.NET

Wstęp do C#

Agenda

- Typy danych / operatory
- Tablice
- Klasy i obiekty
- Atrybuty / indeksatory
- Delegaty / eventy
- Generics

The “Hello World” in C#

```
using System;
class Hello
{
  static void Main(string[] args)
  {
    Console.WriteLine("Hello world!");
    for (int i = 0; i < args.Length; i++)
    {
      Console.WriteLine("args[{0}] = {1}",
                        i, args[i]);
    }
  }
}
```

The “Hello World” in VB.NET

```
Imports System
Module Hello
  Sub Main(ByVal CmdArgs() As String)
    Dim I As Integer
    Console.WriteLine("Hello world!")
    For I = 0 To CmdArgs.GetUpperBound(0)
      Console.WriteLine("CmdArgs[{0}] = {1}", _
                                 I, CmdArgs(I))
    Next I
  End Sub
End Module
```

Pascal

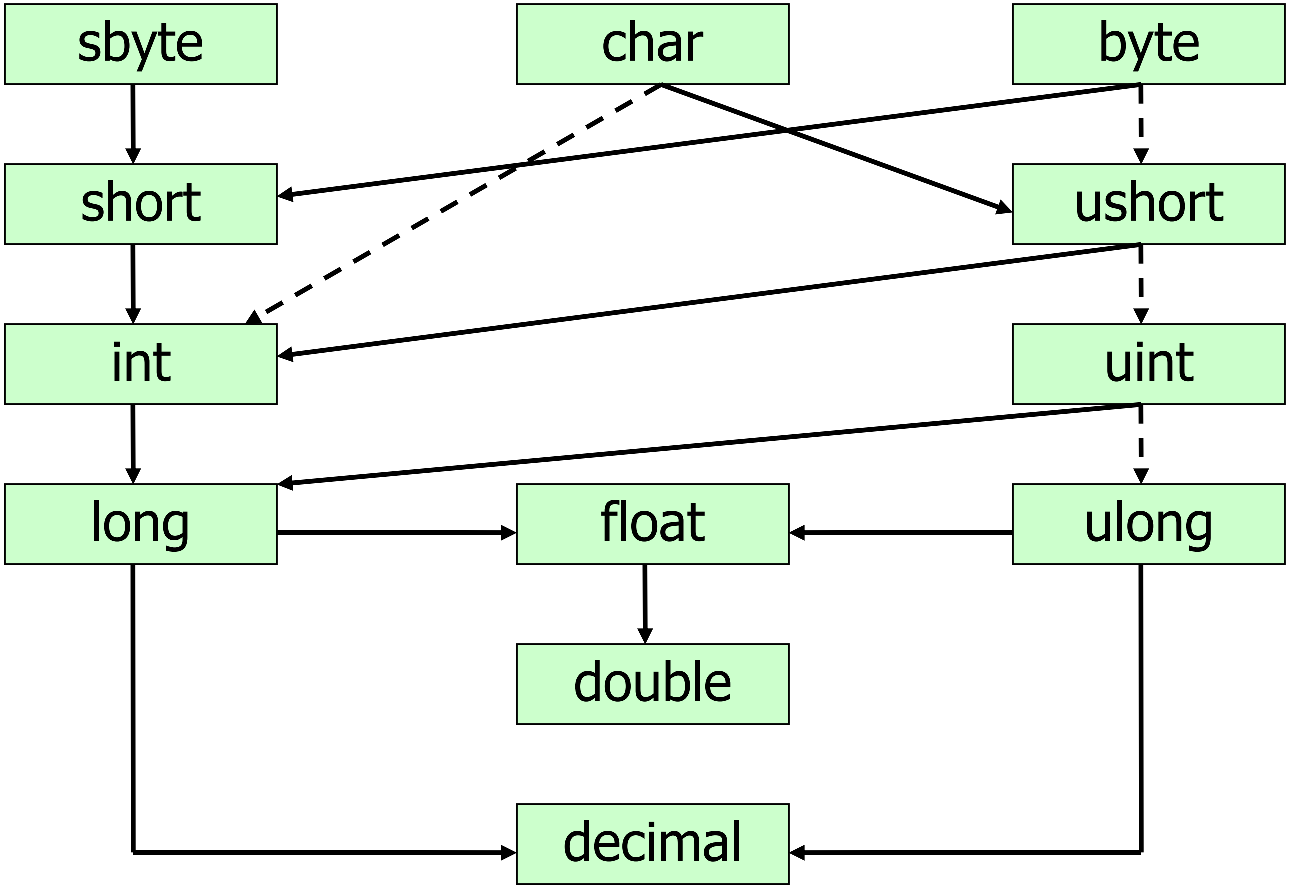
- Module Hello;

- Import Cpmain, Console;
- Begin
  - Console.WriteString(“Hello”);
  - Console.WriteLn;
- End Hello

C# Data Types

- **object** – reference to object
- **string** – sequence of Unicode characters
- **byte**, **sbyte** – 8-bit integer value
- **short**, **ushort** – 16-bit integer value
- **int**, **uint** – 32-bit integer value
- **long**, **ulong** – 64-bit integer value
- **float**, **double** – floating-point number
- **bool** – logical value (*true* or *false*)
- **char** – Unicode character
- **decimal** – fixed-point decimal number

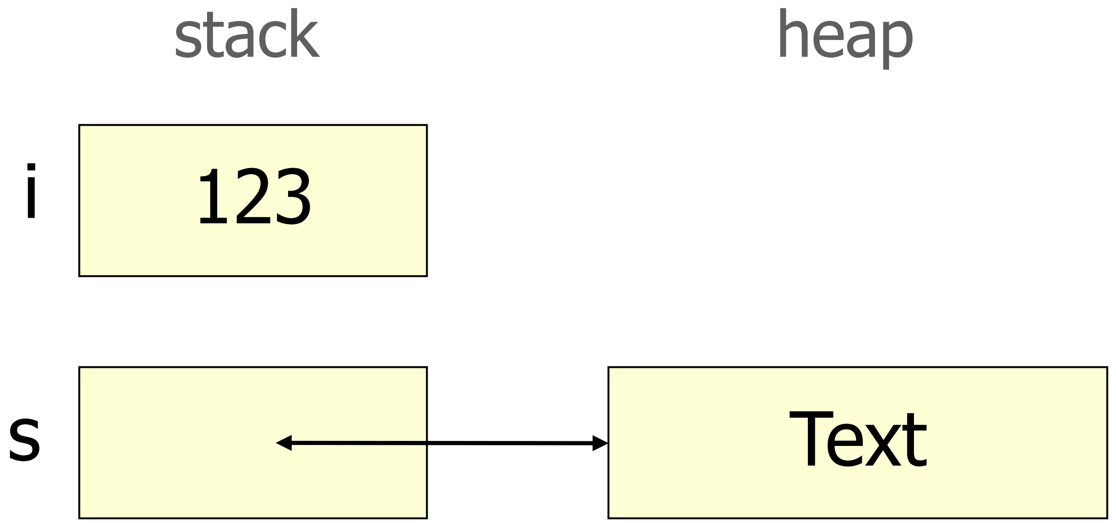
Automatic Type Conversions



sbyte
char
byte
short
ushort
int
uint
long
float
ulong
double
decimal

Value Types and Reference Types

```
int i = 123;
string s = "Text";
```



stack

heap

i

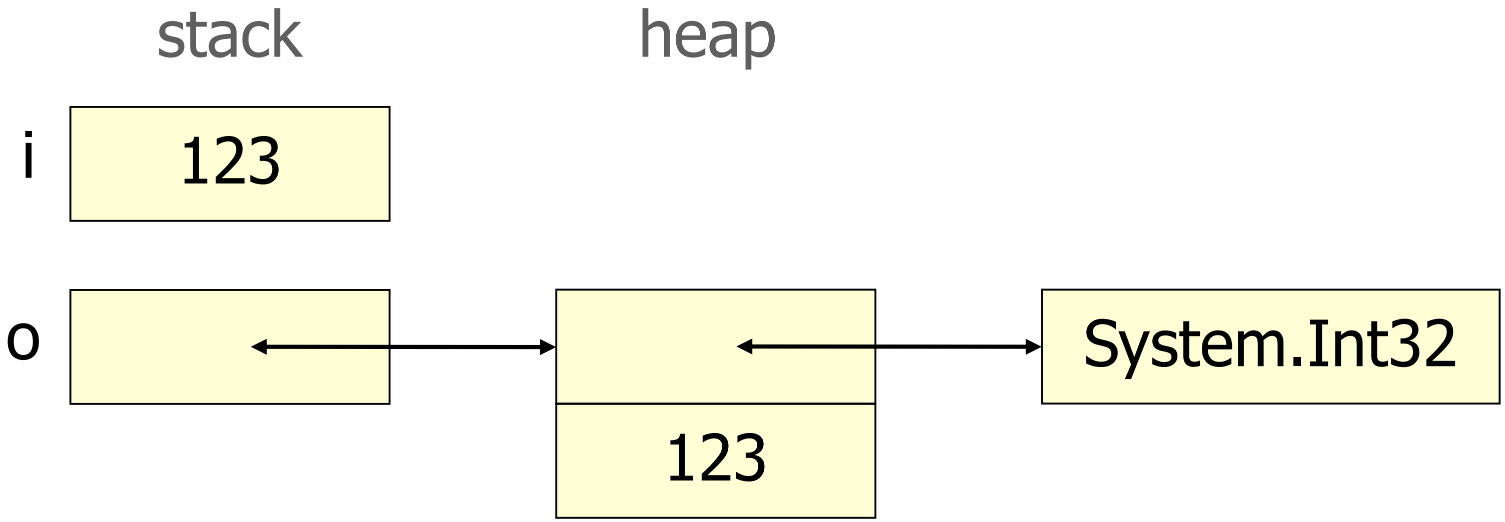
123

s

Text

Boxing

```
int i = 123;
object o = i;
```



Operators

- Arithmetic: + - * / %
- Logical: & | ^ ! ~ && || true false
- String concatenation: +
- Incrementation and decrementation: ++ --
- Bitwise shift: << >>
- Comparisons: == != < > <= >=
- Assignments: = += -= *= /= %= &= |= ^= <<= >>=
- Member access and indexing: . []
- Type cast: () as
- Conditional operator: ?:
- Creating objects, type information: new is sizeof typeof

Namespaces

```
namespace MyNamespace
{
   namespace Inner
   {
      class MyClass
      {
         public static void F() {}
      }
   }
}
```

```
using System;
using NamespaceAlias = MyNamespace.Inner;

static void Main()
{
   NamespaceAlias.MyClass.F();
}
```

Constants and Variables

```
static void Main()
{
   // constants
   const float r = 12.5;
   const float pi = 3.14f;

   // variables
   int a = 12, b = 1;
   int c;
//initialize only once
Public readonly double pi = 3.14

   c = 2;
   Console.WriteLine(a + b + c);
   Console.WriteLine(pi * r * r);
}
```

Enumerations

```
enum Color
{
   Red,
   Green = 10,
   Blue
}

static void Main(string[] args)
{
   Color c = Color.Green;

   DrawBox2D(10, 20, c);
   DrawBox2D(12, 10, Color.Blue);
}
```

Conditional Statement

```
static void Main(string[] args)
{
  if (args.Length == 0)
    Console.WriteLine(“No parameters”);
  else
  {
    Console.WriteLine(“{0} parameters”,
                      args.Length);
  }
}
```

Switch-Case Statement

```
static void Main(string[] args)
{
  switch (args.Length) {
    case 1:
    case 2:
      Console.WriteLine(“Too few parameters”);
      goto case 3;
    case 3:
      UseArgs(args);
      break;
    default:
      Console.WriteLine(“Error”);
      break;
  }
}
```

Loop Statements

```
static void Main(string[] args)
{
  for (int i = 0; i < args.Length; i++)
    Console.WriteLine(args[i]);

  int j = 0;
  while (j < args.Length)
    Console.WriteLine(args[j++]);

  do {
    Console.WriteLine(args[--j]);
  } while (j > 0);
}
```

Foreach statement

```
static void Main(string[] args)
{
  int[] tab = { 2, 4, 5, 11, -2 };

  foreach (int i in tab)
    Console.WriteLine(i);

}
```

Break and Continue

```
static void Main(string[] args)
{
  for (int i = 0; i < args.Length; i++)
  {
    if (args[i] == “end”)
      break;
    if (args[i] == “skip”)
      continue;
    Console.WriteLine(args[i]);
  }
}
```

Functions - Return Statement

```
static int Sum(int a, int b)
{
  return a + b;
}

static void Main(string[] args)
{
  Console.WriteLine(Sum(10, 1));
  return;
}
```

Exceptions – Throw and Try

```
static int Div(int a, int b) {
  if (b == 0)
    throw new Exception(“Dividing by zero”);
  return a / b;
}

static void Main(string[] args) {
  try {
    Console.WriteLine(Div(10, 0));
  }
  catch (Exception ex)
  {
    Console.WriteLine(“Error: ” + ex.Message);
  }
  finally
  {
    // always executes
  }
}
```

Checked Statement

```
static void Main(string[] args)
{
   int x = Int32.MaxValue;

   Console.WriteLine(x + 1); // overflow
                             // without exception
   checked {
      Console.WriteLine(x + 1); // exception !!!
   }
   unchecked {
      Console.WriteLine(x + 1); // overflow
                                // without exception
   }
}
```

Lock Statement

```
// ...

internal void Produce()
{
  lock (this) // critical section
  {
    counter++;
    data[counter] = NewValue();
  }
}

// ...
```

Arrays

```
// Single-dimensional arrays
int[] v1 = { 1, 4, 5, 7, 1 };
int[] v2 = new int[10];
int[] v3 = new int[5] { 1, -2, 3, -4, 5 };

v2[0] = v1[2];
v3[1] = 5;

// Multidimensional arrays
int[,] m1 = { {2, 3, 4}, {4, 5, 7} };
int[,] m2 = new int[2, 3];

m2[0, 0] = -1;
```

Jagged Arrays

```
int[][] t1 = { new int[2], new int[5] };

int[][] t2 = new int[2][];
t2[0] = new int[2];
t2[1] = new int[3];
t2[0, 0] = 1; t2[0, 1] = -5;
t2[1, 0] = 2; t2[1, 1] = 3; t2[1, 2] = 7;

for (int i = 0; i < t2.Length; i++)
  for (int j = 0; j < t2[i].Length; j++)
    Console.WriteLine(“[{0}][{1}] = {3}”,
                      i, j, t2[i][j]);
```

Non-zero based arrays

```
// Construct an array containing ints that has a length of 10 and a lower bound of 1
Array lowerBoundArray = Array.CreateInstance(typeof(int), new int[1] { 10 },
     new int[1] { 1 });

// insert 1 into position 1
lowerBoundArray.SetValue(1, 1);

//insert 2 into position 2
lowerBoundArray.SetValue(2, 2);

// IndexOutOfRangeException the lower bound of the array
// is 1 and we are attempting to write into 0
lowerBoundArray.SetValue(1, 0);
```

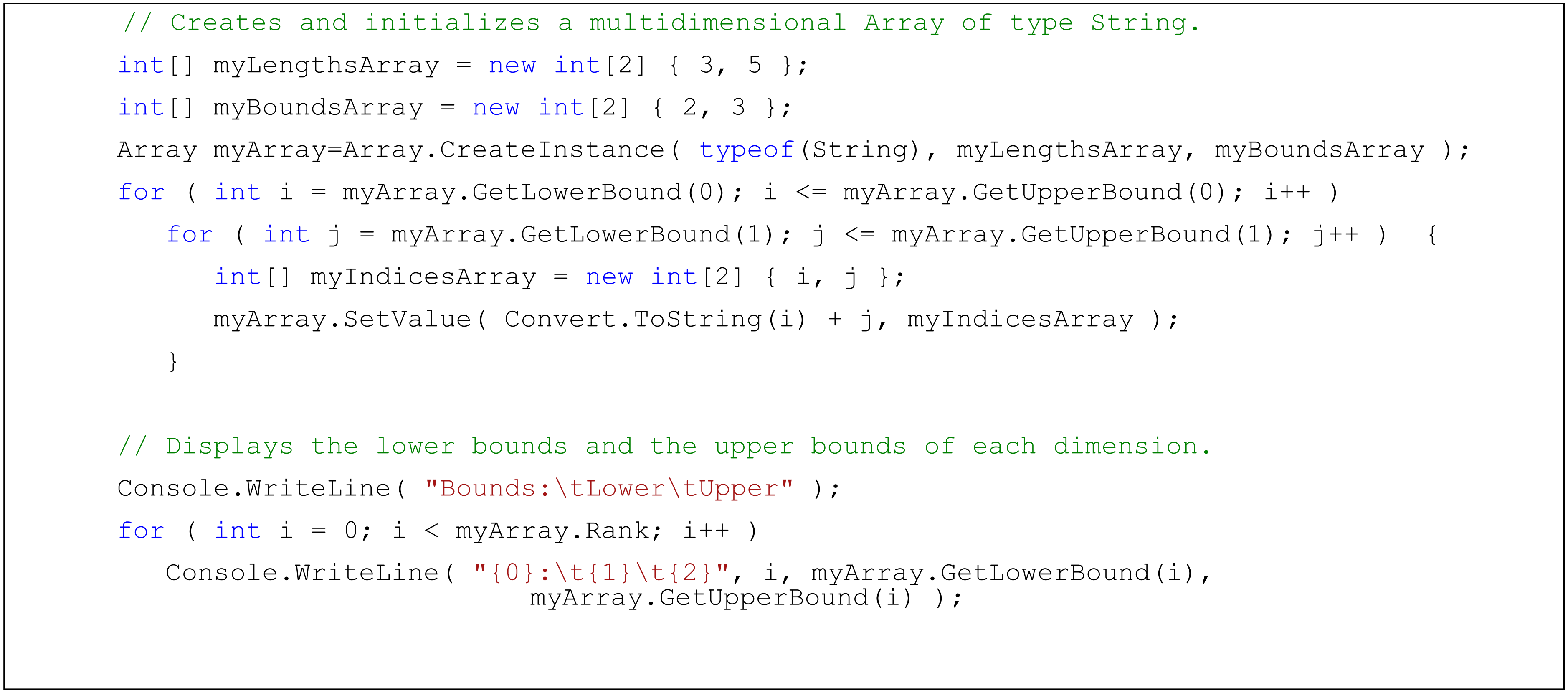
Non-zero based arrays

```
// Create the array.
Array myArray = Array.CreateInstance(typeof(double), new int[1] { 12 },
     new int[1] { 1 });

// Fill the array with random values.
Random rand = new Random();
for (int index = myArray.GetLowerBound(0); index <= myArray.GetUpperBound(0); index++)
{
myArray.SetValue(rand.NextDouble(), index);
}

// Display the values.
for (int index = myArray.GetLowerBound(0); index <= myArray.GetUpperBound(0); index++)
{
Console.WriteLine("myArray[{0}] = {1}", index, myArray.GetValue(index));
}
```

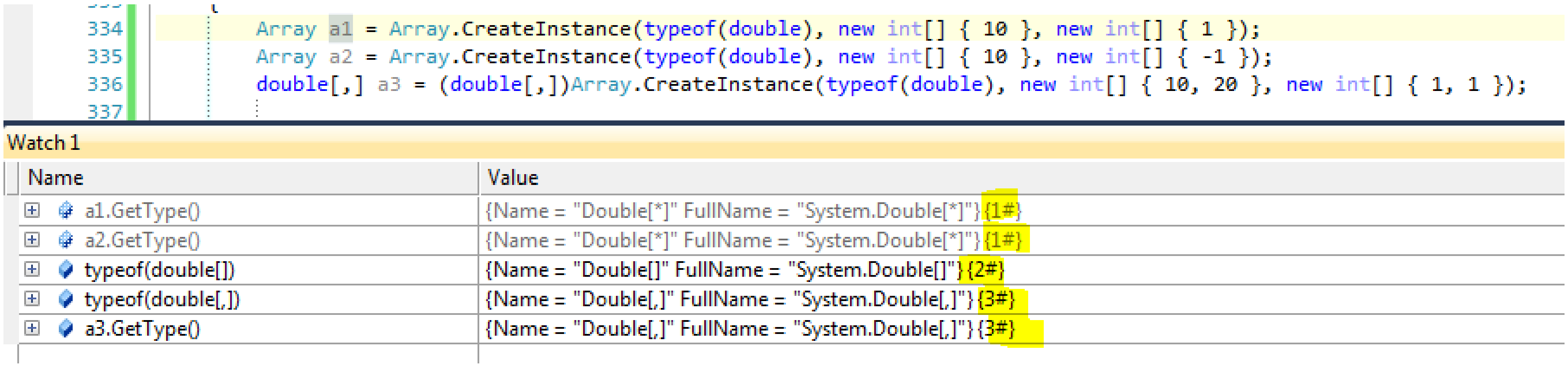
Non-zero based arrays



```
 // Creates and initializes a multidimensional Array of type String.
int[] myLengthsArray = new int[2] { 3, 5 };
int[] myBoundsArray = new int[2] { 2, 3 };
Array myArray=Array.CreateInstance( typeof(String), myLengthsArray, myBoundsArray );
for ( int i = myArray.GetLowerBound(0); i <= myArray.GetUpperBound(0); i++ )
   for ( int j = myArray.GetLowerBound(1); j <= myArray.GetUpperBound(1); j++ )  {
      int[] myIndicesArray = new int[2] { i, j };
      myArray.SetValue( Convert.ToString(i) + j, myIndicesArray );
   }

// Displays the lower bounds and the upper bounds of each dimension.
Console.WriteLine( "Bounds:\tLower\tUpper" );
for ( int i = 0; i < myArray.Rank; i++ )
   Console.WriteLine( "{0}:\t{1}\t{2}", i, myArray.GetLowerBound(i),
                      myArray.GetUpperBound(i) );
```

Non-zero based arrays



```
    {
        Array a1 = Array.CreateInstance(typeof(double), new int[] { 10 }, new int[] { 1 });
        Array a2 = Array.CreateInstance(typeof(double), new int[] { 10 }, new int[] { -1 });
        double[,] a3 = (double[,])Array.CreateInstance(typeof(double), new int[] { 10, 20 }, new int[] { 1, 1 });

```

Name	Value
a1.GetType()	{Name = "Double[*]" FullName = "System.Double[*]"}{1#}
a2.GetType()	{Name = "Double[*]" FullName = "System.Double[*]"}{1#}
typeof(double[])	{Name = "Double[]" FullName = "System.Double[]"}{2#}
typeof(double[,])	{Name = "Double[,]" FullName = "System.Double[,]"}{3#}
a3.GetType()	{Name = "Double[,]" FullName = "System.Double[,]"}{3#}

Basic Arrays Operations

```
int[] arr = new int[20];

// Reversing
Array.Reverse(arr);

// Sorting
Array.Sort(arr);

// Searching
int i1 = Array.IndexOf(arr, 5);
int i2 = Array.IndexOf(arr, 5, i1 + 1);
int i3 = Array.BinarySearch(arr, 10);

// Copying
arr.CopyTo(arr2);
```

Strings

```
string s1 = “Some text”;
string s2 = “c:\\temp\\myfile.dat”;
string s3 = @“c:\temp\myfile.dat”; // immediate
                                   // string
// Length
int len = s1.Length;

// Concatenation
s1 = s1 + “ i psa”;

// Indexing
for (int i = 0; i < s1.Length; i++)
  Console.WriteLine(„Znak {0} = {1}”, i, s1[i]);
```

Basic Strings Operations

```
string s1 = „Tekst”, s2 = „tekst”;
// Case-sensitive comparison
bool cmp1 = s1 == s2;
bool cmp2 = String.Compare(s1, s2) == 0;
// Case-insensitive comparison
bool cmp3 = String.Compare(s1, s2, true) == 0;

// Searching
int i1 = s1.IndexOf(“pattern”);

// Copying a substring
string substr = s1.Substring(2, 4);

// Replacing
s1 = s1.Replace(“old”, “new”);
```

Classes, Structures and Interfaces

- Class
  - defines a set of properties, methods and events
  - reference type (allocated on the heap)
- Structure
  - like class can contains data and methods
  - value type (stored on the stack)
  - cannot be inherited from
- Interface
  - similar to class, but do not provide implementation

Classes, Structures and Interfaces

- Class
  - Constructors
  - Destructors
  - Constants
  - Fields
  - Methods

Classes, Structures and Interfaces

- Class
  - Properties
  - Indexers
  - Operators
  - Events
  - Delegates
  - Classes
  - Interfaces
  - Structs

Classes

```
class Point
{
  public int x = 0, y = 0; // member variables

  public Point(int x, int y) // constructor
  {
    this.x = x;
    this.y = y;
  }

  public void Show() // member method
  {
    Console.WriteLine(“({0}, {1})”, x, y);
  }
}
```

Inheritance

```
// base class
class GraphObject
{
  public string name;

  public GraphObject(string name)
  {
    this.name = name;
  }

  public void Show() { }
}
```

Inheritance

```
// derived class
class Point: GraphObject
{ int x = 0, y = 0;

  public Point(string name, int x, int y):
    base(name)
  {
    this.x = x; this.y = y;
  }

  public void Show()
  {
    Console.WriteLine(“({0}, {1})”, x, y);
  }
}
```

Virtual Members and Overriding

```
class GraphObject
{
  public string name;

  public GraphObject(string name)
  {
    this.name = name;
  }

  // virtual method in base class
  public virtual void Show() { }
}
```

Virtual Members and Overriding

```
class Point: GraphObject
{ int x = 0, y = 0;

  public Point(string name, int x, int y):
    base(name)
  {
    this.x = x; this.y = y;
  }

  // overriden implementation of virtual method
  public override void Show()
  {
    Console.WriteLine(‟({0}, {1})″, x, y);
  }
}
```

Abstract Classes

```
// abstract class
abstract class GraphObject
{
  public string name;

  public GraphObject(string name)
  {
    this.name = name;
  }

  // abstract method (pure-virtual)
  public abstract void Show();
}
```

Sealed Methods and Members

```
// sealed class – inheritance from this class
//                is blocked
sealed class MySealedClass
{
  public string name;
  public int x, y;
}

class MyClass: MyBase
{
  // sealed method, cannot be overridden any more
  // in child classes
  public sealed override int Fn()
  {
    // ...
  }
}
```

Class vs Struct

- Structures cannot inherit.
- Structures cannot use the abstract, sealed, or static keywords because you cannot inherit from a structure.
- Structures are value types, whereas classes are reference types.

Interfaces

- **Signatures of methods,**
- **Properties,**
- **Events**
- **Indexers**

•An interface can inherit from one or more base interfaces

•When a base type list contains a base class and interfaces, the base class must come first in the list

Interfaces

```
interface IGraphObject
{
  void Show();
}

class Point: IGraphObject
{ // ...
  public void Show()
    { Console.WriteLine(„({0}, {1})”, x, y); }
}
// ...
Point p = new Point(2, 5);
IGraphObject graphObj = p as IGraphObject;
if (graphObj != null)
  graphObj.Show();
```

Interfaces

```
interface IGraphObject
{
   int x { get; set; }
   int y { get; set; }
}

class Point: IGraphObject
{
  private int _x;
  private int _y;

// ...
   public int x { get { return _x; } set { _x = value; } }
   public int y { get { return _y; } set { _y = value; } }
}
```

Interfaces

- Two interfaces that contain a member with the same signature

- Implementing that member on the class will cause both interfaces to use that member as their implementation

Interfaces

```
interface IControl
{
    void Paint();
}

interface ISurface
{
    void Paint();
}
class SampleClass : IControl, ISurface
{
   public void Paint()
   {
      Console.WriteLine("Paint method in SampleClass");
   }
}
```

Interfaces

```
SampleClass sc = new SampleClass();
IControl ctrl = (IControl)sc;
ISurface srfc = (ISurface)sc;

sc.Paint();
ctrl.Paint();
srfc.Paint();
```

Explicit Implementation

- If the two interface members do not perform the same function, however, this can lead to an incorrect implementation of one or both of the interfaces

- Used to resolve cases where two interfaces each declare different members of the same name such as a property and a method

Explicit Implementation

```
interface IOne
{
   void Execute();
}

interface ITwo
{
   void Execute();
}

class Tester: IOne, ITwo
{
   void IOne.Execute() { /* ... */ }
   void ITwo.Execute() { /* ... */ }
}
```

Explicit Implementation

```
Tester obj = new Tester();
IOne one = (IOne)obj;
one.Execute();
ITwo two = (ITwo)obj;
two.Execute();

Obj.Execute(); //??
```

Explicit Implementation

```
interface IOne
{
   void Execute();
}

interface ITwo
{
   void Execute();
}

class Tester: IOne, ITwo
{
   void Execute() { /* ... */ }
   void ITwo.Execute() { /* ... */ }
}
```

Explicit Implementation

- An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface (even if only one interface exists)

Members accessibility

- Accessibility modifiers for classes
  - **internal** – accessible from the same module
  - **public** – accessible from anywhere
- Accessibility modifiers for class members
  - **public** – accessible from anywhere
  - **protected**– accessible from the same class and from inherited classes
  - **private** – only from within the same class
  - **internal** – from the same module
  - **internal protected** – from the same module and from inherited classes

Module

- A module is a logical collection of code within an Assembly
- unit of compilation
- contains type metadata and compiled code
- does not contain an assembly manifest.
- A netmodule can not be deployed alone. It has to be linked into an assembly

Assembly

- Assembly is the minimum unit of deployment
- Assembly can contain one or more files
  - resource file,
  - netmodule,
  - native dlls
- Contains an assembly manifest

Finalizers

```
class ResourceWrapper
{
  int handle = 0;

  ResourceWrapper()
  {
    handle = GetWindowsResource();
  }

  ~ResourceWrapper() // finalizer
  {
    // doesn't known, when it will be called !!!
    FreeWindowsResource(handle);
  }
}
```

IDisposable Interface

```
class ResourceWrapper: IDisposable
{ /* ... */

  private void DoDispose() {
    FreeWindowsResource(handle);
    handle = 0;
  }

  public void Dispose() {
    DoDispose();
    GC.SuppressFinalize(this);
  }

  ~ResourceWrapper() { DoDispose(); }
}
```

GC.SuppressFinalize

- Prevent the garbage collector from calling Object.Finalize on an object that does not require it

Using Statement

```
static void Main(string[] args)
{
  using (MyResource r1 = new MyResource(),
         r2 = new MyResource())
  using (MyFile f1 = new MyFile())
  {
    r1.Use();
    r2.Use();
    f1.Read();
  } // calls r1.Dispose, r2.Dispose and f1.Dispose
    // (r1, r2 and f1 must implement IDisposable)
}
```

Using statement

- The using statement calls the Dispose method on the object in the correct way, and (when you use it as shown earlier) it also causes the object itself to go out of scope as soon as Dispose is called
- Within the **using** block, the object is read-only and cannot be modified or reassigned
- Statement ensures that Dispose is called even if an exception occurs

Using statement

```
{
  Font font1 = new Font("Arial", 10.0f);
  try
  {
    byte charset = font1.GdiCharSet;
  }
  finally
  {
    if (font1 != null)
      ((IDisposable)font1).Dispose();
  }
}
```

Using statement

```
using (Font font3 = new Font("Arial", 10.0f),
            font4 = new Font("Arial", 10.0f))
{

   //…
}
```

Using statement

```
Font font2 = new Font("Arial", 10.0f);

using (font2) // not recommended
{
    // use font2
}
float f = font2.GetHeight(); //??
```

Static class

- Contains only static members.
- Cannot be instantiated.
- Is sealed.
- Cannot contain Instance Constructors.
- Can define a static constructor.

Static Members

- Static
  - Methods,
  - Fields,
  - Properties
  - Events
- Are always accessed by the class name
- Only one copy of a static member exists (# of instances is irrelevant)

Static Members

```
class GraphObject
{
  static int counter = 0;
  string name;

  public GraphObject(){
    counter++;
    this.name = “GraphObject” +
                counter.ToString();
  }

  public static void ResetCounter() {
    counter = 0;
  }
}
```

Static Members

- Can be overloaded
- Cannot be overridden

- C# does not support static local variables
- Static members are initialized before the static member is accessed for the first time and before the static constructor

Properties

- Enable a class to expose a public way of getting and setting values
- Hide implementation
- get / set (can have different access levels)
- Value
- Auto-implemented properties

Properties

```
class Point
{
  string name; int x = 0, y = 0;

  public string Name
  {
    get { return name; }
  }

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

Indexer

- Indexers enable objects to be indexed in a similar manner to arrays.
- A get accessor returns a value.
- A set accessor assigns a value.
- The this keyword is used to define the indexers.

Indexer

- The value keyword is used to define the value being assigned by the set indexer.
- Indexers **do not have to be** indexed by an integer value
- Indexers can be overloaded.
- Indexers can have more than one formal parameter

Indexer

- Class
- Struct
- Interface

Indexer (Default Poroperty)

```
class SampleCollection<T>
{
private T[] arr = new T[100];

public T this[int i]
    {
        get
        {
            return arr[i];
        }
        set
        {
            arr[i] = value;
        }
    }
}
```

Indexer (Default Poroperty)

```
class Worksheet
{
  CellValue[,] cells = new CellValue[20,20];

  public CellValue this[string col, int row]
  {
    get { return data[row, ColToIndex(col)]; }
    set { data[row, ColToIndex(col)] = value; }
  }
}

/* ... */
Worksheet sheet = new Worksheet();
sheet[“A”, 10] = “=A1+B1”;
```

Indexer

- Indexer in interface
  - do not use modifiers
  - do not have a body
  - Two interfaces

```
public interface ISomeInterface
{
string this[int index]
    {
        get;
        set;
    }
}
```

Parameter passing

- Value-Type Parameters
  - By value: passing a copy of the variable to the method
  - ref or out keyword
- Reference-Type Parameters
  - By value: by value, it is possible to change the data pointed to by the reference
  - Cannot change the value of the reference itself (i.e. cannot use the same reference to allocate memory for a new class and have it persist outside the block)
  - ref or out keyword

Passing Method Parameters

```
class Point
{
  public int x = 0, y = 0;
  public void SetXY(int x, int y)
   { this.x = x; this.y = y; }
  public void GetXY(ref int x, ref int y)
   { x = this.x; y = this.y; }
}

// ...
Point p = new Point();
int x0 = 5, x1 = 0, y1 = 0;
p.SetXY(x0, 4);// parameters passed by value
p.GetXY(ref x1, ref y1);  // passed by reference
```

Overloaded Methods

```
class Point
{
  public int x = 0, y = 0;

  public Point(int x, int y)
  {
    this.x = x; this.y = y;
  }

  public Point(Point p)
  {
    x = p.x; y = p.y;
  }
}
```

Overloaded Methods

- **Multiple parameters**
- *Better function member* rules
- In order to be selected Method must be *at least as good* for each parameter, and *better* for at least one parameter
- If no method wins outright, the compiler will report an error
- This is done on a method-by-method comparison: a method doesn't have to be better than *all* other methods for any single parameter.

Multiple parameters

```
using System;

class Test
{
    static void Foo(int x, int y)
    {
        Console.WriteLine("Foo(int x, int y)");
    }

    static void Foo(int x, double y)
    {
        Console.WriteLine("Foo(int x, double y)");
    }

    static void Foo(double x, int y)
    {
        Console.WriteLine("Foo(double x, int y)");
    }

    static void Main()
    {
        Foo(5, 10);
    }
}
```

Multiple parameters

```
using System;

class Test
{
    static void Foo(int x, double y)
    {
        Console.WriteLine("Foo(int x, double y)");
    }

    static void Foo(double x, int y)
    {
        Console.WriteLine("Foo(double x, int y)");
    }

    static void Main()
    {
        Foo(5, 10);
    }
}
```

- error CS0121: The call is ambiguous between the following methods or properties: 'Test.Foo(int, double)' and 'Test.Foo(double, int)'

Overloaded Methods

- Inheritance
- Compiler considers the compile-time class of the "target" of the call, and looks at its methods. If it can't find anything appropriate, it then looks at the parent class... Etc.
- Issue: If if finds a method then it uses it, it does not check wheter base classes have better fit functions

```
using System;

class Parent
{
    public void Foo(int x)
    {
        Console.WriteLine("Parent.Foo(int x)");
    }
}

class Child : Parent
{
    public void Foo(double y)
    {
        Console.WriteLine("Child.Foo(double y)");
    }
}


class Test
{
    static void Main()
    {
        Child c = new Child();
        c.Foo(10);
    }
}
```

Interesting

```
class Parent{
    public virtual void Foo(int x) {
            Console.WriteLine("Parent.Foo(int x)");}
}
class Child : Parent {
    public override void Foo(int x){
        Console.WriteLine("Child.Foo(int x)");
    }
    public void Foo(double y) {
        Console.WriteLine("Child.Foo(double y)");
    }}
class Test {
    static void Main() {
        Child c = new Child();
        c.Foo(10);
    }
}
```

- C.foo(double) !!!

- Return types
  - Not considered
- Optional parameters
  - If there is a choice between method versions with filling and not filling of optional parameters, compiler will will pick the method where the caller has specified all the arguments explicitly.
  - If all methods requre filling then compiler raises an error

```
using System;

class Test
{
    static void Foo(int x, int y = 5)
    {
        Console.WriteLine("Foo(int x, int y = 5)");
    }

    static void Foo(int x)
    {
        Console.WriteLine("Foo(int x)");
    }

    static void Main()
    {
        Foo(10);
    }
}
```

```
using System;

class Test
{
    static void Foo(int x, int y = 5, int z = 10)
    {
        Console.WriteLine("Foo(int x, int y = 5, int z = 10)");
    }

    static void Foo(int x, int y = 5)
    {
        Console.WriteLine("Foo(int x, int y = 5)");
    }

    static void Main()
    {
        Foo(10);
    }
}
```

```
using System;

class Test
{
    static void Foo(int x, int y = 5)
    {
        Console.WriteLine("Foo(int x, int y = 5)");
    }

    static void Foo(double x)
    {
        Console.WriteLine("Foo(double x)");
    }

    static void Main()
    {
        Foo(10);
    }
}
```

Named arguments

- Allow for reduction of the set of applicable function members by ruling out ones which have the "wrong" parameter names

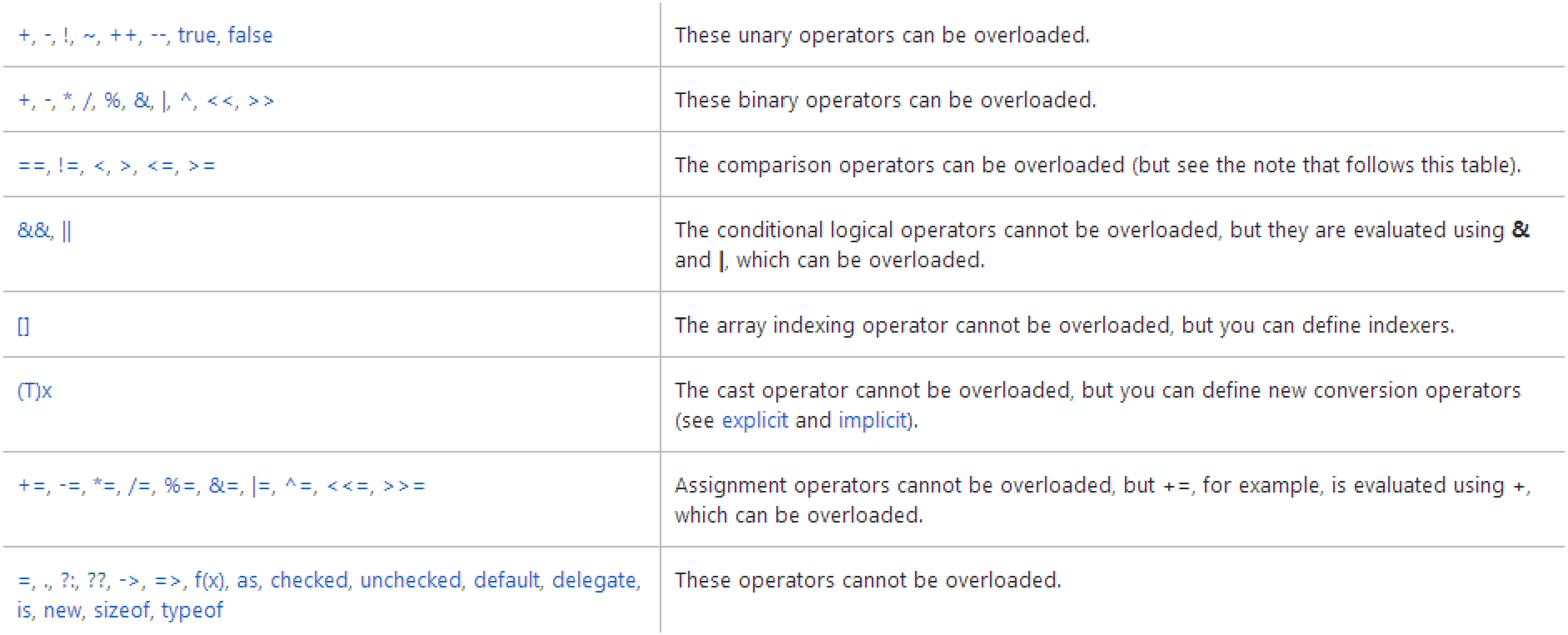
-
  ```
  class Test {
      static void Foo(int x) {
          Console.WriteLine("Foo(int x)");
      }
      static void Foo(double y) {
          Console.WriteLine("Foo(double y)");
      }


      static void Main() {
          Foo(y: 10);
      }
  }
  ```

Overloaded Methods

- New version of the language can introduce new conversions

Operator Overloading



+, -, !, ~, ++, --, true, false	These unary operators can be overloaded.
+, -, *, /, %, &, |, ^, <<, >>	These binary operators can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded (but see the note that follows this table).
&&, ||	The conditional logical operators cannot be overloaded, but they are evaluated using **&** and **|**, which can be overloaded.
[]	The array indexing operator cannot be overloaded, but you can define indexers.
(T)x	The cast operator cannot be overloaded, but you can define new conversion operators (see explicit and implicit).
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Assignment operators cannot be overloaded, but +=, for example, is evaluated using +, which can be overloaded.
=, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof	These operators cannot be overloaded.

Operator Overloading

```
class Complex
{
  double re = 0, im = 0;

  public Complex(double re, double im) {
    this.re = re; this.im = im;
  }

  public static Complex operator+(Complex c1,
                                  Complex c2)
  {
    return new Complex(c1.re + c2.re,
                       c1.im + c2.im);
  }
}
```

Class Conversions

```
class BaseClass
{ public virtual string GetName()
  { return “Base”; } }

class DerivedClass : BaseClass
{ public override string GetName()
  { return “Derived”; } }

DerivedClass d = new DerivedClass();
BaseClass b1 = d, b2 = (BaseClass) d;
d.GetName();                // all return
b1.GetName(); b2.GetName(); // “Derived”

// conversion without raising exception on failure
b = p as BaseClass; if (p!=null) { ... }
```

Class conversion

- Declare conversions on classes or structs
- Conversions are defined like operators and are named for the type to which they convert.
- Either the type of the argument to be converted, or the type of the result of the conversion, but not both, must be the containing type

- Conversions declared as **implicit** occur automatically when it is required.
- Conversions declared as **explicit** require a cast to be called.
- All conversions must be declared as **static**.

```
class SampleClass {
    public static explicit operator SampleClass(int i) {
        SampleClass temp = new SampleClass();
        // code to convert from int to SampleClass...

        return temp;
    }
}
```

- class SampleClass {
- public static explicit operator int(SampleClass i) {
  - // code to convert from SampleClass to int
-
- }
- }

Delegates

- A delegate is a type that represents references to methods with a particular parameter list and return type
- public delegate int PerformCalculation(int x, int y);
- Compatible signature and return type

Delegates

- Any method from any accessible **class or struct** that matches the delegate type can be assigned to the delegate
- The method can be either static or an instance method
- Callback methods

Delegates

- Like C++ function pointers but are type safe.
- Allow methods to be passed as parameters.
- Can be chained together; e.g. multiple methods can be called on a single event.
- Methods do not have to match the delegate type exactly.

Delegates

- A delegate can call more than one method when invoked. (Multicasting)
- =
- +=
- -=
- GetInvocationList()
- Length

Delegates

```
delegate void MyDelegate(string arg);

static void F1(string arg) {
  Console.WriteLine(„F1( {0} )”, arg);
}
static void F2(string arg) {
  Console.WriteLine(„F2( {0} )”, arg);
}

static void Main(string[] args)
{
  MyDelegate fx = new MyDelegate(F1);
  fx += new MyDelegate(F2);
  fx(args[0]); // calls F1 and F2
}              // with parameter args[0]
```

Events

```
delegate void ClickHandler();

class Button
{
  public event ClickHandler Click;
  public void PerformClick() { Click(); }
}
/* ... */
public static void OnClick() { /* ... */ }

static void Main() {
  Button bt = new Button();
  bt.Click += new ClickHandler(OnClick);
  bt.PerformClick();
}
```

Anonymous Methods

```
// without anonymous methods
public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();
    button1.Click += new EventHandler(ButtonClick);

    private void ButtonClick(object sender,
                             EventArgs e)
    {
      MessageBox.Show("Click!");
    };
  }
}
```

Anonymous Methods

```
// using anonymous methods
public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();

    button1.Click += delegate(object sender,
                              EventArgs e)
    {
      // body of anonymous method
      MessageBox.Show("Click!");
    };
  }
}
```

C# 2.0

Partial Class Definitions

```
// part of MyClass defined in file MyClass1.cs
public partial class MyClass
{
  public int MethodA() { }
}
```

```
// part of MyClass defined in file MyClass2.cs
public partial class MyClass
{
  public int MethodB() { }
}
```

C# 2.0

Generic Classes

```
class ObjectList<ItemType>: CollectionBase
{
  public int Add(ItemType value)
    { return InnerList.Add(value); }

  public void Remove(ItemType value)
    { InnerList.Remove(value); }

  public ItemType this[int index]
  {
    get { return (ItemType)InnerList[index]; }
    set { InnerList[index] = value; }
  }
}
```

C# 2.0

Generic Classes

```
// declare the generic class
// with a few type paramters
class ObjectHashTable<ItemType,
  KeyType>: DictionaryBase
{
  // ...
}

// declare the generic class
// with constraints on type parameter
class ObjectList<ItemType>: CollectionBase
  where ItemType: ISerializable
{
  // ...
}
```

C# 2.0

Constraints of Type Parameters

- **where T : struct – the type argument must be a value type (except nullable types)**
- **where T : class – the type argument must be a reference type**
- **where T : new() – the type argument must have a public parameterless constructor**
- **where T : <class_name> – t*he type argument must be or derive from the specified class***
- **where T : <interface> – t*he type argument must be or implement the specified interface***

Generic Methods

```
class MyClass
{
   // declare the generic method
   public ItemType GenericFn<ItemType>(ItemType item)
   {
      // ...
      return item;
   }
}
```

C# 2.0

Generic Interfaces Generic Delegates

```
// generic interfaces
interface IMyInterface<T> { }
interface I2<T>: IMyInterface<T> { }   // ok
class MyClass<T>: MyInterface<T> { }   // ok
class MyClass<T>: MyInterface<int> { } // ok
class MyClass: MyInterface<T> { }      // error!

// generic delegates
public delegate void MyDelegate<T>(T item);
public static void Notify(int i) { }

MyDelegate<int> m = new MyDelegate<int>(Notify);
```

C# 2.0

Default Keyword in Generic Code

```
public class GenericList<T>
{
  /* ... */

  public T GetNext()
  {
    T temp = default(T);// default returns null
                        // if T is reference type,
                        // or 0 if T is value type
    /* ... */

    return temp;
  }
}
```

C# 2.0