|  |  |  |
| --- | --- | --- |
| **Category** | | **Description** |
| Value types | Simple types | Signed integral: sbyte, short, int, long |
| Unsigned integral: byte, ushort, uint, ulong |
| Unicode characters: char |
| IEEE floating point: float, double |
| High-precision decimal: decimal |
| Boolean: bool |
| Enum types | User-defined types of the form enum E {...} |
| Struct types | User-defined types of the form struct S {...} |
| Reference types | Class types | Ultimate base class of all other types: object |
| Unicode strings: string |
| User-defined types of the form class C {...} |
| Interface types | User-defined types of the form interface I {...} |
| Array types | Single- and multi-dimensional, for example, int[] and int[,] |
| Delegate types | User-defined types of the form delegate T D(...) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Bits** | **Type** | **Range/Precision** |
| Signed integral | 8 | sbyte | –128...127 |
| 16 | short | –32,768...32,767 |
| 32 | int | –2,147,483,648...2,147,483,647 |
| 64 | long | –9,223,372,036,854,775,808...9,223,372,036,854,775,807 |
| Unsigned integral | 8 | byte | 0...255 |
| 16 | ushort | 0...65,535 |
| 32 | uint | 0...4,294,967,295 |
| 64 | ulong | 0...18,446,744,073,709,551,615 |
| Floating point | 32 | float | 1.5 × 10−45 to 3.4 × 1038, 7-digit precision |
| 64 | double | 5.0 × 10−324 to 1.7 × 10308, 15-digit precision |
| Decimal | 128 | decimal | 1.0 × 10−28 to 7.9 × 1028, 28-digit precision |

|  |  |
| --- | --- |
| **Member** | **Description** |
| Constants | The constant values associated with the class |
| Fields | The variables of the class |
| Methods | The computations and actions that can be performed by the class |
| Properties | The actions associated with reading and writing named properties of the class |
| Indexers | The actions associated with indexing instances of the class like an array |
| Events | The notifications that can be generated by the class |
| Operators | The conversions and expression operators supported by the class |
| Constructors | The actions required to initialize instances of the class or the class itself |
| Destructors | The actions to perform before instances of the class are permanently discarded |
| Types | The nested types declared by the class |

|  |  |
| --- | --- |
| **Accessibility** | **Meaning** |
| Public | Access not limited |
| Protected | Access limited to this class and classes derived from this class |
| Internal | Access limited to this program |
| protected internal | Access limited to this program and classes derived from this class |
| Private | Access limited to this class |

**Generic Classes**

class List<T> {}

class C {}

class C<V> {}

struct C<U,V> {} // Error, C with two type parameters defined twice

class C<A,B> {} // Error, C with two type parameters defined twice

class A<T> // instance type: A<T>  
{  
 class B {} // instance type: A<T>.B

class C<U> {} // instance type: A<T>.C<U>  
}

class D {} // instance type: D

class Extend<V>: V {} // Error, type parameter used as base class

class C<U,V> {}

interface I1<V> {}

class D: C<string,int>, I1<string> {}

class E<T>: C<int,T>, I1<T> {}

class C<U,V>   
{  
 public virtual void M1(U x, List<V> y) {...}  
}

interface I1<V>   
{  
 V M2(V);  
}

class D: C<string,int>, I1<string>   
{  
 public override void M1(string x, List<int> y) {...}

public string M2(string x) {...}  
}

class C<V>  
{  
 public V f1;  
 public C<V> f2 = null;

public C(V x) {  
 this.f1 = x;  
 this.f2 = this;  
 }  
}

C<int> x1 = new C<int>(1);  
 Console.WriteLine(x1.f1); // Prints 1

C<double> x2 = new C<double>(3.1415);  
 Console.WriteLine(x2.f1); // Prints 3.1415

class C<V>  
{  
 static int count = 0;

public C() {  
 count++;  
 }

public static int Count {  
 get { return count; }  
 }  
}

C<int> x1 = new C<int>();  
 Console.WriteLine(C<int>.Count); // Prints 1

C<double> x2 = new C<double>();  
 Console.WriteLine(C<int>.Count); // Prints 1

C<int> x3 = new C<int>();  
 Console.WriteLine(C<int>.Count); // Prints 2

class Gen<T> where T: struct  
{  
 **static Gen**() {  
 if (!typeof(T).IsEnum) {  
 throw new ArgumentException("T must be an enum");  
 }  
 }  
}

Type parameters can be used in the type of a parameter array. For example, given the declaration

class C<V>  
{  
 static void F(int x, int y, params V[] args);  
}

the following invocations of the expanded form of the method:

C<int>.F(10, 20);  
C<object>.F(10, 20, 30, 40);  
C<string>.F(10, 20, "hello", "goodbye");

correspond exactly to:

C<int>.F(10, 20, new int[] {});  
C<object>.F(10, 20, new object[] {30, 40});  
C<string>.F(10, 20, new string[] {"hello", "goodbye"} );

class Outer<T>  
{  
 class Inner<U>

{  
 public static void F(T t, U u) {...}  
 }  
}

class Outer<T>  
{  
 class Inner<T> // Valid, hides Outer’s T  
 {  
 public T t; // Refers to Inner’s T  
 }  
}

public class Stack<T>  
{  
 T[] items;  
 int count;

public void Push(T item) {...}

public T Pop() {...}  
}

Stack<int> stack = new Stack<int>();  
stack.Push(3);  
int x = stack.Pop();

Stack<Customer> stack = new Stack<Customer>();  
stack.Push(new Customer());  
Customer c = stack.Pop();  
stack.Push(3); // Type mismatch error  
int x = stack.Pop(); // Type mismatch error

public class Dictionary<K,V>  
{  
 public void Add(K key, V value) {...}

public V this[K key] {...}  
}

Dictionary<string,Customer> dict = new Dictionary<string,Customer>();  
dict.Add("Peter", new Customer());  
Customer c = dict["Peter"];

public class Dictionary<K,V>  
{  
 public void Add(K key, V value)  
 {  
 ...

if (key.CompareTo(x) < 0) {...} // Error, no CompareTo method  
 ...  
 }  
}

public class Dictionary<K,V>  
{  
 public void Add(K key, V value)  
 {  
 ...

if (((IComparable)key).CompareTo(x) < 0) {...} // works but dynamic type check at run-time, which adds overhead, defers error reporting to run-time  
 ...  
 }  
}

public class Dictionary<K,V> where K: IComparable  
{  
 public void Add(K key, V value)  
 {  
 ...

if (key.CompareTo(x) < 0) {...}  
 ...  
 }  
}

public class EntityTable<K,E>  
 where K: IComparable<K>, IPersistable  
 where E: Entity, new()  
{  
 public void Add(K key, E entity)  
 {  
 ...

if (key.CompareTo(x) < 0) {...}  
 ...  
 }  
}

The constructor constraint, new(), in the example above ensures that a type used as a type argument for E has a public, parameterless constructor, and it permits the generic class to use new E() to create instances of that type.

interface I<T>  
{  
 void F();  
}

class X<U,V>: I<U>, I<V> // Error: I<U> and I<V> conflict  
{  
 void I<U>.F() {...}  
 void I<V>.F() {...}  
}

I<int> x = new X<int,int>();  
x.F();

interface IList<T>  
{  
 T[] GetElements();  
}

interface IDictionary<K,V>  
{  
 V this[K key];

void Add(K key, V value);  
}

class List<T>: IList<T>, IDictionary<int,T>  
{  
 T[] IList<T>.GetElements() {...}

T IDictionary<int,T>.this[int index] {...}

void IDictionary<int,T>.Add(int index, T value) {...}  
}

delegate bool Predicate<T>(T value);

class X  
{  
 static bool F(int i) {...}

static bool G(string s) {...}

static void Main() {  
 Predicate<int> p1 = F;  
 Predicate<string> p2 = G;  
 }  
}

Predicate<int> p1 = new Predicate<int>(F);  
 Predicate<string> p2 = new Predicate<string>(G);

namespace Widgets  
{  
 class Queue {...}  
 class Queue<ElementType> {...}  
}

namespace MyApplication  
{  
 using Widgets;

class X  
 {  
 Queue q1; // Non-generic Widgets.Queue  
 Queue<int> q2; // Generic Widgets.Queue  
 }  
}

class B<U>  
{  
 public U F(long index) {...}  
}

class D<T>: B<T[]>  
{  
 public T G(string s) {...}  
}

**Generic methods**

void PushMultiple<T>(Stack<T> stack, params T[] values) {  
 foreach (T value in values) stack.Push(value);  
}

Stack<int> stack = new Stack<int>();  
PushMultiple<int>(stack, 1, 2, 3, 4);

type inferencing

Stack<int> stack = new Stack<int>();  
PushMultiple(stack, 1, 2, 3, 4);

.

.

**Anonymous methods**

class InputForm: Form  
{  
 ListBox listBox;  
 TextBox textBox;  
 Button addButton;

public MyForm() {  
 listBox = new ListBox(...);  
 textBox = new TextBox(...);  
 addButton = new Button(...);

addButton.Click += new EventHandler(AddClick);  
 }

void AddClick(object sender, EventArgs e) {  
 listBox.Items.Add(textBox.Text);  
 }  
}

class InputForm: Form  
{  
 ListBox listBox;  
 TextBox textBox;  
 Button addButton;

public MyForm() {  
 listBox = new ListBox(...);  
 textBox = new TextBox(...);  
 addButton = new Button(...);

addButton.Click += delegate(object sender, EventArgs e) {  
 listBox.Items.Add(textBox.Text);  
 // MessageBox.Show(((Button)sender).Text);  
};

// OR

addButton.Click += delegate {  
 listBox.Items.Add(textBox.Text);

// MessageBox.Show(((Button)sender).Text);  
};

}  
}

delegate double Function(double x);

static double SquareIt(double d1, Function f) {  
 return f(d1);  
 }

static void Main() {

double square = SquareIt(1.2, delegate(double x) { return x \* x; });

}

addButton.Click += new EventHandler(AddClick);

can instead be written

addButton.Click += AddClick;

Apply(a, new Function(Math.Sin));

can instead be written

Apply(a, Math.Sin);

.

.

Iterator

----------------

// multiplication table

static IEnumerable<int> FromTo(int from, int to) {

while (from <= to) yield return from++;

}

static void Main() {

IEnumerable<int> e = FromTo(1, 10);

foreach (int x in e) { // x from 1 to 10

foreach (int y in e) { // y from 1 to 10

Console.Write("{0,3} ", x \* y); // multiplication table

}

Console.WriteLine();

}

}

----------------

public class Stack<T>: IEnumerable<T>

{

T[] items;

int count;

public void Push(T data) {...}

public T Pop() {...}

public IEnumerator<T> GetEnumerator() {

for (int i = count – 1; i >= 0; --i) {

yield return items[i];

}

}

public IEnumerable<T> TopToBottom {

get {

return this;

}

}

public IEnumerable<T> BottomToTop {

get {

for (int i = 0; i < count; i++) {

yield return items[i];

}

}

}

}

static void PrintAll(IEnumerable<int> collection) {

foreach (int i in collection) {

Console.Write("{0} ", i);

}

Console.WriteLine();

}

Stack<int> stack = new Stack<int>();

for (int i = 0; i < 10; i++) { stack.Push(i); }

foreach (int i in stack) { Console.Write("{0} ", i); } Console.WriteLine();

foreach (int i in stack.TopToBottom) { Console.Write("{0} ", i); } Console.WriteLine();

foreach (int i in stack.BottomToTop) { Console.Write("{0} ", i); } Console.WriteLine();

static IEnumerable<int> FromToBy(int from, int to, int by) {

for (int i = from; i <= to; i += by) {

yield return i;

}

}

PrintAll(FromToBy(10, 20, 2));

----------------

Partial types allow classes, structs, and interfaces to be broken into multiple pieces stored in different source files for easier development and maintenance. Additionally, partial types allow separation of machine-generated and user-written parts of types so that it is easier to augment code generated by a tool.

public partial class Customer  
{  
 private int id;  
 private string name;  
 private string address;  
 private List<Order> orders;

public Customer() {  
 ...  
 }  
}

public partial class Customer  
{  
 public void SubmitOrder(Order order) {  
 orders.Add(order);  
 }

public bool HasOutstandingOrders() {  
 return orders.Count > 0;  
 }  
}

## Nullable types

int? x = 123;  
int? y = null;

if (x.HasValue) Console.WriteLine(x.Value);   
if (y.HasValue) Console.WriteLine(y.Value);

int i = 123;  
int? x = i; // int --> int?  
double? y = x; // int? --> double?  
int? z = (int?)y; // double? --> int?  
int j = (int)z; // int? --> int

int? x = GetNullableInt();  
int? y = GetNullableInt();  
int? z = x + y;

same as

int? z = x.HasValue && y.HasValue ? x.Value + y.Value : (int?)null;

int? x = GetNullableInt();  
int? y = x + 1;

If x is null, y is assigned null. Otherwise, y is assigned the value of x plus one.

null coalescing operator, ??

string s = GetStringValue();  
Console.WriteLine(s ?? "Unspecified");

int? x = GetNullableInt();  
int? y = GetNullableInt();  
int? z = x ?? y;  
int i = z ?? -1;

class A  
{  
 public string Text {  
 get { return "hello"; }  
 set { }  
 }

public int Count {  
 get { return 5; }  
 set { }  
 }  
}

class B: A  
{  
 private string text = "goodbye";   
 private int count = 0;

new public string Text {  
 get { return text; }  
 protected set { text = value; }  
 }

new protected int Count {   
 get { return count; }  
 set { count = value; }  
 }  
}

class M  
{  
 static void Main() {  
 B b = new B();  
 b.Count = 12; // Calls A.Count set accessor  
 int i = b.Count; // Calls A.Count get accessor  
 b.Text = "howdy"; // Error, B.Text set accessor not accessible  
 string s = b.Text; // Calls B.Text get accessor  
 }  
}

using SIO = System.IO;  
using MIO = MyLibrary.IO;

class Program  
{  
 static void Main() {  
 SIO.Stream s = new MIO.EmptyStream();  
 ...  
 }  
}

using SIO = System.IO;  
using MIO = MyLibrary.IO;

class Program  
{  
 static void Main() {  
 SIO::Stream s = new MIO::EmptyStream();  
 ...  
 }  
}

global::System.IO.Stream s = new global::MyLibrary.IO.EmptyStream();

namespace N  
{  
 public class A {}

public class B {}  
}

namespace N  
{  
 using A = System.IO;

class X  
 {  
 A.Stream s1; // Error, A is ambiguous

A::Stream s2; // Ok  
 }  
}

Consider the following two assemblies:

Assembly a1.dll:

namespace N  
{  
 public class A {}

public class B {}  
}

Assembly a2.dll:

namespace N  
{  
 public class B {}

public class C {}  
}

and the following program:

class Test  
{  
 N.A a; // Ok  
 N.B b; // Error  
 N.C c; // Ok  
}

csc /r:a1.dll /r:a2.dll test.cs

extern alias X;  
extern alias Y;

class Test  
{  
 X::N.A a;  
 X::N.B b1;  
 Y::N.B b2;  
 Y::N.C c;  
}

csc /r:X=a1.dll /r:Y=a2.dll test.cs

using System;

class Program  
{  
 [Obsolete]  
 static void Foo() {}

static void Main() {  
#pragma warning disable 612  
 Foo();  
#pragma warning restore 612  
 }  
}

#define DEBUG

using System;  
using System.Diagnostics;

[Conditional("DEBUG")]  
public class TestAttribute : Attribute {}

[Test]  
class C {}

File test.cs:

using System;  
using System.Diagnostics;

[Conditional(“DEBUG”)]  
public class TestAttribute : Attribute {}

File class1.cs:

#define DEBUG

[Test] // TestAttribute is included  
class Class1 {}

File class2.cs:

#undef DEBUG

[Test] // TestAttribute is excluded  
class Class2 {}

### Type parameters

public class Pair<TFirst,TSecond>  
{  
 public TFirst First;

public TSecond Second;  
}

Pair<int,string> pair = new Pair<int,string> { First = 1, Second = “two” };  
int i = pair.First; // TFirst is int  
string s = pair.Second; // TSecond is string

public class Point  
{  
 int x, y;

public int X { get { return x; } set { x = value; } }  
 public int Y { get { return y; } set { y = value; } }  
}

Point a = new Point { X = 0, Y = 1 };

public class Rectangle  
{  
 Point p1, p2;

public Point P1 { get { return p1; } set { p1 = value; } }  
 public Point P2 { get { return p2; } set { p2 = value; } }  
}

Rectangle r = new Rectangle {  
 P1 = new Point { X = 0, Y = 1 },  
 P2 = new Point { X = 2, Y = 3 }  
};

public class Rectangle  
{  
 Point p1 = new Point();  
 Point p2 = new Point();

public Point P1 { get { return p1; } }  
 public Point P2 { get { return p2; } }  
}

Rectangle r = new Rectangle {  
 P1 = { X = 0, Y = 1 },  
 P2 = { X = 2, Y = 3 }  
};

List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

public class Contact  
{  
 string name;  
 List<string> phoneNumbers = new List<string>();

public string Name { get { return name; } set { name = value; } }

public List<string> PhoneNumbers { get { return phoneNumbers; } }  
}

var contacts = new List<Contact> {  
 new Contact {  
 Name = "Chris Smith",  
 PhoneNumbers = { "206-555-0101", "425-882-8080" }  
 },  
 new Contact {  
 Name = "Bob Harris",  
 PhoneNumbers = { "650-555-0199" }  
 }  
};

x => x + 1 // Implicitly typed, expression body

x => { return x + 1; } // Implicitly typed, statement body

(int x) => x + 1 // Explicitly typed, expression body

(int x) => { return x + 1; } // Explicitly typed, statement body

(x, y) => x \* y // Multiple parameters

() => Console.WriteLine() // No parameters

delegate (int x) { return x + 1; } // Anonymous method expression

delegate { return 1 + 1; } // Parameter list omitted

from c in customers  
group c by c.Country into g  
select new { Country = g.Key, CustCount = g.Count() }

same as

customers.  
GroupBy(c => c.Country).  
Select(g => new { Country = g.Key, CustCount = g.Count() })

from Customer c in customers  
where c.City == "London"  
select c

same as

customers.  
Cast<Customer>().  
Where(c => c.City == "London")

from c in customers  
select c

Is translated into

customers.Select(c => c)

from c in customers  
from o in c.Orders  
select new { c.Name, o.OrderID, o.Total }

is translated into

customers.  
SelectMany(c => c.Orders,  
 (c,o) => new { c.Name, o.OrderID, o.Total }  
)

from c in customers  
from o in c.Orders  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

is translated into

from \* in customers.  
 SelectMany(c => c.Orders, (c,o) => new { c, o })  
orderby o.Total descending  
select new { c.Name, o.OrderID, o.Total }

to

customers.  
SelectMany(c => c.Orders, (c,o) => new { c, o }).  
OrderByDescending(x => x.o.Total).  
Select(x => new { x.c.Name, x.o.OrderID, x.o.Total })

partial class Customer  
{  
 string name;

public string Name {

get { return name; }

set {  
 OnNameChanging(value);  
 name = value;  
 OnNameChanged();  
 }

}

partial void OnNameChanging(string newName);

partial void OnNameChanged();  
}

becomes

class Customer  
{  
 string name;

public string Name {

get { return name; }

set { name = value; }  
 }  
}

partial class Customer  
{  
 string name;

public string Name {

get { return name; }

set {  
 OnNameChanging(value);  
 name = value;  
 OnNameChanged();  
 }

}

partial void OnNameChanging(string newName);

partial void OnNameChanged();  
}

partial class Customer  
{  
 partial void OnNameChanging(string newName)  
 {  
 Console.WriteLine(“Changing “ + name + “ to “ + newName);  
 }

partial void OnNameChanged()  
 {  
 Console.WriteLine(“Changed to “ + name);  
 }  
}

becomes

class Customer  
{  
 string name;

public string Name {

get { return name; }

set {  
 OnNameChanging(value);  
 name = value;  
 OnNameChanged();  
 }

}

void OnNameChanging(string newName)  
 {  
 Console.WriteLine(“Changing “ + name + “ to “ + newName);  
 }

void OnNameChanged()  
 {  
 Console.WriteLine(“Changed to “ + name);  
 }  
}

namespace N  
{  
 using List = System.Collections.ArrayList;

partial class A  
 {  
 List x; // x has type System.Collections.ArrayList  
 }  
}

namespace N  
{  
 using List = Widgets.LinkedList;

partial class A  
 {  
 List y; // y has type Widgets.LinkedList  
 }  
}

public static class Extensions  
{  
 public static int ToInt32(this string s) {  
 return Int32.Parse(s);  
 }

public static T[] Slice<T>(this T[] source, int index, int count) {  
 if (index < 0 || count < 0 || source.Length – index < count)  
 throw new ArgumentException();  
 T[] result = new T[count];  
 Array.Copy(source, index, result, 0, count);  
 return result;  
 }  
}

string[] strings = { "1", "22", "333", "4444" };  
 foreach (string s in strings.Slice(1, 2)) {  
 Console.WriteLine(s.ToInt32());  
 }

same as

string[] strings = { "1", "22", "333", "4444" };  
foreach (string s in Extensions.Slice(strings, 1, 2)) {  
 Console.WriteLine(Extensions.ToInt32(s));  
}

public class Point {  
 public int X { get; set; } // automatically implemented  
 public int Y { get; set; } // automatically implemented  
}

is equivalent to the following declaration:

public class Point {  
 private int x;  
 private int y;  
 public int X { get { return x; } set { x = value; } }  
 public int Y { get { return y; } set { y = value; } }  
}

public class ReadOnlyPoint {  
 public int X { get; private set; }  
 public int Y { get; private set; }  
 public ReadOnlyPoint(int x, int y) { X = x; Y = y; }  
}

class A  
{  
 public string Text {  
 get { return "hello"; }  
 set { }  
 }

public int Count {  
 get { return 5; }  
 set { }  
 }  
}

class B: A  
{  
 private string text = "goodbye";   
 private int count = 0;

new public string Text {  
 get { return text; }  
 protected set { text = value; }  
 }

new protected int Count {   
 get { return count; }  
 set { count = value; }  
 }  
}

class M  
{  
 static void Main() {  
 B b = new B();  
 b.Count = 12; // Calls A.Count set accessor  
 int i = b.Count; // Calls A.Count get accessor  
 b.Text = "howdy"; // Error, B.Text set accessor not accessible  
 string s = b.Text; // Calls B.Text get accessor  
 }  
}

object o = “object”  
dynamic d = “dynamic”;

string s1 = o; // Fails at compile-time – no conversion exists  
string s2 = d; // Compiles and succeeds at run-time  
int i = d; // Compiles but fails at run-time – no conversion exists

class C  
{  
 int i;

public C(int i) { this.i = i; }

public static explicit operator C(string s)   
 {  
 return new C(int.Parse(s));  
 }  
}

object o = "1";  
dynamic d = "2";

var c1 = (C)o; // Compiles, but explicit reference conversion fails  
var c2 = (C)d; // Compiles and user defined conversion succeeds

object o = 5;  
dynamic d = 5;

Console.WriteLine(5); // static binding to Console.WriteLine(int)  
Console.WriteLine(o); // static binding to Console.WriteLine(object)  
Console.WriteLine(d); // dynamic binding to Console.WriteLine(int)

class Test  
{  
 static void F(int x, int y = -1, int z = -2) {  
 System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);  
 }

static void Main() {  
 int i = 0;  
 F(i++, i++, i++);  
 F(z: i++, x: i++);  
 }  
}

x = 0, y = 1, z = 2  
x = 4, y = -1, z = 3

class Test  
{  
 static void F(ref object x) {...}

static void Main() {  
 object[] a = new object[10];  
 object[] b = new string[10];  
 F(ref a[0]); // Ok  
 F(ref b[1]); // ArrayTypeMismatchException  
 }  
}

void F(int x, int y, params object[] args);

F(10, 20);  
F(10, 20, 30, 40);  
F(10, 20, 1, "hello", 3.0);

correspond exactly to

F(10, 20, new object[] {});  
F(10, 20, new object[] {30, 40});  
F(10, 20, new object[] {1, "hello", 3.0});