 2.0. It is a quick reference for the reader who has already learnt or is learning C# from a standard textbook and who wants to know the language in more detail. It should be particularly useful for readers who know the Java programming language and who want to learn C#.

C# is a class-based single-inheritance object-oriented programming language designed for the Common Language Runtime of Microsoft's .Net platform, a managed execution environment with a typesafe intermediate language and automatic memory management. Thus C# is similar to the Java programming language in many respects, but it is different in almost all details. In general, C# favors programmer convenience over language simplicity. It was designed by Anders Hejlsberg, Scott Wiltamuth and Peter Golde from Microsoft Corporation.

C# includes many useful features not found in Java: struct types, operator overloading, reference parameters, rectangular multi-dimensional arrays, user-definable conversions, properties and indexers (stylized methods) and delegates (methods as values), but omits Java's inner classes. See section 29 for a summary of the main differences.

C# may appear similar to C++, but its type safety is much better and its machine model is very different because of managed execution. In particular, there is no need to write destructors and finalizers, nor to aggressively copy objects or keep track of object ownership.

This book presents C# version 2.0 as used in Microsoft Visual Studio 2005, including generics, iterators, anonymous methods and partial type declarations, but excluding most of Microsoft's .Net Framework class libraries except threads, input-output, and generic collection classes. The book does not cover unsafe code, destructors, finalization, reflection, pre-processing directives (`#define`, `#if`, ...) or details of IEEE754 floating-point numbers.

General rules of the language are given on left-hand pages, and corresponding examples are shown on the facing right-hand page for easy reference. All examples are fragments of legal C# programs, available from <http://www.itu.dk/people/sestoft/csharpprecisely/>. For instance, you will find the code for example 17 in file `Example17.cs`.

This second printing has been updated for the November 2005 final release of Microsoft Visual Studio.

Acknowledgements: Thanks to a stay at Microsoft Research in Cambridge, England, we could experiment with a very early version of Generic C#. Later, the .Net Framework Alpha Program provided an implementation of all the new C# 2.0 features, and Ecma International provided C# standards documents. Special thanks to Andrew Kennedy, Don Syme, Claudio Russo and Simon Peyton Jones for directly or indirectly making this possible. The Mono project developers provided another neat C# compiler and run-time environment, and rapid bug fixes. Thanks to Hans Dybkjær, Jørgen Steensgaard-Madsen, Jon Jagger and Niels Peter Svenningsen for comments and suggestions on draft manuscripts, and to Carsten Jørgensen, Lawrence Berg and Ken Friis Larsen for corrections to the published book. It was a pleasure to work with Robert Prior, Valerie Geary and Deborah Cantor-Adams at The MIT Press. Thanks also to the Royal Veterinary and Agricultural University, Denmark and the IT University of Copenhagen, Denmark, for their support.

5 Data and Types

A *type* is a set of data values and operations on them. Every variable, parameter, and field has a declared type, every method has a declared return type, and so on. The compiler will infer a type for every expression based on this information. This *compile-time type* determines which operations can be performed on the value of the expression.

Types are used in declarations of local variables; in declarations of classes, interfaces, struct types, and their members; in delegate types; in object and array creation expressions (sections 9 and 12.9); in type cast expressions (section 12.18); and in instance test expressions (section 12.11).

A type is either a value type (section 5.1) or a reference type (section 5.2).

5.1 Value Types and Simple Types

A *value type* is either a simple type (this section), a struct type (section 14), or an enum type (section 16). A variable of value type directly contains a value of that type, not just a reference to it. Assigning a value of value type to a variable or field or array element of value type makes a copy of the value.

A *simple type* is either `bool` or one of the numeric types. A *numeric* type is a signed or unsigned integer type, including the character type, or a floating-point type, or the fixed-point type `decimal` which is useful for exact calculations such as financial accounting. The tables opposite show the simple types, some example constants, value range, kind, and size (in bytes). For escape sequences such as `\u0000` in character constants, see page 16. Integer constants may be written in decimal or hexadecimal notation:

Notation	Base	Distinction	Example Integer Constants
Decimal	10		`1234567890`, `0127`, `-127`
Hexadecimal	16	Leading `0x`	`0x12ABCDEF`, `0x7F`, `-0x7F`

Two's complement representation is used for the signed integer types (`sbyte`, `short`, `int`, and `long`). The integer types are exact. The floating-point types are inexact and follow the IEEE754 floating point standard, with the number of significant digits indicated opposite.

For each simple type there is a predefined struct type (in the System namespace), also shown opposite. The simple type is an alias for the struct type and therefore has members:

- `int.Parse(String s)` of type `int` is the integer obtained by parsing `s`; see example 1. It throws ArgumentNullException if `s` is `null`, FormatException if `s` cannot be parsed as an integer, and OverflowException if the parsed number cannot be represented as an `int`. All simple types have similar `Parse` methods. The floating-point `Parse` methods are culture sensitive; see section 7.2.
- The smallest and greatest possible values of each numeric type are represented by constant fields `MinValue` and `MaxValue`, such as `int.MinValue` and `int.MaxValue`.
- The `float` and `double` types define several constants: `double.Epsilon` is the smallest number of type `double` greater than zero, `double.PositiveInfinity` and `double.NegativeInfinity` represent positive and negative infinity, and `double.NaN` is a `double` value that is not a number. These values are determined by the IEEE754 standard.
- The `decimal` type defines the constants `MinusOne`, `Zero`, and `One` of type `decimal` along with methods for computing with and converting numbers of type `decimal`.

Example 7 Three Equivalent Declarations of an Integer Variable

```
using System;
...
int i1;
Int32 i2;
System.Int32 i3;
```

Simple Types: Constants, Default Value, and Range

Type	Example Constants	Default Value	Range (`MinValue`...`MaxValue`)
`bool`	`true`	`false`	`false`, `true`
`char`	`'A'`, `'\u0041'`	`'\u0000'`	`'\u0000'` ... `'\uFFFF'`
`sbyte`	`-119`	`0`	$-128 \ldots 127$
`byte`	`219`	`0`	$0 \ldots 255$
`short`	`-30319`	`0`	$-32768 \ldots 32767$
`ushort`	`60319`	`0`	$0 \ldots 65535$
`int`	`-2111222319`	`0`	$-2147483648 \ldots 2147483647$
`uint`	`4111222319`	`0`	$0 \ldots 4294967295$
`long`	`-411122319L`	`0`	$-9223372036854775808 \ldots 9223372036854775807$
`ulong`	`411122319UL`	`0`	$0 \ldots 18446744073709551615$
`float`	`-1.99F`, `3E8F`	`0.0`	$\pm 10^{-44} \ldots \pm 10^{38}$, 7 significant digits
`double`	`-1.99`, `3E8`	`0.0`	$\pm 10^{-323} \ldots \pm 10^{308}$, 15–16 significant digits
`decimal`	`-1.99M`	`0.0`	$\pm 10^{-28} \ldots \pm 10^{28}$, 28–29 significant digits (*)

(*) May be changed to range $\pm 10^{-6143} \ldots \pm 10^{6144}$ and 34 significant digits (IEEE754 decimal128).

Simple Types: Kind, Size, and Struct Name

Type Alias	Kind	Size	Struct Type
`bool`	logical	1	System.Boolean
`char`	unsigned integer	2	System.Char
`sbyte`	integer	1	System.SByte
`byte`	unsigned integer	1	System.Byte
`short`	integer	2	System.Int16
`ushort`	unsigned integer	2	System.UInt16
`int`	integer	4	System.Int32
`uint`	unsigned integer	4	System.UInt32
`long`	integer	8	System.Int64
`ulong`	unsigned integer	8	System.UInt64
`float`	floating-point	4	System.Single
`double`	floating-point	8	System.Double
`decimal`	fixed-point	16	System.Decimal

When `t` is one of these types, then `sizeof(t)` is a compile-time constant expression whose value is Size.

9.3 Class Array

All array types are derived from class Array, and the members of an array type are those inherited from class Array. Let `a` be a reference of array type, `o` an object of any type, and `i`, `i1`, ..., `in` integers. Then:

- `a.Length` of type `int` is the length of `a`, that is, the total number of elements in `a`, if `a` is a one-dimensional or a rectangular multi-dimensional array, or the number of elements in the first dimension of `a`, if `a` is a jagged array.
- `a.Rank` of type `int` is the rank of `a`; see sections 9 and 9.2.2.
- `a.GetEnumerator()` of type IEnumerator is a non-generic enumerator (section 24.2) for iterating through `a`. This enables the `foreach` statement to iterate over an array; see section 13.6.2 and example 37. If `a` is a one-dimensional array of type `t[]`, one can get a type-safe generic enumerator of type IEnumerator<`t`> by computing `((IList<t>)a).GetEnumerator()`.
- `a.GetLength(i)` of type `int` is the number of elements in dimension `i`; see examples 24 and 36.
- `a.SetValue(o, i1,..,in)` of type `void` performs the same assignment as `a[i1,...,in] = o` when `a` has rank `n`; and `a.GetValue(i1,...,in)` of type Object is the same as `a[i1,...,in]`. More precisely, if `a[i1,...,in]` has reference type, then `GetValue` returns the same reference; otherwise it returns a boxed copy of the value of `a[i1,...,in]`.
- `a.Equals(o)` of type `bool` returns true if `a` and `o` refer to the same array object, otherwise false.

Class Array provides static utility methods, some of which are listed below. These methods can be used on the ordinary array types `t[]` which derive from class Array. The methods throw ArgumentNullException if the given array `a` is `null`, and throw RankException if `a` is not one-dimensional.

- `static int BinarySearch(Array a, Object k)` searches the one-dimensional array `a` for `k` using binary search. Returns an index `i>=0` for which `a[i].CompareTo(k) == 0`, if any; otherwise returns `i<0` such that `~i` would be the proper position for `k`. The array `a` must be sorted, as by `Sort(a)`, or else the result is undefined; and its elements must implement IComparable.
- `static int BinarySearch(Array a, Object k, IComparer cmp)` works as above, but compares array elements using the method `cmp.Compare`; see section 24.3. The array must be sorted, as by `Sort(a, cmp)`, or else the result is undefined.
- `static void Reverse(Array a)` reverses the contents of one-dimensional array `a`.
- `static void Reverse(Array a, int i, int n)` reverses the contents of `a[i..(i+n-1)]`.
- `static void Sort(Array a)` sorts the one-dimensional array `a` using quicksort, comparing array elements using their `CompareTo` method; see section 24.3. The array elements must implement IComparable. The sort is not stable: elements that are equal may be swapped.
- `static void Sort(Array a, IComparer cmp)` works as above, but compares array elements using the method `cmp.Compare`; see section 24.3.
- `static void Sort(Array a, int i, int n)` works as above but sorts `a[i..(i+n-1)]`.
- `static void Sort(Array a, int i, int n, IComparer cmp)` sorts `a[i..(i+n-1)]` using `cmp.Compare`.

Expression	Meaning	Section	Assoc'ty	Argument(s)	Result type
`a[...]`	array access	9		`t[]`, integer	`t`
`o[...]`	indexer access	12.17		object	`t`
`o.f`	field or property access	12.13, 12.16		object	type of `f`
`C.f`	static field or property	12.13, 12.16		class/struct	type of `f`
`o.M(...)`	method call	12.15		object	return type of `M`
`C.M(...)`	static method call	12.15		class/struct	return type of `M`
`new t[...]`	create array	9		type	`t[]`
`new t(...)`	create object/struct/delegate	12.9, 17		class/struct/delegate	`t`
`default(t)`	default value for type `t`	6.2		type	`t`
`sizeof(t)`	size in bytes	5.1		(simple) type	`int`
`typeof(t)`	type determination	12.19		type/`void`	System.Type
`checked(e)`	overflow checking	12.3		integer	integer
`unchecked(e)`	no overflow checking	12.3		integer	integer
`delegate ...`	anonymous method	12.20			delegate
`x++`	postincrement	12.2		numeric	numeric
`x--`	postdecrement	12.2		numeric	numeric
`++x`	preincrement	12.2		numeric	numeric
`--x`	predecrement	12.2		numeric	numeric
`-x`	negation (minus sign)	12.2	right	numeric	`int`/`long`
`~e`	bitwise complement	12.5	right	integer	`(u)int`/`(u)long`
`!e`	logical negation	12.4	right	`bool`	`bool`
`(t)e`	type cast	12.18		type, any	`t`
`e1 * e2`	multiplication	12.2	left	numeric, numeric	numeric
`e1 / e2`	division	12.2	left	numeric, numeric	numeric
`e1 % e2`	remainder	12.2	left	numeric, numeric	numeric
`e1 + e2`	addition	12.2	left	numeric, numeric	numeric
`e1 + e2`	string concatenation	7	left	String, any	String
`e1 + e2`	string concatenation	7	left	any, String	String
`e1 + e2`	delegate combination	17	left	delegate, delegate	delegate
`e1 - e2`	subtraction	12.2	left	numeric	numeric
`e1 - e2`	delegate removal	17	left	delegates	delegate
`e1 << e2`	left shift	12.5	left	integer, `int`	`(u)int`/`(u)long`
`e1 >> e2`	right shift	12.5	left	integer, `int`	`(u)int`/`(u)long`
`e1 < e2`	less than	12.6		numeric	`bool`
`e1 <= e2`	less than or equal to	12.6		numeric	`bool`
`e1 >= e2`	greater than or equal to	12.6		numeric	`bool`
`e1 > e2`	greater than	12.6		numeric	`bool`
`e is t`	instance test	12.11		any, type	`bool`
`e as t`	instance test and cast	12.12		any, type	`t`
`e1 == e2`	equal	12.6	left	compatible	`bool`
`e1 != e2`	not equal	12.6	left	compatible	`bool`
`e1 & e2`	bitwise and	12.5	left	integer, integer	`(u)int`/`(u)long`
`e1 & e2`	logical strict and	12.4	left	`bool`, `bool`	`bool`
`e1 ^ e2`	bitwise exclusive-or	12.5	left	integer, integer	`(u)int`/`(u)long`
`e1 ^ e2`	logical strict exclusive-or	12.4	left	`bool`, `bool`	`bool`
`e1 | e2`	bitwise or	12.5	left	integer	`(u)int`/`(u)long`
`e1 | e2`	logical strict or	12.4	left	`bool`	`bool`
`e1 && e2`	logical and	12.4	left	`bool`	`bool`
`e1 || e2`	logical or	12.4	left	`bool`	`bool`
`e1 ?? e2`	null-coalescing	18	right	nullable/reftype, any	any
`e1 ? e2 : e3`	conditional	12.8	right	`bool`, any, any	any
`x = e`	assignment	12.7	right	`e` impl. conv. to `x`	type of `x`
`x += e`	compound assignment	12.7	right	compatible	type of `x`
`x += e`	event assignment	10.17	right	event, delegate	`void`
`x -= e`	event assignment	10.17	right	event, delegate	`void`

12.5 Bitwise Operators and Shift Operators

The operators ~ (bitwise complement) and & (bitwise and) and ^ (bitwise exclusive-or) and | (bitwise or) may be used on operands of enum type or integer type. The operators work in parallel on all bits of the operands and never cause overflow, not even in a checked context. The two's complement representation is used for signed integer types, so ~n equals (-n)-1 and also equals (-1)^n.

The shift operators << and >> shift the bits of the two's complement representation of the first argument; they never cause overflow, not even in a checked context. The two operands are promoted (page 56) separately, and the result type is the promotion type of the first argument.

Thus the shift operation is always performed on a 32-bit (int or uint) or a 64-bit (long or ulong) value. In the former case, the length of the shift is between 0 and 31 as determined by the 5 least significant bits of the second argument; in the latter case, the length of the shift is between 0 and 63 as determined by the 6 least significant bits of the second argument. This holds also when the second argument is negative: the length of the shift is the non-negative number determined by the 5 or 6 least significant bits of its two's complement representation.

Thus the left shift n<<s equals n*2*2*...*2 where the number of multiplications is determined by the 5 or 6 least significant bits of s, according as n was promoted to a 32-bit value or to a 64-bit value.

For signed integer types, the operator >> performs right shift with sign-bit extension: the right shift n>>s of a non-negative n equals n/2/2/.../2 where the number of divisions is determined by the 5 or 6 least significant bits of s. The right shift of a negative n equals ~((~n)>>s). In other words, the low-order s bits of n are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero if n is non-negative and set to one if n is negative.

For unsigned integer types, the operator >> performs right shift with zero extension: the right shift n>>s equals n/2/2/.../2 where the number of divisions is determined by the 5 or 6 least significant bits of s. In other words, the low-order bits of n are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero.

See example 205 for clever and intricate use of bitwise operators. This may be efficient and good style on a tiny embedded processor, but not in general programming.

12.6 Comparison Operators

The *comparison operators* == and != require the operand types to be *compatible*: one must be implicitly convertible to the other. Operand types that are generic type parameters must be constrained to be reference types (section 23.4), unless one operand is null. Two values of simple type are equal (by ==) if they represent the same value after conversion. For instance, 10 and 10.0 are equal. Two values of a reference type that does not override the default implementation of the operators are equal (by ==) if both are null, or both are references to the same object or array, created by the same execution of the new-operator.

Class String redefines the == and != operators so that they compare the characters in the strings. Hence two strings s1 and s2 may be equal by s1==s2, yet be distinct objects and therefore unequal by (Object)s1==(Object)s2; see example 15.

Values of struct type can be compared using == and != only if the operators have been explicitly defined for that struct type. The default Equals method (section 5.2) compares struct values field by field.

The operators < <= >= > can be used on numeric types (and on user-defined types; section 10.15). They perform signed comparison on signed integer types, and unsigned comparison on unsigned ones.

13.9 The `checked` and `unchecked` Statements

An operation on integral numeric values may produce a result that is too large. In a *checked context*, run-time integer overflow will throw an OverflowException, and compile-time integer overflow will produce a compile-time error. In an *unchecked context*, both compile-time and run-time integer overflow will wrap around (discard the most significant bits), not throw any exception nor produce a compile-time error.

A `checked` statement creates a checked context for a *block-statement* and has the form

```
checked
  block-statement
```

An `unchecked` statement creates an unchecked context for a *block-statement* and has the form

```
unchecked
  block-statement
```

In both cases, only arithmetic operations textually inside the *block-statement* are affected. Thus arithmetic operations performed in methods called from the *block-statement* are not affected, whereas arithmetic operations in anonymous delegate expressions textually inside the *block-statement* are affected. Checked and unchecked statements are analogous to checked and unchecked expressions; see section 12.3.

13.10 The `using` Statement

The purpose of the `using` statement is to ensure that a resource, such as a file handle or database connection, is released as soon as possible after its use. The `using` statement may have the form

```
using (t res = initializer)
  body
```

This declares variable `res` to have type `t` which must be implicitly convertible to IDisposable, initializes `res` with the result of evaluating the *initializer* expression, and executes the *body*. Finally method `Dispose()` is called on `res`, regardless of whether *body* terminates normally, throws an exception, or exits by `return` or `break` or `continue` or `goto`. The *body* must be a statement but not a declaration statement or a labeled statement. Variable `res` is read-only; its scope is *body* where it cannot be assigned to, nor used as a ref or out argument. The `using` statement is equivalent to this block statement:

```
{
  t res = initializer;
  try     { body }
  finally { if (res != null) ((IDisposable)res).Dispose(); }
}
```

The IDisposable interface from namespace System describes a single method:

- `void Dispose()` is called to close or release resources, such as files, streams or database connections, held by an object. Calling this method multiple times must not throw an exception; all calls after the first one may just do nothing.

13.11 The `lock` Statement

The `lock` statement is described in section 20.2.

18 Nullable Types over Value Types (C# 2.0)

A *nullable type* `t?` is used to represent possibly missing values of type `t`, where `t` is a value type such as `int`, a non-nullable struct type, or a type parameter that has a `struct` constraint. A value of type `t?` either is non-null and contains a proper value of type `t`, or is the unique null value. The default value of a nullable type is null. The nullable type `t?` is an alias for System.Nullable<`t`> and is itself a value type.

A nullable type such as `int?` is useful because the custom of using `null` to represent a missing value of reference type does not carry over to value types: `null` is not a legal value of the plain value type `int`.

Values `v1` and `v2` of nullable type support the following operations:

- Read-only property `v1.HasValue` of type `bool` returns true if `v1` is non-null; false if it is null.
- Read-only property `v1.Value` of type `t` returns a copy of the proper value in `v1` if `v1` is non-null and has type `t?`; it throws InvalidOperationException if `v1` is null.
- Standard implicit conversions: The implicit conversion from `v` of type `t` to `t?` gives a non-null value containing a copy of `v`. The implicit coercion from `null` to `t?` gives the null value.
- Standard explicit conversions: `(t)v1` coerces from `t?` to `t` and is equivalent to `v1.Value`.
- Lifted conversions: Whenever there is an implicit (or explicit) coercion from value type `ts` to value type `tt`, there is an implicit (or explicit) coercion from `ts?` to `tt?`.
- Boxing and unboxing conversions: Boxing of a non-null `t?` value gives a boxed `t` value (not a boxed `t?` value), and boxing of a null `t?` value gives a `null` reference. Unboxing of a non-`null` `t` reference gives a non-null `t?` value; and unboxing of a `null` reference gives a null `t?` value.
- Lifted unary operators `+ ++ - -- ! ~` : If argument type and result type of an existing operator are non-nullable value types, then an additional lifted operator is automatically defined for the corresponding nullable types. If the argument is null, the result is null; otherwise the underlying operator is applied to the proper value in the argument.
- Lifted binary operators `+ - * / % & | ^ << >>` : If the argument types and the result type of an existing operator are non-nullable value types, then an additional lifted operator is automatically defined for the corresponding nullable types. If any argument is null, the result is null; otherwise the underlying operator is applied to the proper values in the arguments. The corresponding compound assignment operator, such as `+=`, is automatically defined for nullable types also.
- Equality comparisons: `v1==v2` is true if both `v1` and `v2` are null or both are non-null and contain the same value, false otherwise; `v1!=v2` is the negation of `v1==v2`. Unless other definitions are applicable, `v1!=null` means `v1.HasValue` and `v1==null` means `!v1.HasValue`,
- Ordering comparisons: `v1<v2` and `v1<=v2` and `v1>v2` and `v1>=v2` have type `bool`. A comparison evaluates to false if `v1` or `v2` is `null`; otherwise it compares the proper values in `v1` and `v2`.
- The *null-coalescing* operator `v1 ?? v2` evaluates to the proper value in `v1` if it has one; otherwise evaluates `v2`. It can be used on reference types also, and then is equivalent to `v1!=null ? v1 : v2`.

The nullable type `bool?` has values null, false and true as in the three-valued logic of the SQL query language. The operators `&` and `|` have special definitions that compute a proper truth value when possible:

x&y	null	false	true
null	null	false	null
false	false	false	false
true	null	false	true

x|y	null	false	true
null	null	null	true
false	null	false	true
true	true	true	true

Example 138 Partial Function with Nullable Return Type
Instead of throwing an exception, a computation that fails may return the null value of a nullable type.

```
public static int? Sqrt(int? x) {
  if (x.HasValue && x.Value >= 0)
    return (int)(Math.Sqrt(x.Value));
  else
    return null;
}
...
Console.WriteLine(":{0}:{1}:{2}:", Sqrt(5), Sqrt(null), Sqrt(-5));  // Prints :2:::
```

Example 139 Computing with Nullable Integers
Arithmetic operators such as + and += are automatically lifted to nullable numbers. Ordering comparisons such as > are false if any argument is null; the equality comparisons are not. The null-coalescing operator ?? gets the proper value or provides a default: note that variable `sum` has plain type `int`.

```
int? i1 = 11, i2 = 22, i3 = null, i4 = i1+i2, i5 = i1+i3;
// Values: 11 22 null 33 null
int i6 = (int)i1;                            // Legal
// int i7 = (int)i5;                         // Legal but fails at run-time
// int i8 = i1;                              // Illegal
Object o1 = i1, o3 = i3;                     // Boxing of int? gives boxed int
Console.WriteLine(o1.GetType());             // System.Int32
int? ii1 = (int?)o1, ii3 = (int?)o3;         // Unboxing of boxed int gives int?
Console.WriteLine("[{0}] [{1}]", ii1, ii3); // [11] [null]
int?[] iarr = { i1, i2, i3, i4, i5 };
i2 += i1;
i2 += i4;
Console.WriteLine("i2 = {0}", i2);           // 66 = 11+22+33
int sum = 0;
for (int i=0; i<iarr.Length; i++)
  sum += iarr[i] ?? 0;
Console.WriteLine("sum = {0}", sum);         // 66 = 11+22+33
for (int i=0; i<iarr.Length; i++)
  if (iarr[i] > 11)
    Console.Write("[{0}] ", iarr[i]);        // 22 33
for (int i=0; i<iarr.Length; i++)
  if (iarr[i] != i1)
    Console.Write("[{0}] ", iarr[i]);        // 22 null 33 null
```

Example 140 The Nullable Bool Type
Like other lifted operators, the `bool?` operators `!` and `^` return null when an argument is null. The operators `&` and `|` are special and can produce a non-null result although one argument is null.

```
bool? b1 = null, b2 = false, b3 = true;
bool? b4 = b1^b2, b5 = b1&b2, b6 = b1|b2;                    // null false null
bool? b7 = b1^b3, b8 = b1&b3, b9 = b1|b3;                    // null null true
```

Example 168 Declaration of a Generic Class
An object of generic class LinkedList<T> is a linked list whose elements have type T; it implements interface IMyList<T> from example 173. The generic class declaration has a nested class, an indexer with result type T, methods that take arguments of type T, an `Equals` method that checks that its argument can be cast to IMyList<T>, an explicit conversion from array of T to LinkedList<T>, an overloaded operator (+) for list concatenation, and methods that return enumerators. See also examples 177 and 188.

```
public class LinkedList<T> : IMyList<T> {
  protected int size;               // Number of elements in the list
  protected Node first, last;       // Invariant: first==null iff last==null
  protected class Node {
    public Node prev, next;
    public T item;
    ...
  }
  public LinkedList() { first = last = null; size = 0; }
  public LinkedList(params T[] elems) : this() { ... }
  public int Count { get { return size; } }
  public T this[int i] { get { ... }   set { ... } }
  public void Add(T item) { Insert(size, item); }
  public void Insert(int i, T item) { ... }
  public void RemoveAt(int i) { ... }
  public override bool Equals(Object that) {
    if (that is IMyList<T> && this.size == ((IMyList<T>)that).Count) ...
  }
  public override int GetHashCode() { ... }
  public static explicit operator LinkedList<T>(T[] arr) { ... }
  public static LinkedList<T> operator +(LinkedList<T> xs1, LinkedList<T> xs2) { ... }
  public IEnumerator<T> GetEnumerator() { return new LinkedListEnumerator(this); }
  IEnumerator IEnumerable.GetEnumerator() { return GetEnumerator(); }
}
```

Example 169 Subclass Relations between Generic Classes and Interfaces
The constructed type Point<String> is implicitly convertible to IMovable, and both ColorPoint<String,uint> and ColorPoint<String,Color> are implicitly convertible to Point<String> and IMovable.

```
interface IMovable { void Move(int dx, int dy); }
class Point<Label> : IMovable {
  protected internal int x, y;
  private Label lab;
  public Point(int x, int y, Label lab) { this.x = x; this.y = y; this.lab = lab; }
  public void Move(int dx, int dy) { x += dx; y += dy; }
}
class ColorPoint<Label, Color> : Point<Label> {
  private Color c;
  public ColorPoint(int x, int y, Label lab, Color c) : base(x, y, lab) { ... }
}
```

23.4 Constraints on Type Parameters

A declaration of a generic class `C<T1,...,Tn>` may have type parameter constraints:

> *class-modifiers* `class C<T1,...,Tn>` *subtype-clause parameter-constraints*
> *class-body*

The *parameter-constraints* is a list of *constraint-clauses*, each of this form:

```
where T : c1, c2, ..., cm
```

In the constraint clause, `T` is one of the type parameters `T1,...,Tn` and each c_i is a *constraint*. A given type parameter `Ti` can have at most one constraint clause; that clause may involve one or more constraints $c_1, c_2, \ldots, c_m$. The order of the constraint clauses for `T1,...,Tn` does not matter.

A *constraint* `c` must have one of these four forms:

- `c` is a type expression: a non-sealed class or an interface or a type parameter. The type expression may be a constructed type and involve the type parameters `T1,...,Tn` and type parameters of enclosing generic classes and struct types. An array type, or a type parameter that has a `struct` constraint, cannot be used as a constraint. Nor can the classes Object, System.ValueType, System.Delegate, System.Array, or System.Enum. At most one of the constraints on a type parameter can be a class, and that constraint must appear first. There can be any number of constraints that are interfaces or type parameters.
- `c` is the special `class` constraint. This means that the type parameter must be instantiated by a reference type. The special `class` constraint can appear only first in a constraint list, and cannot appear together with a constraint that is a class or together with the special `struct` constraint.
- `c` is the special `struct` constraint. This means that the type parameter must be instantiated by a non-nullable value type: a simple type such as `int`, or a non-nullable struct type. The special `struct` constraint can appear only first in a constraint list, and cannot appear together with a constraint that is a class or together with the special `class` constraint.
- `c` is the special `new()` constraint. This means that the type parameter must be instantiated by a type that has an argumentless constructor. If a type parameter has a `struct` constraint it is thereby guaranteed to have an argumentless constructor; in that case the `new()` constraint is redundant and illegal. Example 174 illustrates the use of the `new()` constraint.

It is illegal for constraints to be circular as in `class C<T,U> where T : U where U : T { ... }`.

The types `t1,...,tn` used when constructing a type must satisfy the *parameter-constraints*: if type parameter `Ti` is replaced by type `ti` throughout in the *parameter-constraints*, then for each resulting constraint `t : c` where `c` is a type expression, it must hold that `t` is convertible to `c` by an implicit reference conversion or a boxing conversion.

An override method or explicit interface implementation method gets its constraints from the base class method or the interface method, and cannot have explicit constraints of its own. A generic method that implements a method described by a base interface must have the same constraints as the interface method.

Example 170 Type Parameter Constraints
Interface IPrintable describes a method `Print` that will print an object on a TextWriter. The generic PrintableLinkedList<T> can implement IPrintable provided the list elements (of type T) do.

```
class PrintableLinkedList<T> : LinkedList<T>, IPrintable where T : IPrintable {
  public void Print(TextWriter fs) {
    foreach (T x in this)
      x.Print(fs);
  }
}
interface IPrintable { void Print(TextWriter fs); }
```

Example 171 Constraints Involving Type Parameters. Multiple Constraints
The elements of a type T are mutually comparable if any T-value `x` can be compared to any T-value `y` using `x.CompareTo(y)`. This is the case if type T implements IComparable<T>; see section 24.3. The requirement that T implements IComparable<T> is expressible by the constraint `T : IComparable<T>`.

Type ComparablePair<T,U> is a type of ordered pairs of (T,U)-values. For (T,U)-pairs to support comparison, both T and U must support comparison, so constraints are required on both T and U.

```
struct ComparablePair<T,U> : IComparable<ComparablePair<T,U>>
  where T : IComparable<T>
  where U : IComparable<U>
{
  public readonly T Fst;
  public readonly U Snd;
  public int CompareTo(ComparablePair<T,U> that) {    // Lexicographic ordering
    int firstCmp = this.Fst.CompareTo(that.Fst);
    return firstCmp != 0 ? firstCmp : this.Snd.CompareTo(that.Snd);
  }
  ...
}
```

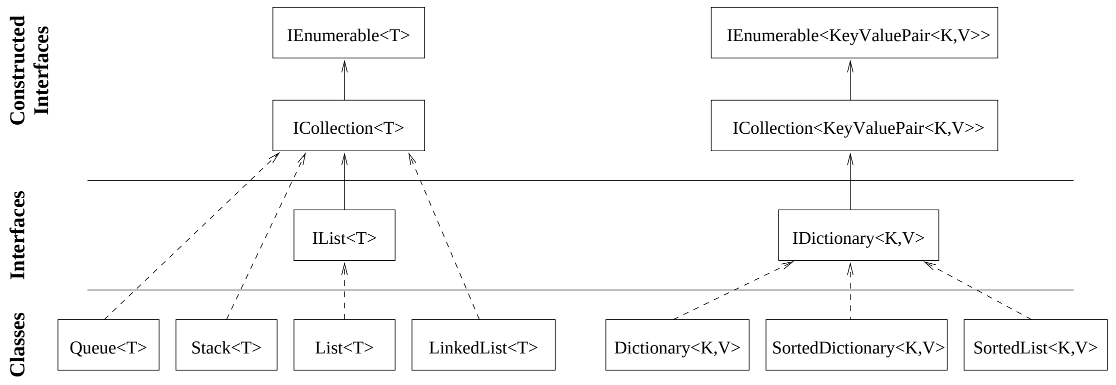
Example 172 The `class` and `struct` Constraints
Without the `class` constraint, type parameter T in `C1` might be instantiated with a value type, and then `null` would not be a legal value of field `f`. Conversely, without the `struct` constraint, type parameter U in `D1` might be instantiated with a reference type or with a nullable value type, and then the nullable type `U?` would be illegal. Thus either class declaration would be rejected by the compiler if its constraint were left out.

```
class C1<T> where T : class {
  T f = null;                         // Legal: T is a reference type
}
class D1<U> where U : struct {
  U? f;                               // Legal: U is a non-nullable value type
}
```

24 Generic Collections: Lists and Dictionaries (C# 2.0)

Namespace System.Collections.Generic provides efficient, convenient and typesafe data structures for representing collections of related data. These data structures include lists, stacks, queues, and dictionaries (also called maps). A list is an ordered sequence where elements can be added and removed at any position; a stack is an ordered sequence where elements can be added and removed only at one end; a queue is an ordered sequence where elements can be added at one end and removed at the other end; and a dictionary associates values with keys. The collection classes are not thread-safe; using the same collection instance from two concurrent threads produces unpredictable results.

The most important generic collection interfaces and classes are related as follows:



24.1 The ICollection<T> Interface

The generic interface ICollection<T> extends IEnumerable<T>, so its elements can be enumerated. In addition, it describes the following members:

- Read-only property `int Count` returns the number of elements in the collection.
- Read-only property `bool IsReadOnly` returns true if the collection is read-only and cannot be modified; otherwise false.
- `void Add(T x)` adds element `x` to the collection. Throws NotSupportedException if the collection is read-only.
- `void Clear()` removes all elements from the collection. Throws NotSupportedException if the collection is read-only.
- `bool Contains(T x)` returns true if element `x` is in the collection; false otherwise.
- `void CopyTo(T[] arr, int i)` copies the collection's members to array `arr`, starting at array index `i`. Throws ArgumentOutOfRangeException if `i<0`, and throws ArgumentException if `i+Count>arr.Length`. Throws InvalidCastException if some collection element is not convertible to the array's element type.
- `bool Remove(T x)` removes an occurrence of element `x` from the collection. Returns true if an element was removed, else false. Throws NotSupportedException if the collection is read-only.

Example 185 Using Generic Collections
The `Print` methods are defined in examples 186 and 187.

```
using System.Collections.Generic;  // IList, IDictionary, List, Dictionary, ...
...
IList<bool> list1 = new List<bool>();
list1.Add(true); list1.Add(false); list1.Add(true); list1.Add(false);
Print(list1);                 // Must print: true false true false
bool b1 = list1[3];           // false
IDictionary<String, int> dict1 = new Dictionary<String, int>();
dict1.Add("Sweden", 46); dict1.Add("Germany", 49); dict1.Add("Japan", 81);
Print(dict1.Keys);            // May print:  Japan Sweden Germany
Print(dict1.Values);          // May print:  81 46 49
int i1 = dict1["Japan"];      // 81
Print(dict1);                 // Print key/value pairs in some order
IDictionary<String, int> dict2 = new SortedDictionary<String, int>();
dict2.Add("Sweden", 46); dict2.Add("Germany", 49); dict2.Add("Japan", 81);
Print(dict2.Keys);            // Must print: Germany Japan Sweden
Print(dict2.Values);          // Must print: 49 81 46
Print(dict2);                 // Print key/value pairs in sorted key order
```

Choosing an Appropriate Collection Class The running time or time complexity of an operation on a collection is usually given in O notation, as a function of the size n of the collection. Thus $O(1)$ means *constant time*, $O(\log n)$ means *logarithmic time* (time at most proportional to the logarithm of n), and $O(n)$ means *linear time* (time at most proportional to n). For accessing, adding, or removing an element, these roughly correspond to *very fast*, *fast*, and *slow*.

In the table, n is the number of elements in the collection and i is an integer index. Thus adding or removing an element of a List is fast only near the end of the list, where $n-i$ is small. The subscript a indicates *amortized complexity*: over a long sequence of operations, the average time per operation is $O(1)$, although any single operation could take time $O(n)$.

Operation	List	Dictionary	SortedDictionary	SortedList
`Add(o)`	$O(1)_a$			
`Insert(i,o)`	$O(n-i)_a$			
`Add(k, v)`		$O(1)_a$	$O(\log n)$	$O(n)$
`Remove(o)`	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$
`RemoveAt(i)`	$O(n-i)$			$O(n-i)$
`Contains(o)`	$O(n)$			
`ContainsKey(o)`		$O(1)$	$O(\log n)$	$O(\log n)$
`ContainsValue(o)`		$O(n)$	$O(n)$	$O(n)$
`IndexOf(o)`	$O(n)$			
`IndexOfKey(k)`				$O(\log n)$
`this[i]`	$O(1)$			
`Keys[i]`				$O(1)$
`Values[i]`				$O(1)$
`this[k] get/set`		$O(1)$	$O(\log n)$	$O(\log n) / O(n)$

24.2 Enumerators and Enumerables

24.2.1 The IEnumerator and IEnumerator<T> Interfaces

An enumerator is an object that enumerates (produces) a stream of elements, such as the elements of a collection. A class or struct type that has a method `GetEnumerator` with return type IEnumerator or IEnumerator<T> can be used in a `foreach` statement (section 13.6.2).

Interface System.Collections.IEnumerator describes these members:

- Read-only property `Object Current` returns the enumerator's current value, or throws InvalidOperationException if the enumerator has not reached the first element or is beyond the last element.
- `bool MoveNext()` advances the enumerator to the next (or first) element, if possible; returns true if it succeeded so that `Current` is valid; false otherwise. Throws InvalidOperationException if the underlying collection has been modified since the enumerator was created.
- `void Reset()` resets the enumerator so that the next call to `MoveNext` will advance it to the first element, if any. Should throw NotSupportedException if not supported.

The generic interface System.Collections.Generic.IEnumerator<T> extends interfaces IEnumerator and IDisposable (section 13.10) and describes these members:

- Read-only property `T Current` returns the enumerator's current value, or throws InvalidOperationException if the enumerator has not reached the first element or is beyond the last element.
- `bool MoveNext()` is just as for IEnumerator above.
- `void Dispose()` is called by the consumer (for instance, a `foreach` statement) when the enumerator is no longer needed. It should release the resources held by the enumerator. Subsequently, `Current` should throw InvalidOperationException and `MoveNext` should return false.

A user type that implements IEnumerator<T> must also implement IEnumerator, using explicit interface member implementation (section 15.3) to declare the `Current` property twice as in example 188.

24.2.2 The IEnumerable and IEnumerable<T> Interfaces

An enumerable type is one that implements interface IEnumerable or IEnumerable<T>. This means that it has a method `GetEnumerator` that can produce an enumerator; see example 168.

Interface System.Collections.IEnumerable describes this method:

- `IEnumerator GetEnumerator()` returns a non-generic enumerator.

The generic interface System.Collections.Generic.IEnumerable<T> describes this method:

- `IEnumerator<T> GetEnumerator()` returns a generic enumerator.

A collection type with element type T implements IEnumerable<T> and IEnumerable. Type Array (section 9.3) implements IEnumerable. For a given type `t`, array type `t[]` also implements IList<`t`> and therefore IEnumerable<`t`> and IEnumerable. A user type that implements IEnumerable<T> must also implement IEnumerable, using explicit interface member implementation (section 15.3) to declare the `GetEnumerator` method twice with different return types, as in example 168.

Example 186 Traversing a Collection
The prototypical traversal of a collection `coll` (which implements IEnumerable<T>) uses a `foreach` statement:

```
public static void Print<T>(ICollection<T> coll) {
  foreach (T x in coll)
    Console.Write("{0} ", x);
  Console.WriteLine();
}
```

Example 187 Traversing a Dictionary
A dictionary `dict` implements IEnumerable<KeyValuePair<K,V>> so its key/value pairs (see section 24.8) can be printed like this:

```
public static void Print<K,V>(IDictionary<K,V> dict) {
  foreach (KeyValuePair<K,V> entry in dict)
    Console.WriteLine("{0} --> {1}", entry.Key, entry.Value);
  Console.WriteLine();
}
```

Example 188 An Enumerator Class for LinkedList<T>
Class LinkedListEnumerator is a member class of and implements an enumerator for class LinkedList<T> from example 168. The `Dispose` method releases any data and list nodes reachable through the current element. The non-generic `Current` property and the `Reset` method are required by interface IEnumerator, which IEnumerator<T> extends. The `yield` statement (section 13.12) provides an alternative simpler way to define enumerators.

```
  private class LinkedListEnumerator : IEnumerator<T> {
    T curr;                    // The enumerator's current element
    bool valid;                // Is the current element valid?
    Node next;                 // Node holding the next element, or null
    public LinkedListEnumerator(LinkedList<T> lst) {
      next = lst.first; valid = false;
    }
    public T Current {
      get { if (valid) return curr; else throw new InvalidOperationException(); }
    }
    public bool MoveNext() {
      if (next != null)  {
        curr = next.item; next = next.next; valid = true;
      } else
        valid = false;
      return valid;
    }
    public void Dispose() { curr = default(T); next = null; valid = false; }
    Object IEnumerator.Current { get { return Current; } }
    void IEnumerator.Reset() { throw new NotSupportedException(); }
  }
```

24.3 Comparables, Equatables, Comparers and EqualityComparers

Some values can be compared for ordering (such as less-than), and some values only for equality. A type may implement interfaces that describe methods for order comparison and equality comparison.

An order comparison method such as `CompareTo` returns a negative number to indicate less-than, zero to indicate equality, and a positive number to indicate greater-than. The method must define a *partial ordering*: it must be reflexive, anti-symmetric, and transitive. Let us define that negative is the opposite sign of positive and vice versa, and that zero is the opposite sign of zero. Then the requirements are:

- `x.CompareTo(x)` must be zero.
- `x.CompareTo(y)` and `y.CompareTo(x)` must be have opposite signs.
- If one of `x.CompareTo(y)` and `y.CompareTo(z)` is zero, then `x.CompareTo(z)` must have the sign of the other one; and if both have the same sign then `x.CompareTo(z)` must have that sign too.

24.3.1 The IComparable, IComparable<T> and IEquatable<T> Interfaces

The non-generic interface System.IComparable describes this method:

- `int CompareTo(Object that)` must return a negative number when the current object (`this`) is less than `that`, zero when they are equal, and a positive number when `this` is greater than `that`.

The generic interface System.IComparable<T> describes this method:

- `int CompareTo(T that)` must return a negative number when the current object (`this`) is less than `that`, zero when they are equal, and a positive number when `this` is greater than `that`.

The generic interface System.IEquatable<T> describes this method:

- `bool Equals(T that)` must return true if the current object (`this`) is equal to `that`, else false.

Type `int` implements IComparable, IComparable<`int`> and IEquatable<`int`>, and similarly for the other numeric types, for the String class (section 7), and for enum types.

24.3.2 The IComparer, IComparer<T>, and IEqualityComparer<T> Interfaces

The non-generic interface System.Collections.IComparer describes this method:

- `int Compare(Object o1, Object o2)` must return a negative number when `o1` is less than `o2`, zero when they are equal, and a positive number when `o1` is greater than `o2`.

The generic interface System.Collections.Generic.IComparer<T> describes these methods:

- `int Compare(T v1, T v2)` must return a negative number when `v1` is less than `v2`, zero when they are equal, and a positive number when `v1` is greater than `v2`.

The generic interface System.Collections.Generic.IEqualityComparer<T> describes these methods:

- `bool Equals(T v1, T v2)` must return true if `v1` is equal to `v2`, else false.
- `int GetHashCode(T v)` must return a hashcode for `v`; see section 5.2.

Example 189 A Class of Comparable Points in Time
A Time object represents the time of day 00:00–23:59. The method call `t1.CompareTo(t2)` returns a negative number if `t1` is before `t2`, a positive number if `t1` is after `t2`, and zero if they are the same time. By defining two overloads of `CompareTo`, the class implements both the non-generic IComparable interface and the constructed interface IComparable<Time>.

```
using System;                                 // IComparable, IComparable<T>, IEquatable<T>
public class Time : IComparable, IComparable<Time>, IEquatable<Time> {
  private readonly int hh, mm;                // 24-hour clock
  public Time(int hh, int mm) { this.hh = hh; this.mm = mm; }
  public int CompareTo(Object that) {         // For IComparable
    return CompareTo((Time)that);
  }
  public int CompareTo(Time that) {           // For IComparable<T>
    return hh != that.hh ? hh - that.hh : mm - that.mm;
  }
  public bool Equals(Time that) {             // For IEquatable<T>
    return hh == that.hh && mm == that.mm;
  }
  public override String ToString() { return String.Format("{0:00}:{1:00}", hh, mm); }
}
```

Example 190 A Comparer for Integer Pairs
Integer pairs are ordered lexicographically by this comparer, which has features from examples 73 and 189.

```
using System.Collections;                     // IComparer
using System.Collections.Generic;             // IComparer<T>, IEqualityComparer<T>

public struct IntPair {
  public readonly int Fst, Snd;
  public IntPair(int fst, int snd) { this.Fst = fst; this.Snd = snd; }
}
public class IntPairComparer : IComparer, IComparer<IntPair>, IEqualityComparer<IntPair>  {
  public int Compare(Object o1, Object o2) {   // For IComparer
    return Compare((IntPair)o1, (IntPair)o2);
  }
  public int Compare(IntPair v1, IntPair v2) { // For IComparer<T>
    return v1.Fst<v2.Fst ? -1 : v1.Fst>v2.Fst ? +1
         : v1.Snd<v2.Snd ? -1 : v1.Snd>v2.Snd ? +1 : 0;
  }
  public bool Equals(IntPair v1, IntPair v2) { // For IEqualityComparer<T>
    return v1.Fst==v2.Fst && v1.Snd==v2.Snd;
  }
  public int GetHashCode(IntPair v) {         // For IEqualityComparer<T>
    return v.Fst ^ v.Snd;
  }
}
```

24.4 The IList<T> Interface

The generic interface IList<T> extends ICollection<T> and describes lists with elements of type T. It has the following members in addition to those of ICollection<T>:

- Read-write indexer `T this[int i]` returns or sets list element number `i`, counting from 0. Throws ArgumentOutOfRangeException if `i<0` or `i>=Count`. Throws NotSupportedException if used to set an element on a read-only list.
- `void Add(T x)` adds element `x` at the end of the list.
- `int IndexOf(T x)` returns the least position whose element equals `x`, if any; otherwise −1.
- `void Insert(int i, T x)` inserts `x` at position `i`. Existing elements at position `i` and higher have their position incremented by one. Throws ArgumentOutOfRangeException if `i<0` or `i>Count`. Throws NotSupportedException if the list is read-only.
- `bool Remove(T x)` removes the first element of the list that equals `x`, if any. Returns true if an element was removed. All elements at higher positions have their position decremented by one. Throws NotSupportedException if the list is read-only.
- `void RemoveAt(int i)` removes the element at index `i`. All elements at higher positions have their position decremented by one. Throws ArgumentOutOfRangeException if `i<0` or `i>=Count`. Throws NotSupportedException if the list is read-only.

Interface IList<T> is implemented by class List<T> which represents a list using an array; see section 24.6. The .Net collection library also includes a class LinkedList<T> of doubly-linked lists, but it does not implement IList<T>, so class LinkedList<T> is not further described here.

24.5 The IDictionary<K,V> Interface

The generic interface IDictionary<K,V> extends ICollection<KeyValuePair<K,V>>, so it can be seen as a collection of key/value pairs (entries), where the keys have type K and the values have type V. Since a dictionary implements also IEnumerable<KeyValuePair<K,V>>, the key/value pairs can be enumerated.

There can be no two entries with the same key, and a key of reference type cannot be `null`.

The interface describes these members in addition to those of ICollection<KeyValuePair<K,V>>:

- Read-only property `ICollection<K> Keys` returns a collection of the keys in the dictionary.
- Read-only property `ICollection<V> Values` returns a collection of the values in the dictionary.
- Read-write indexer `V this[K k]` gets or sets the value at dictionary key `k`. Throws ArgumentException when getting (but not when setting) if key `k` is not in the dictionary. Throws NotSupportedException if used to set an element in a read-only dictionary.
- `void Add(K k, V v)` inserts value `v` at key `k` in the dictionary. Throws ArgumentException if key `k` is already in the dictionary. Throws NotSupportedException if the dictionary is read-only .
- `bool ContainsKey(K k)` returns true if the dictionary contains an entry for key `k`, else false.
- `bool Remove(K k)` removes the entry for key `k`, if any. Returns true if a key was removed.

24.7 The Dictionary<K,V> Class

The generic class Dictionary<K,V> implements IDictionary<K,V> and is used to represent dictionaries or maps with keys of type K and associated values of type V. A dictionary is implemented as a hash table, so the keys should have a good `GetHashCode` method but need not be ordered. In an unordered dictionary any key can be looked up, inserted, updated, or deleted in amortized constant time.

Objects used as dictionary keys should be treated as immutable, or else subtle errors may be encountered. If an object is used as key in a dictionary, and the object is subsequently modified so that its hashcode changes, then the key and its entry may be lost in the dictionary.

Class Dictionary<K,V> has the members described by IDictionary<K,V> as well as these:

- Constructor `Dictionary()` creates an empty dictionary.
- Constructor `Dictionary(int capacity)` creates an empty dictionary with given initial capacity.
- Constructor `Dictionary(int capacity, IEqualityComparer<K> cmp)` creates an empty dictionary with the given initial capacity and the given equality comparer.
- Constructor `Dictionary(IDictionary<K,V> dict)` creates a new dictionary that contains `dict`'s key/value pairs.
- Constructor `Dictionary(IDictionary<K,V> dict, IEqualityComparer<K> cmp)` creates a new dictionary from `dict`'s key/value pairs, using the given equality comparer.
- `bool ContainsValue(V v)` returns true if the dictionary contains an entry with value `v`. In contrast to `ContainsKey(k)`, this is slow: it requires a linear search of all key/value pairs.
- `bool TryGetValue(K k, out V v)` binds `v` to the value associated with key `k` and returns true if the dictionary contains an entry for `k`; otherwise binds `v` to `default(V)` and returns false.

24.8 The KeyValuePair<K,V> Struct Type

A struct of generic type KeyValuePair<K,V> is used to hold a key/value pair, or entry, from a dictionary (sections 24.5 and 24.7). See example 187. The KeyValuePair<K,V> struct type has the following members:

- Constructor `KeyValuePair(K k, V v)` creates a pair of key `k` and value `v`.
- Read-only property `K Key` returns the key in the key/value pair.
- Read-only property `V Value` returns the value in the key/value pair.

24.9 The SortedDictionary<K,V> and SortedList<K,V> Classes

The generic classes SortedDictionary<K,V> and SortedList<K,V> implement IDictionary<K,V> and represent a dictionary with ordered keys of type K and associated values of type V. The former dictionary class is implemented by a binary tree and the latter by a sorted array. The key type K must implement IComparable<K> or IComparable (section 24.3) or else an IComparer<K> or an IComparer must be provided when the dictionary is created. This determines the key order and the enumeration order.

These classes have the same methods as Dictionary<K,V>, but their constructors take no `capacity` argument and take an IComparer<K> argument instead of an IEqualityComparer<K> argument.

30 References

- The C# 2.0 programming language, including generic types and methods, iterators, anonymous methods, partial types, nullable types, and so on, has been standardized by Ecma International in June 2005. Download the *C# Language Specification*, adopted as Ecma Standard ECMA-334, 3rd edition, from <http://www.ecma-international.org/publications/standards/Ecma-334.htm>
- Anders Hejlsberg, Scott Wiltamuth and Peter Golde: *The C# Programming Language*, Addison-Wesley November 2003, contains a version of the C# Language Specification, including a description of most C# 2.0 features.
- The Microsoft Windows .Net Framework Software Development Kit (SDK), including a C# compiler and run-time system, is available from <http://msdn.microsoft.com/netframework/>
- The Mono implementation of C# and run-time system is available for Microsoft Windows, Linux, MacOS X, Solaris and other platforms from <http://www.mono-project.com/>
- The Microsoft .Net Framework class reference documentation is included with the .Net Framework SDK and is also available online at <http://msdn.microsoft.com/library/>
- The design and implementation of generics in C# and .Net is described in Don Syme and Andrew Kennedy: Design and Implementation of Generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation (PLDI)*, Snowbird, Utah, 2001.
  Download from <http://research.microsoft.com/~dsyme/papers/generics.pdf>
- A comprehensive library of collection classes for C# is provided by the library C5, available at <http://www.itu.dk/research/c5/> under a liberal license.
- The Unicode character encoding (<http://www.unicode.org/>) corresponds to part of the Universal Character Set (UCS), which is international standard ISO 10646-1:2000. The UTF-8 is a variable-length encoding of UCS, in which 7-bit ASCII characters are encoded as themselves. It is described in Annex R of the above-mentioned ISO standard.
- Floating-point arithmetics is described in the ANSI/IEEE Standard for Binary Floating-Point Arithmetic (IEEE Std 754-1985).
- Advice on writing high-performance C# programs for the .Net platform can be found in Gregor Noriskin: *Writing High-Performance Managed Applications: A Primer*, June 2003; and in Jan Gray: *Writing Faster Managed Code: Know What Things Cost*, June 2003. Both are available from the MSDN Developer Library <http://msdn.microsoft.com/library/>.