Basic Introduction to C#

Eero Huusko
PSK-Service Oy
2011

Why C# ?

- Builds on COM+ experience
- Native support for
  - Namespaces
  - Versioning
  - Attribute-driven development
- Power of C with ease of Microsoft Visual Basic®
- Minimal learning curve for everybody
- Much cleaner than C++
- More structured than Visual Basic
- More powerful than Java

C# – The Big Ideas

A component oriented language

- The first “component oriented” language in the C/C++ family
  - In OOP a component is: A reusable program that can be combined with other components in the same system to form an application.
  - Example: a single button in a graphical user interface, a small interest calculator
  - They can be deployed on different servers and communicate with each other

- Enables one-stop programming
  - No header files, IDL, etc.
  - Can be embedded in web pages

PSK SERVICE OY

C# – The Big Ideas cond’

Robust and durable software

- Garbage collection
  - No memory leaks and stray pointers
- Exceptions
  - Error handling is not an afterthought
- Type-safety
  - No uninitialized variables, unsafe casts
- Versioning
  - Pervasive versioning considerations in all aspects of language design

C# Overview

- Object oriented
- Everything belongs to a class
  - no global scope
- Complete C# program:

```
using System;
namespace ConsoleTest
{
        class Class1
        {
                static void Main(string[] args)
                {
                }
        }
}
```

C# Program Structure

- Namespaces
  - Contain types and other namespaces
- Type declarations
  - Classes, structs, interfaces, enums, and delegates
- Members
  - Constants, fields, methods, properties, indexers, events, operators, constructors, destructors
- Organization
  - No header files, code written “in-line”
  - No declaration order dependence

C# Program Structure

```
using System;

namespace System.Collections
{
   public class Stack
   {
      Entry top;

      public void Push(object data) {
         top = new Entry(top, data);
      }

      public object Pop() {
         if (top == null) throw new InvalidOperationException();
         object result = top.data;
         top = top.next;
         return result;
     }
   }
}
```

Simple Types

- Integer Types
  - **byte**, **sbyte** (8bit), **short**, **ushort** (16bit)
  - **int**, **uint** (32bit), **long**, **ulong** (64bit)
- IEEE Floating Point Types
  - **float** (precision of 7 digits)
  - **double** (precision of 15–16 digits)
- Exact Numeric Type
  - **decimal** (28 significant digits)
- Character Types
  - **char** (single character)
  - **string** (rich functionality, by-reference type)
- Boolean Type
  - **bool** (distinct type, **not** interchangeable with **int**)
- You can create your own types
- All data and code is defined within a type
  - No global variables, no global functions

PSK SERVICE OY

Types - Unified Type System

	Value (Struct)	**Reference (Class)**
Variable holds	Actual value	Memory location
Allocated on	Stack, member	Heap
Nullability	Always has value	May be null
Default value	0	null
Aliasing (in a scope)	No	Yes
Assignment means	Copy data	Copy reference

PSK SERVICE OY

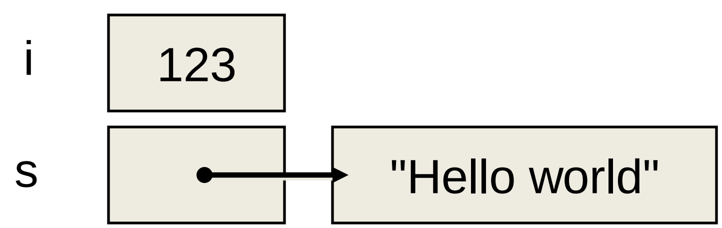
Types - Overview

- Types can be instantiated…
  - and then used: call methods, get and set properties, etc.
- Can convert from one type to another
  - Implicitly and explicitly
- Types are organized
  - Namespaces, files, assemblies
- There are two categories of types: value and reference
- Types are arranged in a hierarchy

Types - Unified Type System

- Value types
  - Directly contain data
  - Cannot be null
- Reference types
  - Contain references to objects
  - May be null

```
int i = 123;
string s = "Hello world";
```



Predefined Types - Integral Types

C# Type	System Type (.Net Type)	Size (bytes)	Signed?
`sbyte`	`System.Sbyte`	1	Yes
`short`	`System.Int16`	2	Yes
`int`	`System.Int32`	4	Yes
`long`	`System.Int64`	8	Yes
`byte`	`System.Byte`	1	No
`ushort`	`System.UInt16`	2	No
`uint`	`System.UInt32`	4	No
`ulong`	`System.UInt64`	8	No

Type System

- Value types
  - Primitives `int i;`
  - Enums `enum State { Off, On }//or {Off=1, On}`
  - Structs `struct Point { int x, y; }`
- Reference types
  - Classes `class Foo: Bar, IFoo {...}`
  - Interfaces `interface IFoo: IBar {...}`
  - Arrays `string[] a = new string[10];`
  - Delegates `delegate void Empty();`

Types - Conversions

- Implicit conversions
  - Occur automatically
  - Guaranteed to succeed
  - No information (precision) loss
- Explicit conversions
  - Require a cast
  - May not succeed
  - Information (precision) might be lost
- Both implicit and explicit conversions can be user-defined

Types - Conversions

```
int x = 123456;
long y = x;                        // implicit
short z = (short)x;                // explicit

double d = 1.2345678901234;
float f = (float)d;                // explicit
long l = (long)d;                  // explicit
```

Types - Unified Type System

- Polymorphism
  - The ability to perform an operation on an object without knowing the precise type of the object

```
void Poly(object o) {
  Console.WriteLine(o.ToString());
}
```

```
Poly(42);
Poly("abcd");
Poly(12.345678901234m);
Poly(new Point(23,45));
```

PSK SERVICE OY

Statements and Comments

- Case sensitive (myVar != MyVar)
- Statement delimiter is semicolon ;
- Block delimiter is curly brackets { }
- Single line comment is //
- Block comment is /* */
  - Save block comments for debugging!

Data

- All data types derived from ***System.Object***
- Declarations:

  datatype varname;

  datatype varname = initvalue;

- C# does not automatically initialize local variables (but will warn you)!

Value Data Types

- Directly contain their data:
  - int (numbers)
  - long (really big numbers)
  - bool (true or false)
  - char (unicode characters)
  - float (7-digit floating point numbers)
  - string (multiple characters together)

Predefined Types - Floating Point Types

- Follows IEEE 754 specification
- Supports ± 0, ± Infinity, NaN

C# Type	System Type	Size (bytes)
`float`	`System.Single`	4
`double`	`System.Double`	8

Predefined Types - `decimal`

- 128 bits
- Essentially a 96 bit value scaled by a power of 10
- Decimal values represented precisely

C# Type	System Type	Size (bytes)
`decimal`	`System.Decimal`	16

Predefined Types - `decimal`

- All integer types can be implicitly converted to a decimal type
- Conversions between decimal and floating types require explicit conversion due to possible loss of precision
- s * m * 10e
  - s = 1 or –1
  - $0 \leq m \leq 296$
  - $-28 \leq e \leq 0$

Data Manipulation

=	assignment
+	addition
-	subtraction
*	multiplication
/	division
%	modulus
++	increment by one
--	decrement by one

strings

- Immutable sequence of Unicode characters (char)

- Creation:
  - string s = “Eero”;
  - string s = new String(“Eero”);

- Backslash is an escape:
  - Newline: “\n”
  - Tab: “\t”

PSK SERVICE OY

string/int conversions

- string to numbers:
  - int i = int.Parse(“12345”);
  - float f = float.Parse(“123.45”);

- Numbers to strings:
  - string msg = “Your number is ” + 123;
  - string msg = “It costs ” +
    string.Format(“{0:C}”, 1.23);

String Example

```
using System;
namespace ConsoleTest
{
        class Class1
        {
                static void Main(string[ ] args)
                {
                        int myInt;
                        string myStr = "2";
                        bool myCondition = true;

                        Console.WriteLine("Before: myStr = " + myStr);
                        myInt = int.Parse(myStr);
                        myInt++;
                        myStr = String.Format("{0}", myInt);
                        Console.WriteLine("After: myStr = " + myStr);

                        while(myCondition) ;
                }
        }
}
```

Arrays

- Zero based, type bound
- Built on .NET **System.Array** class
- Declared with type and shape, but no bounds
  - `int [ ] SingleDim;`
  - `int [ , ] TwoDim;`
  - `int [ ][ ] Jagged;`
- Created using **new** with bounds or initializers
  - `SingleDim = new int[20];`
  - `TwoDim = new int[,]{{1,2,3},{4,5,6}};`
  - `Jagged = new int[1][ ];`
    `Jagged[0] = new int[ ]{1,2,3};`

Arrays

- Derived from System.Array
- Use square brackets [ ]
- Zero-based
- Static size
- Initialization:
  - int [ ] nums;
  - int [ ] nums = new int[3];   // 3 items
  - int [ ] nums = new int[ ] {10, 20, 30};

Arrays Continued

- Use Length for # of items in array:
  - nums.Length
- Static Array methods:
  - Sort System.Array.Sort(myArray);
  - Reverse System.Array.Reverse(myArray);
  - IndexOf
  - LastIndexOf

  Int myLength = myArray.Length;

  System.Array.IndexOf(myArray, “K”, 0, myLength)

Arrays Final

- Multidimensional

```
// 3 rows, 2 columns
int [ , ] myMultiIntArray = new int[3,2]

for(int r=0; r<3; r++)
{
        myMultiIntArray[r][0] = 0;
        myMultiIntArray[r][1] = 0;
}
```

Types – Arrays examples

- Declare

```
int[] primes;
```

- Allocate

```
int[] primes = new int[9];
```

- Initialize

```
int[] prime = new int[] {1,2,3,5,7,11,13,17,19};
int[] prime = {1,2,3,5,7,11,13,17,19};
```

- Access and assign

```
prime2[i] = prime[i];
```

- Enumerate

```
foreach (int i in prime) Console.WriteLine(i);
```

Conditional Operators

== equals

!= not equals

< less than

<= less than or equal

> greater than

>= greater than or equal

&& and

|| or

If, Case Statements

```
if (expression)
    { statements; }
else if
    { statements; }
else
    { statements; }
```

```
switch (i) {
    case 1:
        statements;
        break;
    case 2:
        statements;
        break;
    default:
        statements;
        break;
}
```

Loops

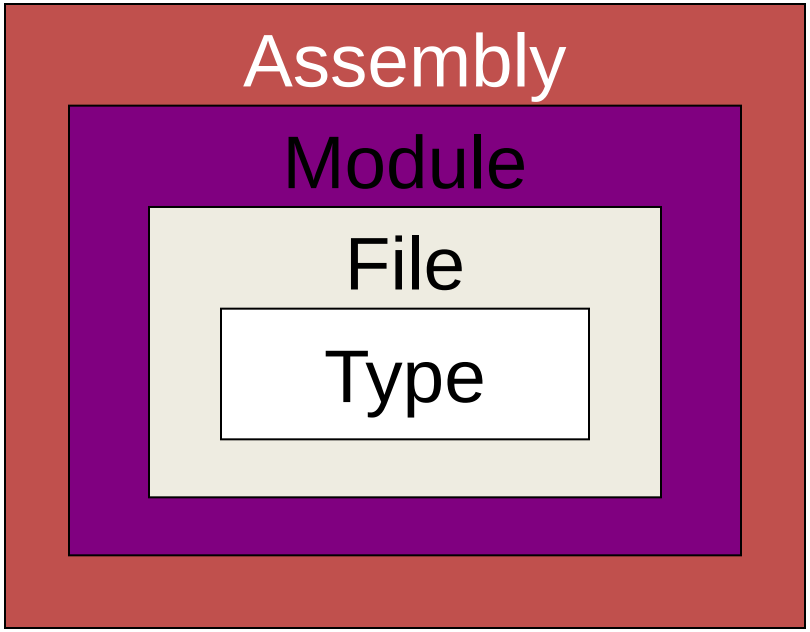
```
for (initialize-statement; condition; increment-statement);
{
    statements;
}
```

```
while (condition)
{
    statements;
}
```

Note: can include *break* and *continue* statements

Program Structure - Organizing Types

- Physical organization
  - Types are defined in files
  - Files are compiled into modules
  - Modules are grouped into assemblies



Program Structure - Organizing Types

- Types are defined in files
  - A file can contain multiple types
  - Each type is defined in a single file
- Files are compiled into modules
  - Module is a DLL or EXE
  - A module can contain multiple files
- Modules are grouped into assemblies
  - Assembly can contain multiple modules
  - Assemblies and modules are often 1:1

Program Structure - Organizing Types

- Types are defined in ONE place
  - “One-stop programming”
  - No header and source files to synchronize
  - Code is written “in-line”
  - Declaration and definition are one and the same
  - A type must be fully defined in one file
    - Can’t put individual methods in different files
- No declaration order dependence
  - No forward references required

Program Structure - Namespaces

- Namespaces provide a way to uniquely identify a type
- Provides logical organization of types
- Namespaces can span assemblies
- Can nest namespaces
- There is no relationship between namespaces and file structure (unlike Java)
- The fully qualified name of a type includes all namespaces

Program Structure - Namespaces

```
namespace N1 {                    // N1
  class C1 {                      // N1.C1
    class C2 {                    // N1.C1.C2
    }
  }
  namespace N2 {                  // N1.N2
    class C2 {                    // N1.N2.C2
    }
  }
}
```

Program Structure - Namespaces

- The using statement lets you use types without typing the fully qualified name
- Can always use a fully qualified name

```
using N1;

C1 a;           // The N1. is implicit
N1.C1 b;        // Fully qualified name

C2 c;           // Error! C2 is undefined
N1.N2.C2 d;     // One of the C2 classes
C1.C2 e;        // The other one
```

Note that it is N1.C1, not N1::C1

Program Structure - Namespaces

- The `using` statement also lets you create aliases

```
using C1 = N1.N2.C1;
using N2 = N1.N2;

C1 a;               // Refers to N1.N2.C1
N2.C1 b;            // Refers to N1.N2.C1
```

Program Structure - Namespaces

- Best practice: Put all of your types in a unique namespace
- Have a namespace for your company, project, product, etc.
- Look at how the .NET Framework classes are organized

Program Structure - References

- In Visual Studio you specify references for a project
- Each reference identifies a specific assembly
- Passed as reference (`/r` or `/reference`) to the C# compiler

```
csc HelloWorld.cs /reference:System.WinForms.dll
```

Program Structure - Namespaces vs. References

- Namespaces provide language-level naming shortcuts
  - Don’t have to type a long fully qualified name over and over
- References specify which assembly to use

Program Structure - Main Method

- Execution begins at the static `Main()` method
- Can have only one method with one of the following signatures in an assembly
  - `static void Main()`
  - `static int Main()`
  - `static void Main(string[] args)`
  - `static int Main(string[] args)`

Program Structure - Syntax

- Identifiers
  – Names for types, methods, fields, etc.
  – Must be whole word – no white space
  – Unicode characters
  – Begins with letter or underscore
  – Case sensitive
  – Must not clash with keyword
    - Unless prefixed with @

Statements - Syntax

- Statements are terminated with a semicolon (**;**)
- Just like C, C++ and Java
- Block statements { ... } don’t need a semicolon

Statements - Variables and Constants

```
static void Main() {
  const float pi = 3.14f;
  const int r = 123;
  Console.WriteLine(pi * r * r);

  int a;
  int b = 2, c = 3;
  a = 1;
  Console.WriteLine(a + b + c);
}
```

Statements - Variables

- Variables must be assigned a value before they can be used
  - Explicitly or automatically
  - Called definite assignment
- Automatic assignment occurs for static fields, class instance fields and array elements

```
void Foo() {
  string s;
  Console.WriteLine(s);      // Error
}
```

PSK SERVICE OY

Statements - Labeled Statements & goto

- `goto` can be used to transfer control within or out of a block, but not into a nested block

```
static void Find(int value, int[,] values,
                 out int row, out int col) {
   int i, j;
   for (i = 0; i < values.GetLength(0); i++)
      for (j = 0; j < values.GetLength(1); j++)
         if (values[i, j] == value) goto found;
   throw new InvalidOperationException("Not found");
found:
   row = i; col = j;
}
```

Statements - Expression Statements

- Statements must do work
  - Assignment, method call, ++, --, new

```
static void Main() {
  int a, b = 2, c = 3;
  a = b + c;
  a++;
  MyClass.Foo(a,b,c);
  Console.WriteLine(a + b + c);
  a == 2;                          // ERROR!
}
```

Statements - Exception Handling

- Exceptions are the C# mechanism for handling unexpected error conditions
- Superior to returning status values
  - Can’t be ignored
  - Don’t have to handled at the point they occur
  - Can be used even where values are not returned (e.g. accessing a property)
  - Standard exceptions are provided

Statements - Exception Handling

- `try...catch...finally` statement
- `try` block contains code that could throw an exception
- `catch` block handles exceptions
  - Can have multiple catch blocks to handle different kinds of exceptions
- `finally` block contains code that will always be executed
  - Cannot use jump statements (e.g. `goto`) to exit a finally block

Statements - Exception Handling

- `throw` statement raises an exception
- An exception is represented as an instance of `System.Exception` or derived class
  - Contains information about the exception
  - Properties
    - **`Message`**
    - **`StackTrace`**
    - **`InnerException`**
- You can rethrow an exception, or catch one exception and throw another

PSK SERVICE OY

Statements

Exception Handling

```
try {
  Console.WriteLine("try");
  throw new Exception("message");
}
catch (ArgumentNullException e) {
  Console.WriteLine("caught null argument");
}
catch {
  Console.WriteLine("catch");
}
finally {
  Console.WriteLine("finally");
}
```

Classes, Members and Methods

- Everything is encapsulated in a class
- Can have:
  - member data
  - member methods

```
Class clsName
 {
      modifier dataType varName;
      modifier returnType methodName (params)
      {
         statements;
          return returnVal;
      }
 }
```

Class Constructors

- Automatically called when an object is instantiated:

```
public className(parameters)
{
    statements;
}
```

Hello World

```
namespace Sample
{
    using System;

    public class HelloWorld
    {
        public HelloWorld()
        {
        }

        public static int Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            return 0;
        }
    }
}
```

Constructor

Another Example

```
using System;
namespace ConsoleTest
{
  public class Class1
  {
    public string FirstName = “Eero";
    public string LastName = “Huusko";

    public string GetWholeName()
    {
             return FirstName + " " + LastName;
    }

    static void Main(string[] args)
    {
             Class1 myClassInstance = new Class1();

             Console.WriteLine("Name: " +
             myClassInstance.GetWholeName());

             while(true) ;
    }
  }
 }
```

Hello World Anatomy

- Contained in its own namespace
- References other namespaces with "using"
- Declares a publicly accessible application class
- Entry point is "`static int Main( ... )`"
- Writes "Hello World!" to the system console
  - Uses static method **WriteLine** on **System.Console**

Classes

- Single inheritance
- Multiple interface implementation
- Class members
  - Constants, fields, methods, properties, indexers, events, operators, constructors, destructors
  - Static and instance members
  - Nested types
- Member access
  - public, protected, internal, private

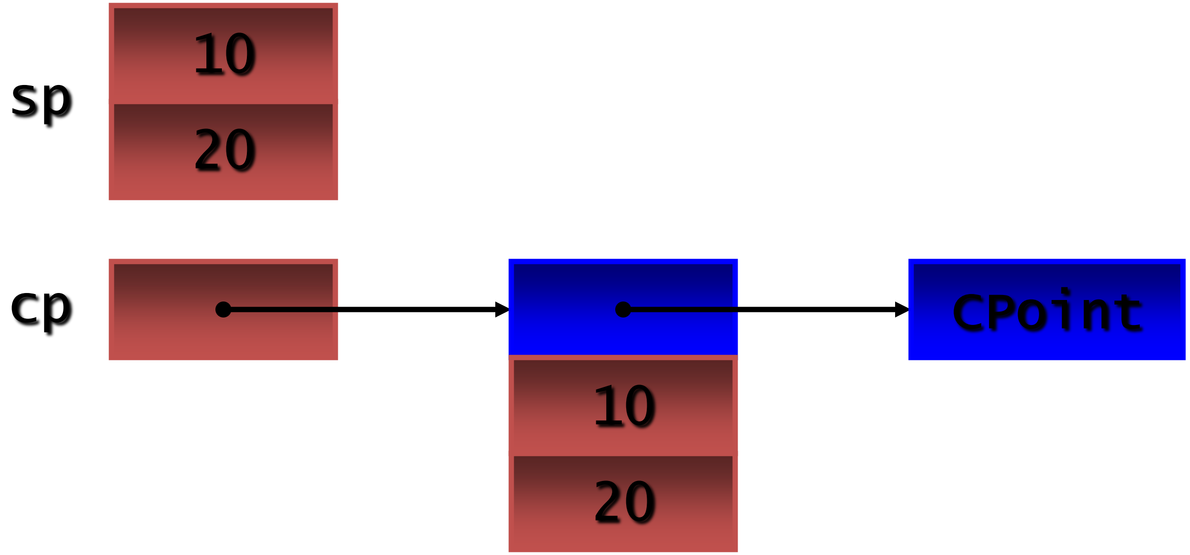
Structs

- Like classes, except
  - Stored in-line, not heap allocated
  - Assignment copies data, not reference
  - No inheritance
- Ideal for light weight objects
  - Complex, point, rectangle, color
  - int, float, double, etc., are all structs
- Benefits
  - No heap allocation, less GC pressure
  - More efficient use of memory

Classes And Structs

```
class CPoint { int x, y; ... }
struct SPoint { int x, y; ... }

CPoint cp = new CPoint(10, 20);
SPoint sp = new SPoint(10, 20);
```



Interfaces

- Multiple inheritance
- Can contain methods, properties, indexers, and events
- Private interface implementations

```
interface IDataBound
{
   void Bind(IDataBinder binder);
}

class EditBox: Control, IDataBound
{
   void IDataBound.Bind(IDataBinder binder) {...}
}
```

Enums

- All enums derive from `System.Enum`
- Strongly typed
  - No implicit conversions to/from int
  - Operators: +, -, ++, --, &, |, ^, ~
- Can specify underlying type
  - Byte, short, int, long

```
enum Color: byte {
   Red   = 1,
   Green = 2,
   Blue  = 4,
   Black = 0,
   White = Red | Green | Blue
}

Color c = Color.Black;
Console.WriteLine(c);                              // 0
Console.WriteLine(c.Format());                     // Black
```

Delegates

- Object oriented function pointers
- Multiple receivers
  - Each delegate has an invocation list
  - Thread-safe + and - operations
- Foundation for events

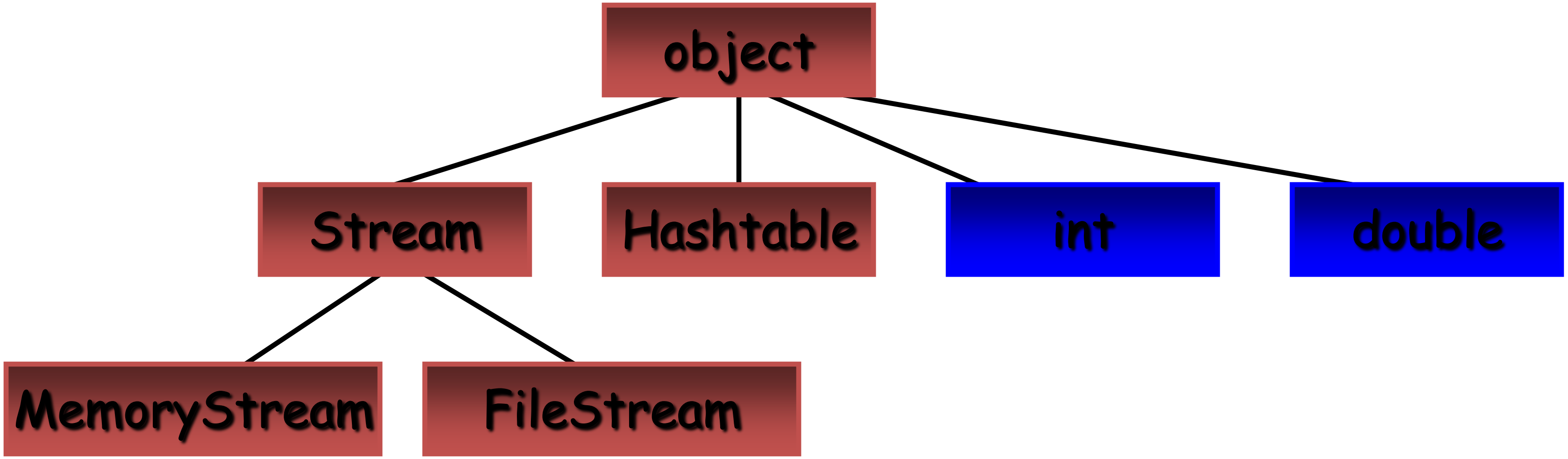
```
delegate void MouseEvent(int x, int y);

delegate double Func(double x);

Func func = new Func(Math.Sin);
double x = func(1.0);
```

Unified Type System

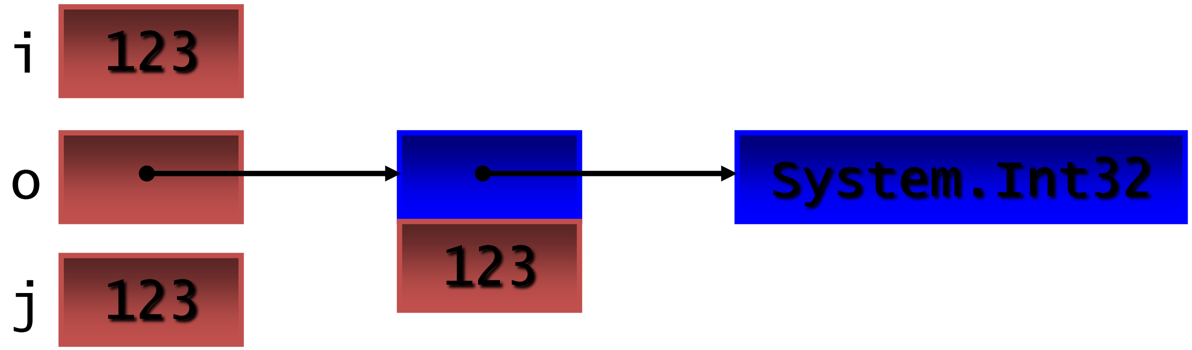
- Everything is an object
  - All types ultimately inherit from object
  - Any piece of data can be stored, transported, and manipulated with no extra work



Unified Type System

- Boxing
  - Allocates box, copies value into it
- Unboxing
  - Checks type of box, copies value out

```
int i = 123;
object o = i;
int j = (int)o;
```



Unified Type System

- Benefits
  - Eliminates “wrapper classes”
  - Collection classes work with all types
  - Replaces OLE Automation's Variant
- Lots of examples in .NET Framework

```
string s = string.Format(
   "Your total was {0} on {1}", total, date);

Hashtable t = new Hashtable();
t.Add(0, "zero");
t.Add(1, "one");
t.Add(2, "two");
```

Types - Unified Type System

- Question: How can we treat value and reference types polymorphically?
  – How does an int (value type) get converted into an object (reference type)?
- Answer: Boxing!
  – Only value types get boxed
  – Reference types do not get boxed

Types - Unified Type System

- Boxing
  - Copies a value type into a reference type (`object`)
  - Each value type has corresponding “hidden” reference type
  - Note that a reference-type copy is made of the value type
    - Value types are never aliased
  - Value type is converted implicitly to `object`, a reference type
    - Essentially an “up cast”

Boxing and Unboxing

- Boxing and Unboxing is one of the key innovations of C# language.
- Instead of requiring the programmer to write wrapper code to convert from stack based memory to heap memory, you just need to assign a value type to an object and C# takes care of allocating the memory in the heap and generating a copy of that on the heap.
- When you assign the object to a stack based int, the value is converted to the stack again.
- This process is what we call Boxing and Unboxing.
- So…
- If an int is boxed, it still knows it’s an int.

Types - Unified Type System

- Unboxing
  - Inverse operation of boxing
  - Copies the value out of the box
    - Copies from reference type to value type
  - Requires an explicit conversion
    - May not succeed (like all explicit conversions)
    - Essentially a “down cast”

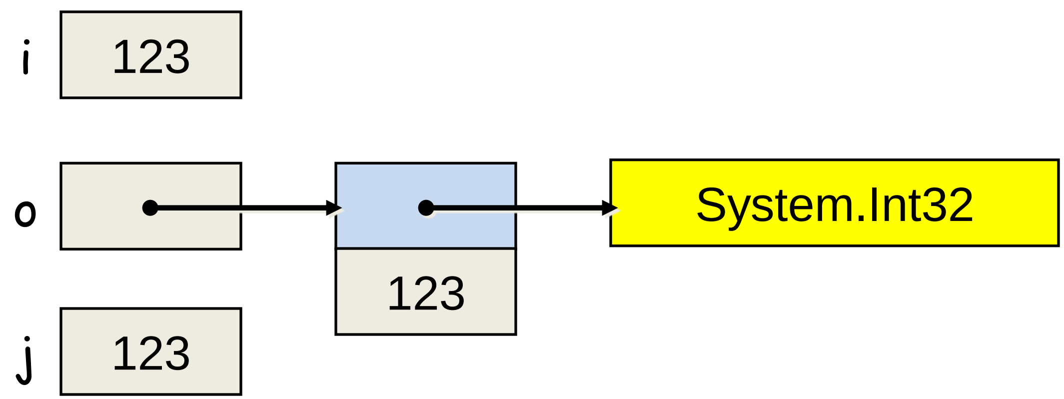
Types - Unified Type System

- Boxing and unboxing

```
int i = 123;

object o = i;

int j = (int)o;
```



Types - Unified Type System

- Benefits of boxing
  - Enables polymorphism across all types
  - Collection classes work with all types
  - Eliminates need for wrapper classes
- Lots of examples in .NET Framework

```
Hashtable t = new Hashtable();
t.Add(0, "zero");
t.Add(1, "one");
t.Add(2, "two");
```

```
string s = string.Format(
  "Your total was {0} on {1}",
  total, date);
```

Types - Unified Type System

- Disadvantages of boxing
  - Performance cost
- The need for boxing will decrease when the CLR supports generics (similar to C++ templates)

Component Development

- What defines a component?
  - Properties, methods, events
  - Integrated help and documentation
  - Design-time information
- C# has first class support
  - Not naming patterns, adapters, etc.
  - Not external files
- Components are easy to build and consume

Properties

- Properties are “smart fields”
  - Natural syntax, accessors, inlining

```
public class Button: Control
{
   private string caption;

   public string Caption {
      get {
         return caption;
      }
      set {
         caption = value;
         Repaint();
      }
   }
}
```

```
Button b = new Button();
b.Caption = "OK";
String s = b.Caption;
```

Indexers

- Indexers are “smart arrays”
  - Can be overloaded

```
public class ListBox: Control
{
   private string[] items;

   public string this[int index] {
      get {
         return items[index];
      }
      set {
         items[index] = value;
         Repaint();
      }
   }
}
```

```
ListBox listBox = new ListBox();
listBox[0] = "hello";
Console.WriteLine(listBox[0]);
```

Events - Sourcing

- Define the event signature

```
public delegate void EventHandler(object sender, EventArgs e);
```

- Define the event and firing logic

```
public class Button
{
   public event EventHandler Click;

   protected void OnClick(EventArgs e) {
      if (Click != null) Click(this, e);
   }
}
```

Events - Handling

- Define and register event handler

```
public class MyForm: Form
{
   Button okButton;

   public MyForm() {
      okButton = new Button(...);
      okButton.Caption = "OK";
      okButton.Click += new EventHandler(OkButtonClick);
   }

   void OkButtonClick(object sender, EventArgs e) {
      ShowMessage("You pressed the OK button");
   }
}
```

Attributes

- How do you associate information with types and members?
  - Documentation URL for a class
  - Transaction context for a method
  - XML persistence mapping
- Traditional solutions
  - Add keywords or pragmas to language
  - Use external files, e.g., .IDL, .DEF
- C# solution: Attributes

Attributes

```
public class OrderProcessor
{
   [WebMethod]
   public void SubmitOrder(PurchaseOrder order) {...}
}

[XmlRoot("Order", Namespace="urn:acme.b2b-schema.v1")]
public class PurchaseOrder
{
   [XmlElement("shipTo")]  public Address ShipTo;
   [XmlElement("billTo")]  public Address BillTo;
   [XmlElement("comment")] public string Comment;
   [XmlElement("items")]   public Item[] Items;
   [XmlAttribute("date")]  public DateTime OrderDate;
}

public class Address {...}

public class Item {...}
```

Attributes

- Attributes can be
  - Attached to types and members
  - Examined at run-time using reflection
- Completely extensible
  - Simply a class that inherits from System.Attribute
- Type-safe
  - Arguments checked at compile-time
- Extensive use in .NET Framework
  - XML, Web Services, security, serialization, component model, COM and P/Invoke interop, code configuration…

PSK SERVICE OY

XML Comments

```
class XmlElement
{
   /// <summary>
   ///    Returns the attribute with the given name and
   ///    namespace</summary>
   /// <param name="name">
   ///    The name of the attribute</param>
   /// <param name="ns">
   ///    The namespace of the attribute, or null if
   ///    the attribute has no namespace</param>
   /// <return>
   ///    The attribute value, or null if the attribute
   ///    does not exist</return>
   /// <seealso cref="GetAttr(string)"/>
   ///
   public string GetAttr(string name, string ns) {
      ...
   }
}
```

Statements And Expressions

- High C++ fidelity
- If, while, do require bool condition
- goto can't jump into blocks
- Switch statement
  - No fall-through, "goto case" or "goto default"
- foreach statement
- Checked and unchecked statements
- Expression statements must do work

```
void Foo() {
   i == 1;    // error
}
```

foreach Statement

- Iteration of arrays

```
public static void Main(string[] args) {
   foreach (string s in args) Console.WriteLine(s);
}
```

- Iteration of user-defined collections

```
foreach (Customer c in customers.OrderBy("name")) {
   if (c.Orders.Count != 0) {
      ...
   }
}
```

Parameter Arrays

- Can write “printf” style methods
  - Type-safe, unlike C++

```
void printf(string fmt, params object[] args) {
   foreach (object x in args) {
      ...
   }
}
```

```
printf("%s %i %i", str, int1, int2);

object[] args = new object[3];
args[0] = str;
args[1] = int1;
Args[2] = int2;
printf("%s %i %i", args);
```

Operator Overloading

- First class user-defined data types
- Used in base class library
  - Decimal, DateTime, TimeSpan
- Used in UI library
  - Unit, Point, Rectangle
- Used in SQL integration
  - SQLString, SQLInt16, SQLInt32, SQLInt64, SQLBool, SQLMoney, SQLNumeric, SQLFloat…

Operator Overloading

```
public struct DBInt
{
   public static readonly DBInt Null = new DBInt();

   private int value;
   private bool defined;

   public bool IsNull { get { return !defined; } }

   public static DBInt operator +(DBInt x, DBInt y) {...}

   public static implicit operator DBInt(int x) {...}
   public static explicit operator int(DBInt x) {...}
}
```

```
DBInt x = 123;
DBInt y = DBInt.Null;
DBInt z = x + y;
```

Versioning

- Problem in most languages
  - C++ and Java produce fragile base classes
  - Users unable to express versioning intent
- C# allows intent to be expressed
  - Methods are not virtual by default
  - C# keywords “virtual”, “override” and “new” provide context
- C# can't guarantee versioning
  - Can enable (e.g., explicit override)
  - Can encourage (e.g., smart defaults)

Versioning

```
class Base                          // version 2
{
   public virtual void Foo() {
      Console.WriteLine("Base.Foo");
   }
}
```

```
class Derived: Base                 // version 2b
{
   public override void Foo() {
      base.Foo();
      Console.WriteLine("Derived.Foo");
   }
}
```

Conditional Compilation

- #define, #undef
- #if, #elif, #else, #endif
  - Simple boolean logic
- Conditional methods

```
public class Debug
{
   [Conditional("Debug")]
   public static void Assert(bool cond, String s) {
      if (!cond) {
         throw new AssertionException(s);
      }
   }
}
```

Unsafe Code

- Platform interoperability covers most cases
- Unsafe code
  - Low-level code “within the box”
  - Enables unsafe casts, pointer arithmetic
- Declarative pinning
  - Fixed statement
- Basically “inline C”

```
unsafe void Foo() {
   char* buf = stackalloc char[256];
   for (char* p = buf; p < buf + 256; p++) *p = 0;
   ...
}
```

Unsafe Code

```
class FileStream: Stream
{
   int handle;

   public unsafe int Read(byte[] buffer, int index, int count) {
      int n = 0;
      fixed (byte* p = buffer) {
         ReadFile(handle, p + index, count, &n, null);
      }
      return n;
   }

   [dllimport("kernel32", SetLastError=true)]
   static extern unsafe bool ReadFile(int hFile,
      void* lpBuffer, int nBytesToRead,
      int* nBytesRead, Overlapped* lpOverlapped);
}
```

Statements - Synchronization

- Multi-threaded applications have to protect against concurrent access to data
  - Must prevent data corruption
- The `lock` statement uses an instance to provide mutual exclusion
  - Only one `lock` statement can have access to the same instance
  - Actually uses the .NET Framework `System.Threading.Monitor` class to provide mutual exclusion - > see section threads in C#

Statements - Synchronization

```
public class CheckingAccount {
  decimal balance;
  public void Deposit(decimal amount) {
    lock (this) {
      balance += amount;
    }
  }
  public void Withdraw(decimal amount) {
    lock (this) {
      balance -= amount;
    }
  }
}
```

Statements - `using` Statement

- C# uses automatic memory management (garbage collection)
  - Eliminates most memory management problems
- However, it results in non-deterministic finalization
  - No guarantee as to when and if object destructors are called

Statements

`using` Statement

- Objects that need to be cleaned up after use should implement the `System.IDisposable` interface
  - One method: `Dispose()`
- The `using` statement allows you to create an instance, use it, and then ensure that `Dispose` is called when done
  - `Dispose` is guaranteed to be called, as if it were in a `finally` block

Statements - `using` Statement

```
public class MyResource : IDisposable {
  public void MyResource() {
    // Acquire valuble resource
  }
  public void Dispose() {
    // Release valuble resource
  }
  public void DoSomething() {
     ...
  }
}
```

```
using (MyResource r = new MyResource()) {
  r.DoSomething();
}                          // r.Dispose() is called
```

Statements - checked and unchecked Statements

- The checked and unchecked statements allow you to control overflow checking for integral-type arithmetic operations and conversions
- checked forces checking
- unchecked forces no checking
- Can use both as block statements or as an expression
- Default is unchecked
- Use the /checked compiler option to make checked the default

PSK SERVICE OY

Statements - Basic Input/Output Statements

- Console applications
  - `System.Console.WriteLine();`
  - `System.Console.ReadLine();`
- Windows applications
  - `System.WinForms.MessageBox.Show();`

```
string v1 = "some value";
MyObject v2 = new MyObject();
Console.WriteLine("First is {0}, second is {1}",
                               v1, v2);
```

PSK SERVICE OY

Questions?