Softwareudvikling 2017

software development

C# Programming

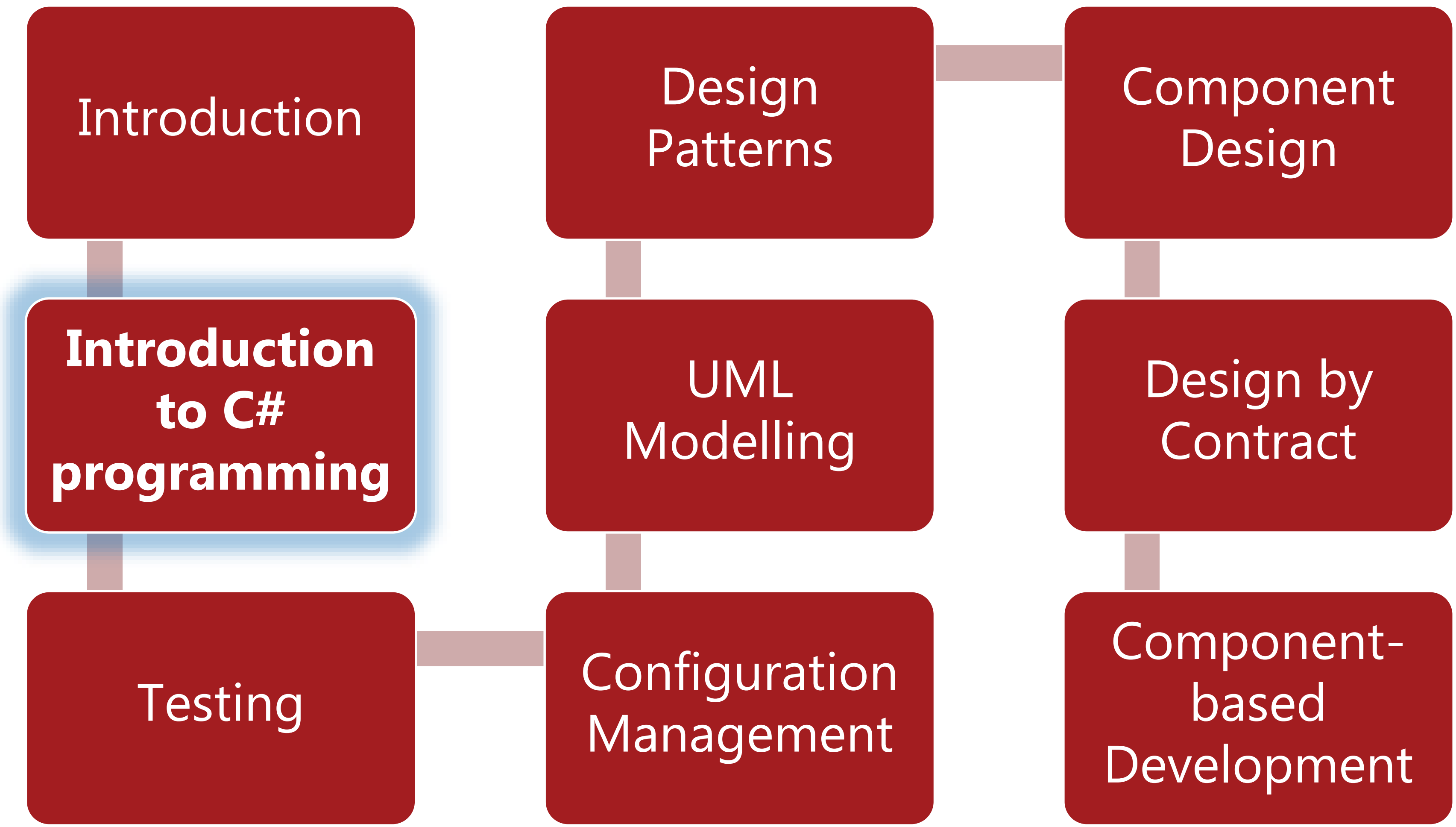
Boris Düdder, Datalogisk Institut
7.2.2017

UNIVERSITY OF COPENHAGEN

Based on slides from

- Stephen Turner
- Luca Bolognese
- Anders Hejlsberg
- Peter Sestoft

Course outline block 3: Lectures



Learning goals

- Understanding concepts of Object-Orientation (O-O)
- Understanding of basics of object-oriented programming (OOP)
- Understanding of language concepts of C# 2.0

Literature

Please read (and constantly reread) our literature!

- Peter Sestoft, Henrik Hansen, C# Precisely, 2nd ed., MIT Press, 2011

For deeper understanding and further topics see

- Andrew Troelsen and Philip Japikse. C# 6.0 and the .NET 4.6 Framework. 7th ed., Apress, 2016

We will link tutorial videos on C# in Absalon

Agenda

- **Object-Orientation (O-O)**
- Object-Oriented Programming (OOP)
- Programming language C# (2.0)
- Component-based Programming with C#
- Outlook on current and future C#

Object Oriented Programming

- Over time, data abstraction has become essential as programs became complicated.
- Benefits:
  1. Reduce conceptual load (minimum detail).
  2. Fault containment.
  3. Independent program components.
     (difficult in practice).

- Code reuse possible by extending and refining abstractions.

Object Oriented Programming

- A methodology of programming

- Four (Five ?) major principles:

  1. Data Abstraction.
  2. Encapsulation.
  3. Information Hiding.
  4. Polymorphism (dynamic binding).
  5. Inheritance. (particular case of polymorphism ?)

Object-Oriented Programming

Object-oriented programming is a programming methodology characterized by the following concepts:

1. Data Abstraction: problem solving via the formulation of abstract data types (ADT's).

2. Encapsulation: the proximity of data definitions and operation definitions.

3. Information hiding: the ability to selectively hide implementation details of a given ADT.

4. Polymorphism: the ability to manipulate different kinds of objects, with only one operation.

5. Inheritance: the ability of objects of one data type, to inherit operations and data from another data type. Embodies the "*is a*" notion: a horse is a mammal, a mammal is a vertebrate, a vertebrate is a lifeform.

O-O Principles and C# Constructs

O-O Concept		C# Construct(s)
Abstraction	⟷	Classes
Encapsulation	⟷	Classes
Information Hiding	⟷	Public and Private Members
Polymorphism	⟷	Operator overloading, generics, virtual functions
Inheritance	⟷	Derived Classes

Agenda

- Object-Orientation (O-O)
- **Object-Oriented Programming (OOP)**
- Programming language C# (2.0)
- Component-based Programming with C#
- Outlook on current and future C#

O-O is a different Paradigm

- Central questions when programming.
  - Imperative Paradigm:
    - What to do next ?
  - Object-Oriented Programming
    - What does the object do ? (vs. how)
- Central activity of programming:
  - Imperative Paradigm:
    - Get the computer to do something.
  - Object-Oriented Programming
    - Get the object to do something.

Concept: An object has behaviors

- In old style programming, you had:
  - data, which was completely passive
  - functions, which could manipulate any data
- An object contains both data and methods that manipulate that data
  - An object is *active,* not passive; it *does* things
  - An object is *responsible* for its own data
    - But: it can *expose* that data to other objects

Concept: An object has state

- An object contains both data and methods that manipulate that data
  - The data represent the state of the object
  - Data can also describe the relationships between this object and other objects
- Example: A `CheckingAccount` might have
  - A `balance` (the internal state of the account)
  - An `owner` (some object representing a person)

Concept: Classes describe objects

- Every object belongs to (is an instance of) a class
- An object may have fields, or variables
  - The class describes those fields
- An object may have methods
  - The class describes those methods
- A class is like a template, or cookie cutter
  - You use the class's constructor to make objects

Concept: Classes are like Abstract Data Types

- An Abstract Data Type (ADT) bundles together:
  - some data, representing an object or "thing"
  - the operations on that data
- The operations defined by the ADT are the *only* operations permitted on its data
- Example: a `CheckingAccount`, with operations `deposit`, `withdraw`, `getBalance`, etc.
- Classes enforce this bundling together
  - *If* all data values are private, a class can also enforce the rule that its defined operations are the only ones permitted on the data

Concept: Classes form a hierarchy

- Classes are arranged in a treelike structure called a hierarchy
- The class at the root is named `Object`
- Every class, except `Object`, has a superclass
- A class may have several ancestors, up to `Object`
- When you define a class, you specify its superclass
  - If you don't specify a superclass, `Object` is assumed
- Every class may have one or more subclasses

Concept: Objects inherit from superclasses

- A class describes fields and methods
- Objects of that class have those fields and methods
- But an object *also* inherits:
  - the fields described in the class's superclasses
  - the methods described in the class's superclasses
- A class is *not* a complete description of its objects!

Concept: Objects must be created

- `int n;` does two things:
  - It declares that `n` is an integer variable
  - It allocates space to hold a value for `n`
  - For a primitive, this is all that is needed
- `Employee secretary;` also does two things
  - It declares that `secretary` is type `Employee`
  - It allocates space to hold a *reference* to an Employee
  - For an object, this is ***not*** all that is needed
- `secretary = new Employee ( );`
  - This allocate space to hold a *value* for the Employee
  - Until you do this, the Employee is `null`

Notation: How to declare and create objects

```
Employee secretary; // declares secretary
secretary = new Employee (); // allocates space
Employee secretary = new Employee(); // does both
```

- But the secretary is still "blank" (`null`)

```
secretary.name = "Adele";  // dot notation
secretary.birthday ();  // sends a message
```

Concept: `this` object

- Inside a class, no dots are necessary, because
  - you are working on `this` object
- If you wish, you can make it explicit:

```
class Person { ... this.age = this.age + 1; ...}
```

- `this` is like an extra parameter to the method
- You usually don't need to use `this`

Concept: A variable can hold subclass objects

- Suppose **B** is a subclass of **A**
  - **A** objects can be assigned to **A** variables
  - **B** objects can be assigned to **B** variables
  - **B** objects can be assigned to **A** variables, but
  - **A** objects can *not* be assigned to **B** variables
    - Every **B** is also an **A** *but* not every **A** is a **B**
- You can cast: `bVariable = (B) aObject;`
  - In this case, C# does a runtime check

Concept: Methods can be overridden

```
class Bird : Animal {
   void virtual fly (String destination)
     {
      location = destination;
     }
}

class Penguin : Bird {
   void override fly (String whatever)
     { }
}
```

- So birds can fly. Except penguins.

Concept: Don't call functions, send messages

```
Bird someBird = pingu;
someBird.fly ("South America");
```

- Did `pingu` actually go anywhere?
  - You sent the message `fly(...)` to `pingu`
  - If `pingu` is a penguin, he ignored it
  - Otherwise he used the method defined in `Bird`
- You did *not* directly call any method
  - You cannot tell, without studying the program, which method actually gets used
  - The same statement may result in different methods being used at different times

Concept: Constructors make objects

- Every class has a constructor to make its objects
- Use the keyword **new** to call a constructor

```
secretary = new Employee ( );
```

- You can write your own constructors; but if you don't,
- C# provides a default constructor with no arguments
  - simply invokes the parameterless constructor of the direct base class. If none provided, then an exception is raised.
  - If this is good enough, you don't need to write your own
- The syntax for writing constructors is almost like that for writing methods

Syntax for constructors

- *Do not* use a return type and a name; use *only* the class name
- You can supply arguments

```
Employee (String theName, double theSalary)
{
   name = theName;
   salary = theSalary;
}
```

Internal workings: Constructor chaining

- If an `Employee` is a `Person`, and a `Person` is an `Object`, then when you say `new Employee ()`
  - The `Employee` constructor calls the `Person` constructor
  - The `Person` constructor calls the `Object` constructor
  - The `Object` constructor creates a new `Object`
  - The `Person` constructor adds its own stuff to the `Object`
  - The `Employee` constructor adds its own stuff to the `Person`

Concept: You can control access

```
class Person {
    public String name;
    private String age;
    protected double salary;
    internal void birthday { age++; }
}
```

- Each object is responsible for its own data
- Access control lets an object protect its data *and* its methods
- Access control is the subject of a different lecture

Concept: *Classes* can have fields and methods

- Usually a class describes fields (variables) and methods for its objects (instances)
  - These are called instance variables and instance methods
- A class can have its own fields and methods
  - These are called class variables and class methods
- There is exactly *one* copy of a class variable, not one per object
- Use the special keyword `static` to say that a field or method belongs to the class instead of to objects

Agenda

- Object-Orientation (O-O)
- Object-Oriented Programming (OOP)
- **Programming language C# (2.0)**
- Component-based Programming with C#
- Outlook on current and future C#

C#

C# is a

- simple,
- modern,
- general-purpose,
- object-oriented programming language
- component-oriented programming language

developed by Microsoft within its .NET initiative led by Anders Hejlsberg.

Important Features of C#

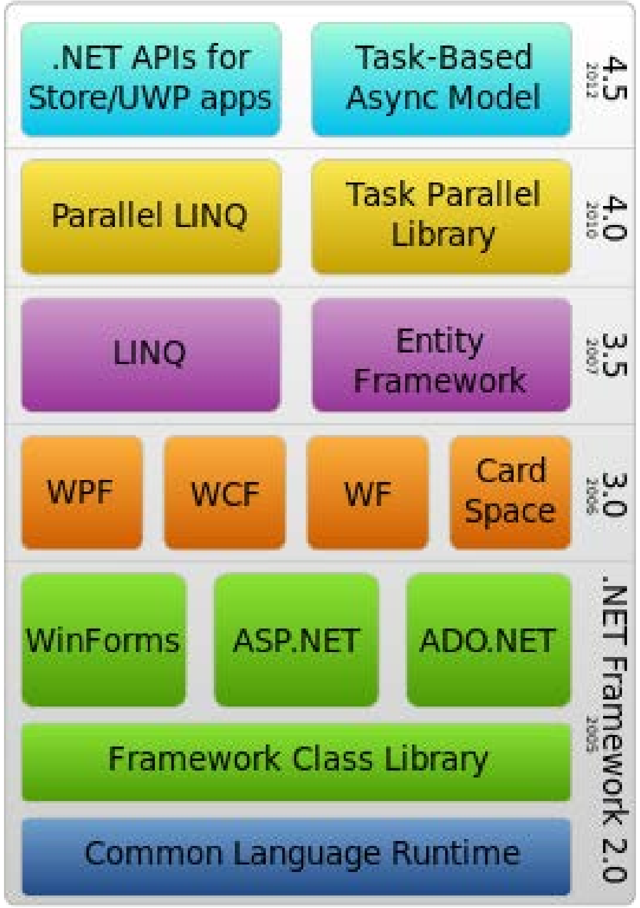
- Follows tradition of C and C++
- Strong resemblance with Java
- Features
  - Boolean Conditions
  - Automatic Garbage Collection
  - Standard Library
  - Assembly Versioning
  - Properties and Events
- Features ctd.
  - Delegates and Events Management
  - Easy-to-use Generics
  - Indexers
  - Conditional Compilation
  - Simple Multithreading
  - LINQ and Lambda Expressions
  - Integration with Windows

C# and .NET

- Languages like C# are not isolated entities
- They interoperate in two ways:
  - By being part of a system written in more than one language
  - By accessing services and operating on a distributed environment
- Requires support from run time:
  - .NET and the Common Language Runtime
- .NET is *many* things, in particular binary object access
- C# interoperates with .NET

.NET Framework

- Common Language Runtime (CLR)
  - Application virtual machine for .NET
- Framework Class Library (FCL)
  - Language-interoperable class library
- .NET Framework
  - predominant implementation of .NET technologies
- .NET Core
  - targets cross-platform and cloud-based workloads
- Mono
  - Open-source Ecma standard-compliant, .NET Framework-compatible set of tools



The Class Libraries

- The common classes used in many programs
  - like Java Class Library
  - offer various functionality
  - e.g.
    - System.Console.WriteLine
    - XML, Networking, Filesystem, Crypto, containers
  - Can inherit from many of these classes
- Many languages run on .NET framework
  - C#, C++, J#, Visual Basic
  - even have Python (see IronPython)

Assemblies

- Code contained in files called “assemblies”
  - Code and metadata
  - .dll as before (DLL=dynamic linked library)
  - to run: `public static void Main(string[] args)`
  - types
    - private: local directory, not accessible by others
    - shared: well-known location, can be GAC
    - strong names: uses cryptography for signatures
    - then can add some versioning and trust

MSBuild

- Build system for .NET projects
- Compiles code to assemblies
- Independent of Visual Studio
- XML based

Example:

```
<?xml version="1.0" encoding="utf-8" ?>
<Project
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Target Name="Build">
    <Message Text="Building msbuildintro" />
    <MSBuild Projects="msbuildintro.csproj" Targets="Build" />
  </Target>
</Project>
```

Project Files

- Define what the build tool has to include
  - Code (C#, VB.NET, …)
  - Ressources (e.g. images)
- Defines references to assemblies
- Defines the target platform
- Defines multiple configurations (build parameters)
- XML-based

Solution Files

- Solution files bind multiple .NET projects to a solution
- Not XML-based

```
Microsoft Visual Studio Solution File, Format Version
12.00

# Visual Studio 2012

Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
"GameConcepts", "GameConcepts\GameConcepts.csproj",
"{397821A7-31C9-4275-893B-C24DED40FEA4}"

EndProject

Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") =
"GalagaGame", "GalagaGame\GalagaGame.csproj",
"{6C945B67-0D2F-4FB4-9DAA-657FDB3FDD72}"

EndProject
```

Simple Stuff

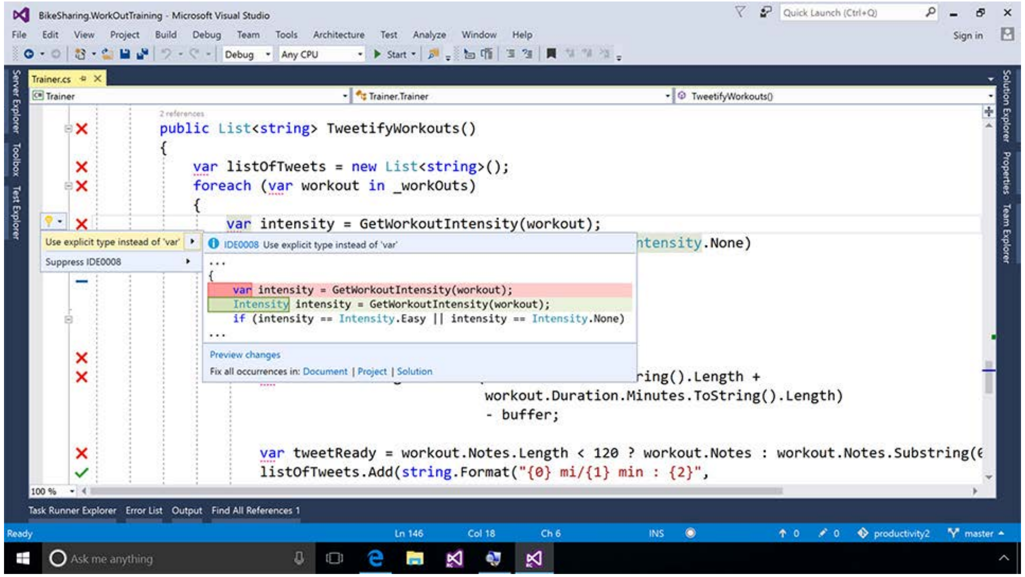
- Most of C# is *pretty* similar to languages you are used to from F#:
  - Declarations
  - Expressions
  - Assignment and control statements
- Other elements are *quite* similar:
  - Classes
  - Functions
  - Polymorphism

Environment (Library)

- Common Language Runtime (CLR)
- The .NET Framework Class Library
- Common Language Specification
- Common Type System
- Metadata and Assemblies
- Windows Forms
- ASP.NET and ASP.NET AJAX
- ADO.NET
- Windows Workflow Foundation (WF)
- Windows Presentation Foundation (WPF)
- Windows Communication Foundation (WCF)
- LINQ

Integrated Environments (IDEs)

- Windows platform (native)
  - Visual Studio Professional / Enterprise (2015 / 2017 RC)
  - Visual Studio Community
- Linux & Mac (using Mono)
  - Visual Studio Code
  - Visual Studio for Mac (2017 RC)
  - MonoDevelop
  - SharpDevelop



Source: Microsoft Corp.

Program Structure

A C# program constists of the following parts

- Namespaces
  - Contain types and other namespaces
- Type declarations
  - class, struct, interface, enum and delegate
- Members
  - constant, field, method, property, indexer, event, operator, constructor, destructor (finalizer)
- Organization
  - No header files, code written “in-line”
  - No declaration order dependence

„Hello-World“ program in C#

```
using System;
namespace HelloWorldApplication
{
    0 references
    class HelloWorld
    {
        0 references
        static void Main(string[] args)
        {
            /* my first program in C# */
            Console.WriteLine("Hello World");
            Console.ReadKey();
        }
    }
}
```

Nota bene

- C# is case sensitive.
- All statements and expression must end with a semicolon (;).
- The program execution starts at the Main method.
- Unlike Java, program filename could be different from the class name.

C# Basic Syntax

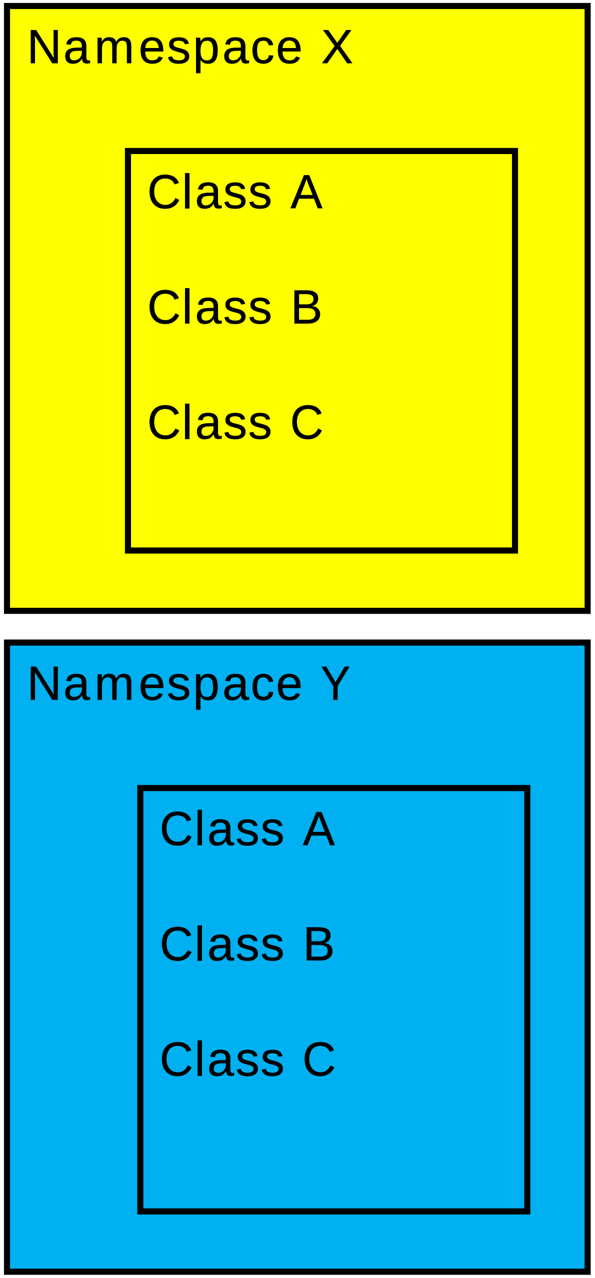
- `using` keyword
  - used for including (multiple) namespaces in program.
- `class` keyword
  - used for declaring a class.
- Comments in C#
  - Multiline comments `/* … */`
  - Single line comment `// …`
- Member variables
  - attributes or data members of a class, used for storing data
- Member functions
  - set of statements that perform a specific task.

Namespaces

- Permits isolation of names
- Can be nested
- Access via fully qualified names

`X.A`

`Y.A`



C# Keywords

Reserved Keywords						
abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	in (generic modifier)	int
interface	internal	is	lock	long	namespace	new
null	object	operator	out	out (generic modifier)	override	params
private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct
switch	this	throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using	virtual	void
volatile						
Contextual Keywords						
add	alias	ascending	descending	dynamic	from	get
global	group	into	join	let	orderby	partial (type)
partial (method)	remove	select	set			

Type System

- Type should be consistent:
  - Predefined and user-defined
- All C# types derive from `System.Object` (unified)
- Single rooted hierarchy
- Provides four standard methods:
  - bool Equals — Same object (ref) or same value (val)
  - int GetHashCode
  - Type GetType — Retrieve object type (reflection)
  - String ToString — Retrieve object type (default)

Types of Types

- Value types and reference types
- Value types:
  - Program variables have a value
  - Space allocated on stack
  - Cannot be **`null`**

- Reference types:
  - Program variable is just a reference
  - Allocated space on stack
  - Reference is a "type-safe" pointer
  - Data space allocated on heap
  - May be **`null`**

Stack

(more details in Computersystemer)

The place where

- arguments of a function call are stored
- registers of the calling function are saved
- local data of called function is allocated
- called function leaves result for calling function

Supports recursive function calls

Stack – a linear data structure in which items are added and removed in *last-in, first-out* order.

Heap

- A place for allocating memory that is **not** part of *last-in, first-out* discipline
- I.e., dynamically allocated data structures that survive function calls
  - E.g., strings in C#
  - `new` objects in C#

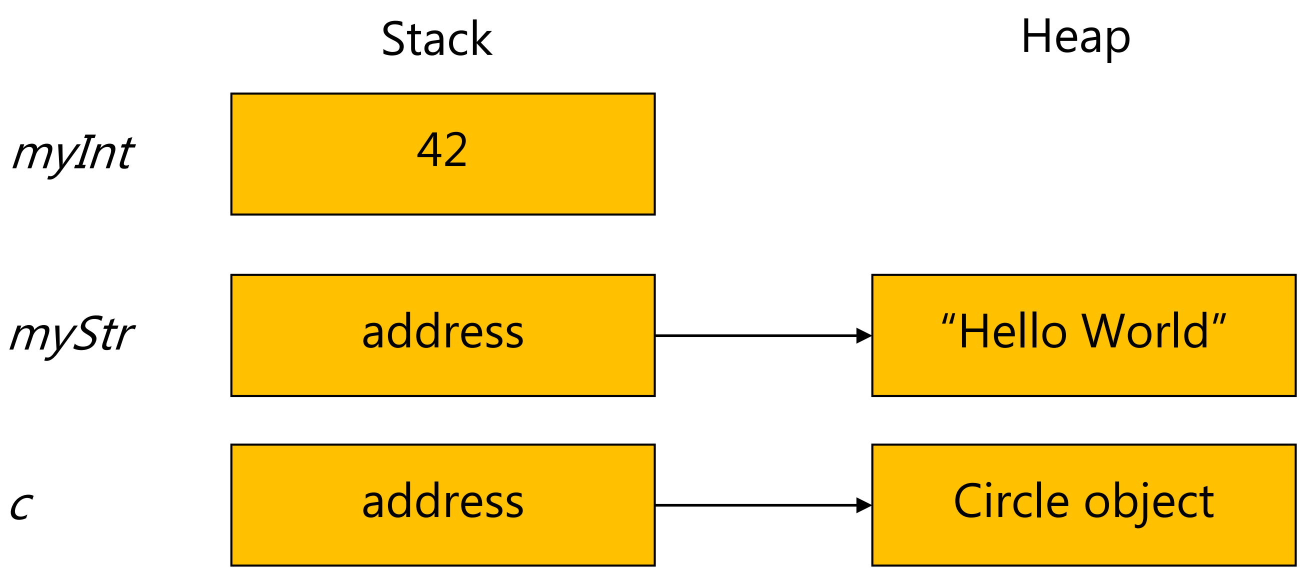
Value types and reference types
(C# Precisely section 5.1, examples 84 and 86)

- A C# type is either a *reference type* (class, interface, array type) or a *value type* (int, double, . . .).
  - A value (object) of reference type is always stored in the managed (garbage-collected) heap.
  - Assignment to a variable of reference type copies only the reference.
  - A value of value type is stored in a local variable or parameter, or inline in an array or object or struct value.
- Assignment to a variable of value type copies the entire value.
- Just as in Java. But in C#, there are also user defined value types, called struct types (as in C/C++)

Value Type vs. Reference Type

Note the "special" status of primitive types

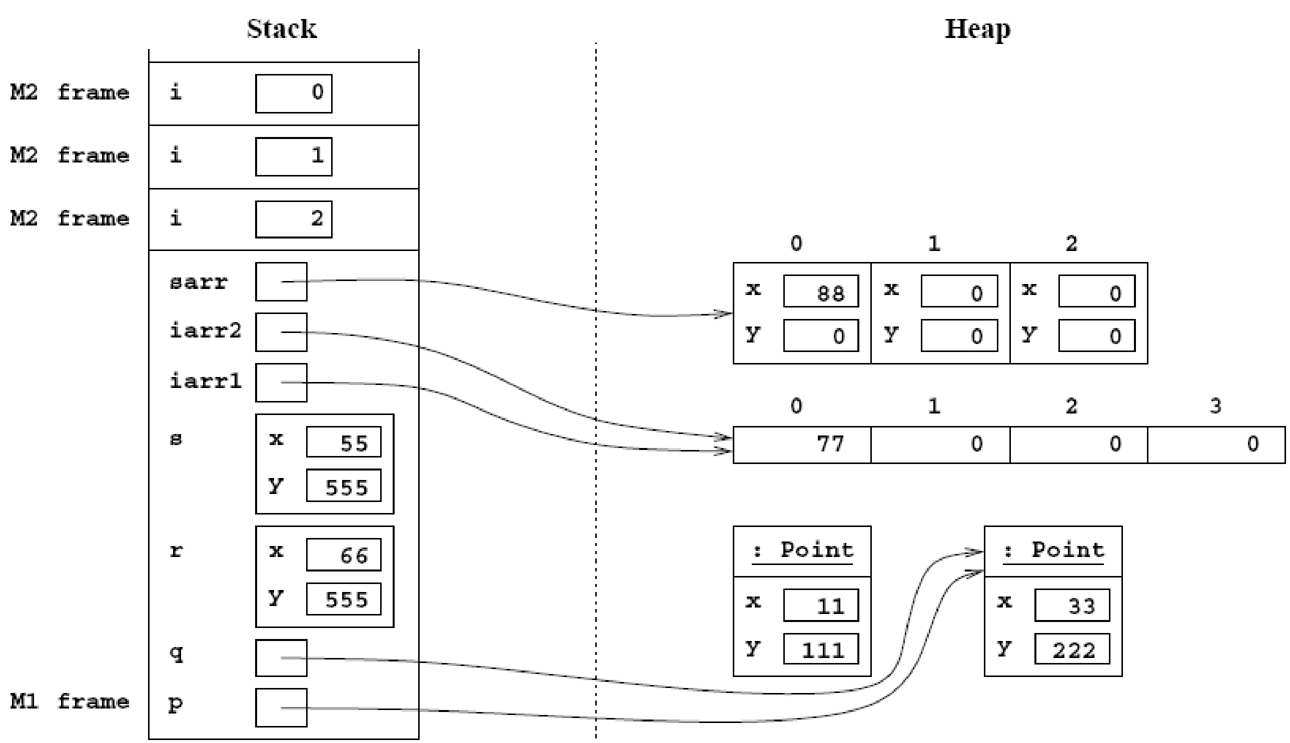
```
System.Int32  myInt = 42;
System.String myStr = "Hello World";
Circle c;
c = new Circle(...);
```



Be careful with deallocation of the space

The machine model (C# Precisely example 64)

- Class instances (objects) are individuals in the heap.
- Struct instances are in the stack, or inline in other structs, objects or arrays.



Value & Reference Types

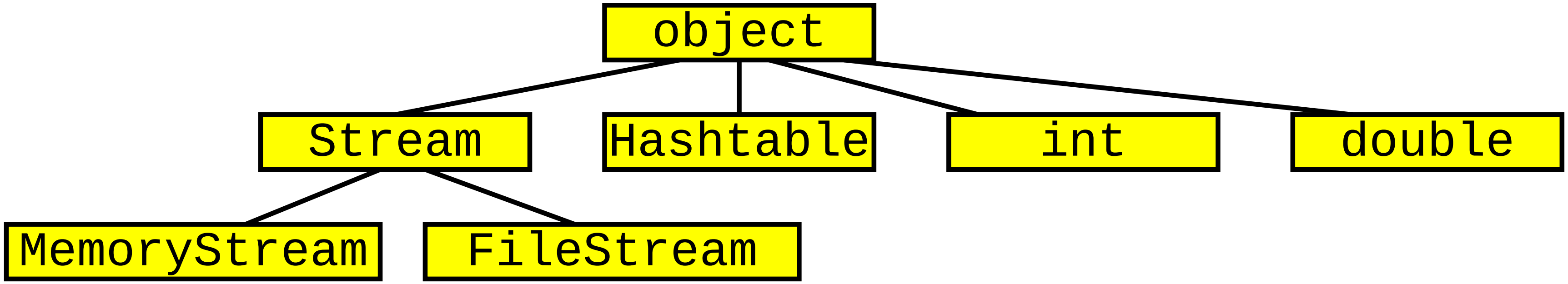
- Value types
  - Primitives `int i; float f;`
  - Enums `enum State { Off, On }`
  - Structs `struct Point { int x, y; }`
- Reference types
  - Classes `class Foo: Bar, IFoo { … }`
  - Interfaces `interface IFoo: IBar { … }`
  - Arrays `string[]s = new string[10];`
  - Delegates `delegate void Empty();`

Unified Type System

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

Benefits of a Unified Type System

- Eliminates the need for wrapper classes
- Collection classes work with all types
- Replaces OLE Automation's Variant
- Simple, consistent programming model



Predefined Types

- C# predefined types
  - Reference `object, string`
  - Signed `sbyte, short, int, long`
  - Unsigned `byte, ushort, uint, ulong`
  - Character `char`
  - Floating-point `float, double, decimal`
  - Logical `bool`
- Predefined types are simply aliases for system-provided types
  - For example, `int = System.Int32`

Classes

- Inheritance
  - Single base class
  - 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

Abstract Classes

- Classes marked with keyword **`abstract`**
- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- Abstract classes used as base classes in hierarchy.
- Subclasses can have instances
- So why use them?
  - Abstract classes express abstract concepts (not a real thing)

```
abstract class C {
      public abstract void M();
}
```

Interfaces

- An interface is a reference type with no implementation
- Specify methods, properties, indexers & events

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

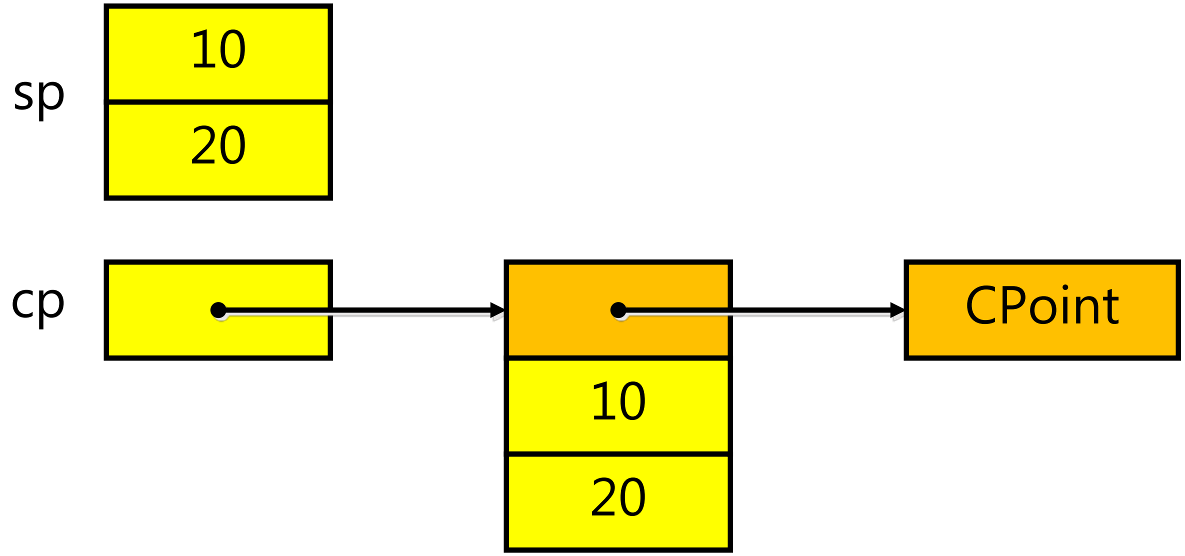
Structs

- Like classes, except
  - Stored on stack or in-line, not heap allocated
  - Assignment copies data, not reference
  - Derive from System.ValueType
  - Can not inherit or be inherited
  - Can implement multiple interfaces
- Ideal for light weight objects
  - Complex, point, rectangle, color
  - int, float, double, etc., are all structs
- Benefits
  - No heap allocation, so fast!
  - 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);
```



C# Memory Management

- Static vs. dynamic
- Dynamic storage—stack and heap
- Stack (Dynamic):
  - Managed algorithmically by implementation of function calls
- Heap (Dynamic)
  - Mostly managed by system
  - Provision for management by programmer

C# Memory Management

- Allocation using **new**
- Deallocation by *Garbage Collection*
- Garbage collection:
  - Tracks objects that are accessible
  - Frees storage associated with objects that are inaccessible
  - Garbage collector is a system provided service that runs periodically
  - Deals with fragmentation

Garbage Collector Pros & Cons

- Pros:
  - Programmer does not have to implement
  - Memory management done right
- Cons:
  - No guarantee when it runs, hence no control
  - Takes processor resources
  - Does not delete storage if it is still reachable even if you don’t want it···
  - Memory leaks *can* (and *do*) still occur

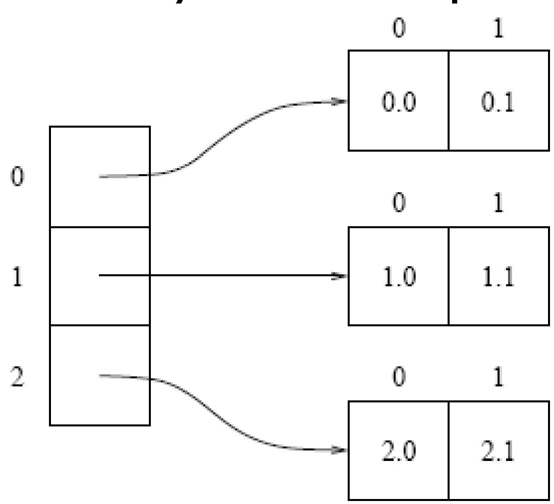
Some specifics of C#

- Object destruction via **`Object.Finalize`**:
  - Inherited from Object type
  - Override to destroy object as desired
- Garbage collector available via GC class:
  - Runs via separate thread
  - Various methods available for access
  - e.g., **`GC.collect()`**
- Pointers—yes, they are provided:
  - Syntax like C++, code marked unsafe
  - Objects managed by GC or user—pinned

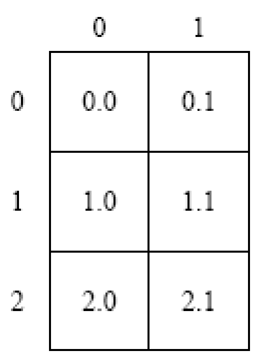
Object can not be moved

Rectangular multi-dimensional arrays (C# Precisely section 9.2.1)

- In Java, a 'multi-dimensional' array is a one-dimensional array of arrays.
- C# in addition has C-style rectangular multi-dimensional arrays.
- This improves memory locality (speed) and reduces memory consumption (space).



Array of arrays (`double[][]`)



Rectangular array (`double[,]`)

Agenda

- Object-Orientation (O-O)
- Object-Oriented Programming (OOP)
- Programming language C# (2.0)
- **Component-based Programming with C#**
- Outlook on current and future C#

1st Class Component Support

- C# is the first “component oriented” language in the C/C++ family
- What defines a component?
  - Properties & events
  - Design-time & runtime information
  - Integrated help & documentation
- C# has first class support
  - Not naming patterns, adapters, etc.
  - Not external header or IDL files
- Easy to build & consume

Properties

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

```
public class Person
{
        private int age;

        public int Age {
                get {
                        return age;
                }
                set {
                        age = value;
                        Party();
                }
        }
}
```

```
// Natural
Person p = new Person();
p.Age = 27;
p.Age ++;
```

```
// Unnatural
Person p = new Person();
p.set_Age(27);
p.set_Age(p.get_Age() + 1);
```

Indexers

- Indexers are "smart arrays"
  - Overloadable for different index signatures

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

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

```
ListBox lb = new ListBox();

lb[0] = "hello";
Console.WriteLine(lb[0]);
```

Delegates

- Object oriented function pointers
  - Actually a method type
- Multiple receivers
  - Each delegate has an invocation list (+= & -= ops)

```
delegate double Func(double x);

static void Main() {
   Func f = new Func(MySin);
   double x = f(1.0);
}

private static double MySin(double x) {
   return Math.Sin(x);
}
```

Events

- A "protected delegate"
  - Owning class gets full access
  - Consumers can only hook or unhook handlers
  - Similar to a property – supported with metadata
- Used throughout the frameworks
- Very easy to extend

Event Sourcing

- Define the event signature

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

- Define the event and firing logic

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

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

Event Handling

- Define and register event handler

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

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

   private void OkButtonClick(object sender, EventArgs e) {
      MessageBox.Show("You pressed the OK button");
   }
}
```

Metadata

- Associate with types and members
  - Design time information
  - Transaction context for a method
  - XML persistence mapping
- Traditional solutions
  - Add keywords or pragmas to language
  - Use external files, e.g., .IDL, .DEF
- Extend the language semantics
  - Expose methods to web services
  - Set transaction context for a method

Attributes

- Attributes can be
  - Attached to any type and its 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
  - Web Services, code security, serialization, XML persistence, component / control model, COM and P/Invoke interop, code configuration…

Attributes

- Appear in square brackets
- Attached to code elements
  - Types, members & parameters

```
[WebService(Namespace="http://microsoft.com/demos/")]
class SomeClass
{
   [WebMethod]
   void GetCustomers() {
   }

   string Test([SomeAttr] string param1) {
   }
}
```

Creating Attributes

- Attributes are simply classes
  - Derived from System.Attribute
  - Class functionality = attribute functionality

```
public class HelpURLAttribute : System.Attribute
{
   public HelpURLAttribute(string url) { … }

   public string URL { get { … } }
   public string Tag { get { … } set { … } }
}
```

Using Attributes

- Just attach it to a class

```
[HelpURL("http://someurl/")]
Class MyClass { … }
```

- Use named parameters

```
[HelpURL("http://someurl/", Tag="ctor")]
Class MyClass { … }
```

- Use multiple attributes

```
[HelpURL("http://someurl/"),
 HelpURL("http://someurl/", Tag="ctor")]
Class MyClass { … }
```

Querying Attributes

- Use reflection to query attributes

```
Type type = typeof(MyClass); // or myObj.GetType()

foreach (Attribute attr in type.GetCustomAttributes())
{
   if (attr is HelpURLAttribute)
   {
      HelpURLAttribute ha = (HelpURLAttribute) attr;
      myBrowser.Navigate(ha.URL);
   }
}
```

Some standard .NET attribute classes

Attribute Name	Targets	Meaning
Flags	Enum type	Print enum value combinations symbolically
Serializable	Class	Instances can be serialized and deserialized
NonSerialized	Field	Field is omitted when class is serialized
AttributeUsage	Class	Permissible targets for this attribute class
Diagnostics.Conditional	Method	Determine when (diagnostic) method should be called
Obsolete	All	Inform users that target is deprecated and should not be used

Productivity Features

- Enums
- foreach statement
- Parameter arrays
- Ref and out parameters
- Overflow checking
- Using statement
- Switch on string
- Operator overloading
- XML comments
- Conditional compilation
- Unsafe code
- Platform Invoke
- COM interop

Enums

- Strongly typed
  - No implicit conversions to/from int
  - Operators: +, -, ++, --, &, |, ^, ~
- Can specify underlying type
  - Byte, short, int, long
- Supported my metadata & reflection

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

foreach Statement

- Iteration of arrays

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

- Iteration of user-defined collections
  - Or any type that supports IEnumerable

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

Parameter Arrays

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

```
static void Main() {
   printf("%s %i %o", "Hello", 29, new Object());

   object[] args = new object[3];
   args[0] = "Hello";
   args[1] = 29;
   args[2] = new Object();
   printf("%s %i %o", args);
}

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

ref and out Parameters

- Use “ref” for in/out parameter passing
- Use “out” to return multiple values
- Must repeat ref/out at call site

```
static void Swap(ref int a, ref int b) {...}

static void Divide(int dividend, int divisor,
     out int result, out int remainder) {...}

static void Main() {
   int x = 1, y = 2;
   Swap(ref x, ref y);
   int r0, r1;
   Divide(3, 2, out r0, out r1);
}
```

Overflow Checking

- Integer arithmetic operations
  - C, C++, Java silently overflow
- checked vs. unchecked contexts
  - Default is unchecked, except for constants
  - Change with "/checked" compiler switch

```
int m0 = checked(x * y);  // check operator

checked {                 // check statement
   int m1 = x * y;
}
```

using Statement

- Acquire, Execute, Release pattern
- Works with any IDisposable object
  - Data access classes, streams, text readers and writers, network classes, etc.

```
Resource res = new Resource(...);
try {
   res.DoWork();
}
finally {
   if (res != null) {
      ((IDisposable)res).Dispose();
   }
}
```

```
using (Resource res = new Resource()) {
   res.DoWork();
}
```

Switch on String

```
Color ColorFromFruit(string s)
{
   switch(s.ToLower())
   {
      case "apple":
         return Color.Red;
      case "banana":
         return Color.Yellow;
      case "carrot":
         return Color.Orange;
      default:
         throw new InvalidArgumentException();
   }
}
```

Operator Overloading

- First class user-defined data types
- Used in base class library
  - Decimal, DateTime, TimeSpan
- Used in the Windows Forms library
  - Point, Size, Rectangle
- Used in the SQL libraries
  - 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;
```

User-defined conversions (C# Precisely section 10.16)

A user-defined conversion is an operator named by the conversion’s target typ
Conversions may be implicit (require no cast) or explicit (require a type cast).
Converting between integers, doubles and fractions is useful:

```
struct Frac : IComparable {
        public readonly long n, d;
        public Frac(long n, long d) { … }
        public static implicit operator Frac(int n) { return new Frac(n, 1); }
        public static implicit operator Frac(long n) { return new Frac(n, 1); }
        public static explicit operator long(Frac r) { return r.n/r.d; }
        public static explicit operator float(Frac r) { return ((float)r.n)/r.d; }
        …
}
```

XML Comments

- /// denotes an XML comment
- Compiler processing:
  - Verifies well-formed XML
  - Verifies parameter names
  - Generates globally unique names, so links can be resolved
- Any XML is okay, can use "standard" tags if you want to
- Enables post-processing to final form

XML Comments

```
class XmlElement
{
   /// <summary>
   /// Returns the attribute with the given name
   /// </summary>
   /// <param name="name">
   /// The name of the attribute
   /// </param>
   /// <return>
   /// The attribute value, or null
   /// </return>
   /// <seealso cref="GetAttr(string)"/>

   public string GetAttribute(string name) {
      ...
   }
}
```

Extending the Type System

- Most users think of two types of objects
  - “Real” objects – Customer, Order, etc.
  - Primitive types – int, float, bool
- Different expectations for each
  - Real objects are more expensive to create
  - Primitives always have a value
  - Primitives have operator support
- Classes and Structs – best of both worlds!
- Natural semantics
  - Operator overloading & User conversions
- Interface support

Rational Numbers (½, ¾, 1½)

```
Rational r1 = new Rational(1,2);
Rational r2 = new Rational(2,1);

Rational r3 = r1.AddRational(r2);

double d = Rational.ConvertToDouble(r3);
```

```
Rational r1 = new Rational(1,2);
Rational r2 = 2;

Rational r3 = r1 + r2;

double d = (double) r3;
```

Rational Number – Class?

- Heap allocated
- Can be null
- "=" assigns reference not value
- Arrays allocate references not values

```
public class Rational
{
   public Rational(int n, int d) { … }
}

…

Rational[] array = new Rational[100];
```

Structs Provide an Answer

- Behavior differences versus Classes
  - Stored in-line, not heap allocated
  - Never null
  - Assignment copies data, not reference
- Implementation differences
  - Always inherit from object
  - Always has a default constructor

Rational Number – Struct

```
public struct Rational
{
   public Rational(int n, int d) { … }

   public int Numerator   { get{…} }
   public int Denominator { get{…} }

   public override string ToString() { … }
}
```

```
Rational r = new Rational(1,2);

string s = r.ToString();
```

Implicit Conversions

- No loss of data

```
public struct Rational
{
   …
   public static implicit operator Rational(int i)
   {
      return new Rational(i,1);
   }
}
```

```
Rational r = 2;
```

Explicit Conversions

- Possible loss of precision and can throw exceptions

```
public struct Rational
{
   …
   public static explicit operator double(Rational r)
   {
      return (double) r.Numerator / r.Denominator;
   }
}
```

```
Rational r = new Rational(2,3);
double d = (double) r;
```

Operator Overloading

- Static operators
- Must take its type as a parameter

```
public struct Rational
{
   …
   public static Rational operator+ (
      Rational lhs, Rational rhs)
   {
      return new Rational( … );
   }
}
```

```
Rational r3 = r1 + r2;

r3 += 2;
```

Equality Operators

- .NET Framework equality support

```
public override bool Equals(object o)
```

- .Equals() should use operator==()

```
public static bool operator== (Rational lhs, Rational rhs)
public static bool operator!= (Rational lhs, Rational rhs)
```

```
if ( r1.Equals(r2) ) { … }    if ( !r1.Equals(r2)) { … }

if ( r1 == r2 ) { … }         if ( r1 != r2 ) { … }
```

Structs and Interfaces

- Structs can implement interfaces to provide additional functionality
- Why? The same reasons classes can!
- Examples
  - System.IComparable
    - Search and sort support in collections
  - System.IFormattable
    - Placeholder formatting

System.IFormattable

- Types can support new formatting options through IFormattable

```
Rational r1 = new Rational(2,4);

Console.WriteLine("Rational {0}", r1);
Console.WriteLine("Rational {0:reduced}", r1);
```

Implementing IFormattable

```
public struct Rational : IFormattable {
   public string Format(
      string formatStr, IServiceObjectProvider isop) {
      s = this.ToString();
      if ( formatStr == “reduced" ) { s = … }
      return s;
   }
}
```

```
Rational r1 = new Rational(2,4);

Console.WriteLine("No Format = {0}", r1);
Console.WriteLine("Reduced Format = {0:reduced}", r1);
```

```
No Format = 2/4
Reduced Format = 1/2
```

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

Language Safety vs. C++

- If, while, do require bool condition
- Goto can't jump into blocks
- Switch statement
  - No fall-through
  - Break, goto <case> or goto default
- Checked and unchecked statements
- Expression statements must do work

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

Versioning

- Overlooked 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
- But C# can't guarantee versioning
  - Can enable (e.g., explicit override)
  - Can encourage (e.g., smart defaults)

Method Versioning in Java

```
class Base    // v2.0
{
 public  int Foo()
 {
 Database.Log("Base.Foo");
 }
}


class Derived extends Base // v1.0
{
 public void Foo()
 {
 System.out.println("Derived.Foo");
 }
}
```

Method Versioning in C#

```
class Base    // v2.0
{
 public virtual int Foo()
 {
 Database.Log("Base.Foo"); return 0;
 }
}


class Derived : Base // v2.0
{
 public override void Foo()
 {
 super.Foo();
 Console.WriteLine("Derived.Foo");
 }
}
```

Non-virtual methods
(C# Precisely section 10.7 and 10.8)

C# has four kinds of (non-abstract) method declarations; the three first ones are known from C++:

- Static: `static void M()`

  A call `C.M()` is evaluated by finding the method `M` in class `C` or a superclass of `C`.

- Non-virtual and non-static: `void M()`

  Assume that `T` is the *compile-time type* of `o`.

  Then a call `o.M()` is evaluated by finding the method `M` in class `T` or a superclass of `T`.

- Virtual and non-static: `virtual void M()`

  Assume that `R` is the *runtime class* of the object that `o` evaluates to.

  Then a call `o.M()` is evaluated by finding the method `M` in class `R` or a superclass of `R`.

- Explicit interface member implementation: `void I.M()` — see C# Precisely section 15.3.

  Implements `M` from interface `I`, which must be a base interface of the enclosing class or struct type.

  Assume that `R` is the *runtime class* of the object `o` evaluates to, and that `I` is the *compile-time type* of `o`.

  Then a call `o.M()` is evaluated by calling method `I.M` in class `R` or a superclass of `R`.

  Useful when a class implements several interfaces that describe distinct methods with the same name.

C#: Virtual and non-virtual method example

```
class A {
  public virtual void m1() { Console.WriteLine("A.m1()"); }
  public void m2() { Console.WriteLine("A.m2()"); }
}

class B : A {
  public override void m1() { Console.WriteLine("B.m1()"); }
  public new void m2() { Console.WriteLine("B.m2()"); }
}

class Override {
  static void Main(string[] args) {
    B b = new B();
    A a = b;
    a.m1();              // B.m1()
    b.m1();              // B.m1()
    a.m2();              // A.m2()
    b.m2();              // B.m2()
  }
}
```

A virtual method call is governed by the *runtime class* of `a`; a non-virtual method call by the *compile-time type*.

The `override` and `new` keywords are needed (their purpose is to prevent accidental overriding or hiding).

Unsafe Code – Pointers

- Developers sometime need total control
  - Performance extremes
  - Dealing with existing binary structures
  - Advanced COM Support, DLL Import
- C# “unsafe” = limited “inline C”
  - Pointer types, pointer arithmetic
  - unsafe casts
  - Declarative pinning (fixed statement)
- Power comes at a price!
  - Unsafe means unverifiable code

Unsafe Code & P/Invoke

```
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);
}
```

C# 2.0

Release September 2005

- Generics
- Anonymous methods
- Nullable value types
- Iterators
- Partial types
- ···and many more

Generics

- Why generics?
  - Type safety, performance, increased sharing
- C# generics vs. Java generics
  - Exact run-time type information vs. erasure
  - Entire type system vs. only reference types
  - Invariant vs. wildcards

```
public class List<T>
{…}

List<int> numbers = new List<int>();
List<Customer> customers = new
List<Customer>();
```

Generics

```
public class List<T>
{
   private T[] elements;
   private int count;

   public void Add(T element) {
      if (count == elements.Length) Resize(count * 2);
      elements[count++] = element;
   }

   public T this[int index] {
      get { return elements[index]; }
      set { elements[index] = value; }
   }

   public
      get
   }
}
```

```
List<int> intList = new List<int>();

intList.Add(1);            // No boxing
intList.Add(2);            // No boxing
intList.Add("Three");      // Compile-time error

int i = intList[0];        // No cast required
```

Generics

- Why generics?
  - Type checking, no boxing, no downcasts
  - Reduced code bloat (typed collections)

- How are C# generics implemented?
  - Instantiated at run-time, not compile-time
  - Checked at declaration, not instantiation
  - Work for both reference and value types
  - Complete run-time type information

Generics

- Type parameters can be applied to
  - Class, struct, interface, delegate types

```
class Dictionary<K,V> {...}

struct HashBucket<K,V> {...}

interface IComparer<T> {...}

delegate R Function<A,R>(A arg);
```

```
Dictionary<string,Customer> customerLookupTable;

Dictionary<string,List<Order>> orderLookupTable;

Dictionary<string,int> wordCount;
```

Generics

- Type parameters can be applied to
  - Class, struct, interface, delegate types
  - Methods

```
class Utils
{
   public static T[] CreateArray<T>(int size) {
      return new T[size];
   }

   public static void SortArray<T>(T[] array) {
      ...
   }
}
```

```
string[] names = Utils.CreateArray<string>(10);
names[0] = "Jones";
...
Utils.SortArray(names);
```

Generics

- Type parameters can be applied to
  - Class, struct, interface, delegate types
  - Methods
- Type parameters can have constraints

```
class Dictionary<K,V>: IDictionary<K,V>
   where K: IComparable<K>
   where V: IKeyProvider<K>, IPersistable, new()
{
   public void Add(K key, V value) {
      ...
   }
}
```

Generics

- Zero or one primary constraint
  - Actual class, `class`, or `struct`
- Zero or more secondary constraints
  - Interface or type parameter
- Zero or one constructor constraint
  - `new()`

```
class Link<T> where T: class {...}

class Nullable<T> where T: struct {...}

class Relation<T,U> where T: class where U: T {...}
```

Generics

- Collection classes
- Collection interfaces
- Collection base classes
- Utility classes
- Reflection

List<T>
Dictionary<K,V>
SortedDictionary<K,V>
Stack<T>
Queue<T>

IList<T>
IDictionary<K,V>
ICollection<T>
IEnumerable<T>
IEnumerator<T>
IComparable<T>
IComparer<T>

Collection<T>
KeyedCollection<T>
ReadOnlyCollection<T>

Nullable<T>
EventHandler<T>
Comparer<T>

Anonymous Methods

- Allows code block in place of delegate
- Delegate type automatically inferred
  - Code block can be parameterless
  - Or code block can have parameters
  - In either case, return types must match

```
button.Click += delegate { MessageBox.Show("Hello"); };
```

```
button.Click += delegate(object sender, EventArgs e) {
   MessageBox.Show(((Button)sender).Text);
};
```

Nullable Types

- System.Nullable<T>
  - Provides nullability for any value type
  - Struct that combines a T and a bool

```
public struct Nullable<T> where T: struct
{
   public Nullable(T value) {...}
   public T Value { get {...} }
   public bool HasValue { get {...} }
   ...
}
```

```
Nullable<int> x = new Nullable<int>(123);
...
if (x.HasValue) Console.WriteLine(x.Value);
```

Nullable Types

- T? same as System.Nullable<T>

```
int? x = 123;
double? y = 1.25;
```

- null literal conversions

```
int? x = null;
double? y = null;
```

- Nullable conversions

```
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
```

Nullable Types

- Lifted conversions and operators

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

- Comparison operators

```
int? x = GetNullableInt();
if (x == null) Console.WriteLine("x is null");
if (x < 0) Console.WriteLine("x less than zero");
```

- Null coalescing operator

```
int? x = GetNullableInt();
int i = x ?? 0;
```

Iterators

- foreach relies on “enumerator pattern”
  - GetEnumerator() method

```
foreach (object obj in list) {
   DoSomething(obj);
}
```

```
Enumerator e = list.GetEnumerator();
while (e.MoveNext()) {
   object obj = e.Current;
   DoSomething(obj);
}
```

- forea
  - But enumerators are hard to write!

Iterators

```
public class List
{
   internal object[] eleme
   internal int count;

   public IEnumerator GetE
      return new ListEnume
   }
}
```

```
public class ListEnumerator : IEnumerator
{
   List list;
   int index;

   internal ListEnumerator(List list) {
      this.list = list;
      index = -1;
   }

   public bool MoveNext() {
      int i = index + 1;
      if (i >= list.count) return false;

rent {
ist.elements[index]; }
```

```
public class List
{
   internal object[] elements;
   internal int count;

   public IEnumerator GetEnumerator() {
      for (int i = 0; i < count; i++) {
         yield return elements[i];
      }
   }
}
```

Iterators

- Method that increme
sequence of values
  - yield return and yield
  - Must return IEnumera

```
public class Test
{
    public IEnumerator (
        yield return "He
        yield return "Wo
    }
}
```

```
public IEnumerator GetEnumerator() {
    return new __Enumerator(this);
}

private class __Enumerator : IEnumerator
{
    object current;
    int state;

    public bool MoveNext() {
        switch (state) {
            case 0:
                current = "Hello";
                state = 1;
                return true;
            case 1:
                current = "World";
                state = 2;
                return true;
            default:
                return false;
        }
    }

    public object Current {
        get { return current; }
    }
}
```

Iterators

```
public class List<T>
{
   public IEnumerator<T> GetEnumerator() {
      for (int i = 0; i < count; i++)
         yield return elements[i];
   }

   public IEnumerable<T> Descending() {
      for (int i = count - 1; i >= 0; i--)
         yield return elements[i];
   }

   public IEnumerable<T> Subrange(int index, int n) {
      for (int i = 0; i < n; i++)
         yield return elements[index + i];
   }
}
```

```
List<Item> items = GetItemList();
foreach (Item x in items) {...}
foreach (Item x in items.Descending()) {...}
foreach (Item x in Items.Subrange(10, 20)) {...}
```

Partial Types

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

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

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

```
public class Customer
{
   private int id;
   private string name;
   private string address;
   private List<Orders> orders;

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

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

Static Classes

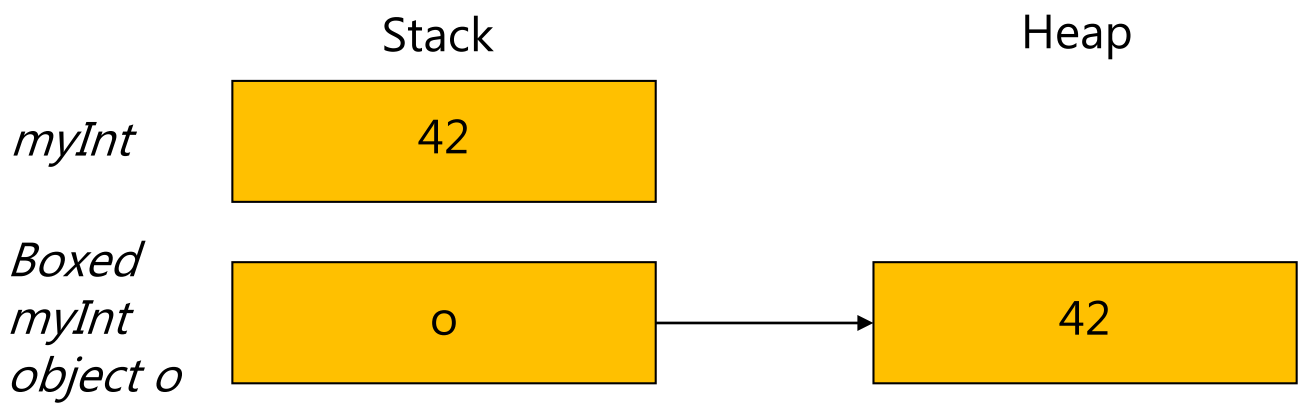
- Only static members
- Cannot be used as type of variable, parameter, field, property, ···
- Examples include System.Console, System.Environment

```
public static class Math
{
   public static double Sin(double x) {...}
   public static double Cos(double x) {...}
   ...
}
```

Boxing And Unboxing

Conversion between value variable and reference variable

```
System.Int32  myInt = 42;
object        o     = myInt;
int           ymInt = (int)o;
```



Exceptions

- **Why do we need exceptions?**
- **How should they be made available in programming languages?**
- **What benefits do they provide?**
- **What problems could they cause for us?**

- **`Throw` raises** an exception
- **`Catch`** defines a block that **handles** the exception

Note on Exceptions

- Exceptions are not for dealing with errors
- They are a mechanism for changing the flow of control from sequential to a branch if certain conditions exist
- They always indicate expected circumstances. Otherwise they could not possibly be generated.

Agenda

- Object-Orientation (O-O)
- Object-Oriented Programming (OOP)
- Programming language C# (2.0)
- Component-based Programming with C#
- **Outlook on current and future C#**

C# Present and Future

- Slides cover programming language features of C# 2.0
- Visual Studio 2015 comes with C# 6.0 and .NET Framework 4.6
- What features have been included or a planned to be included in future releases?

C# 3.0

released August 2007

- Implicitly typed local variables
- Object and collection initializers
- Auto-Implemented properties
- Anonymous types
- Extension methods
- Query expressions
- Lambda expressions
- Expression trees
- Partial methods

C# 4.0 and C# 5.0

released April 2010 / June 2013

C# 4.0

- Dynamic binding
- Named and optional arguments
- Generic co- and contravariance
- Embedded interop types ("NoPIA")

C# 5.0

- Asynchronous methods
- Caller info attributes

C# 6.0
released July 2015

- Compiler-as-a-service (Roslyn)
- Import of static type members into namespace
- Exception filters
- Await in catch/finally blocks
- Auto property initializers
- Default values for getter-only properties
- Expression-bodied members
- Null propagator (null-conditional operator, succinct null checking)
- String Interpolation
- nameof operator
- Dictionary initializer

C# 7.0
planned

- Binary Literals
- Digit Separators
- Local Functions
- Type switch
- Ref Returns
- Tuples
- Out var
- Pattern Matching
- Arbitrary async returns
- Records

Comparison Java, C# and C++

Development of C# has been influenced by experiences on other OO languages

Feature	Java	C#	C++	C
Automatic memory management	+	+	−	−
Exceptions	+	+	+	−
Array bounds checks	+	+	−	−
Classes	+	+	+	−
Structs (value types)	−	+	+	+
Interfaces	+	+	−	−
Inheritance	+	+	+	−
Multiple inheritance	−	−	+	−
Virtual methods	+	+	+	−
Non-virtual methods	−	+	+	−
Method overloading	+	+	+	−
Nested classes	+	+	+	−
Inner classes	+	−	−	−
Call-by-value parameters	+	+	+	+
Reference parameters	−	+	+	−
Safe variable-arity methods	5.0	+	−	−
Properties	−	+	−	−
Indexers	−	+	+	−
Looping over iterators	5.0	+	+	−
Defining iterators (`yield`)	−	2.0	−	−
User-defined operators	−	+	+	−
User-defined conversions	−	+	−?	−
Enum types	5.0	+	+	+
Autoboxing simple values	5.0	+	−	−
Nullable value types	−	2.0	−	−
Rectangular arrays	−	+	+	+
Arrays of arrays	+	+	(−)	(−)
Compile-time conditional code	−	+	+	+
Generic types and methods	5.0	2.0	(+)	−
Runtime type parameter info	−	2.0	−	−
Wildcard types (existentials)	5.0	−	−	−
Generic collection library	5.0	(−)	(+)	−
Anonymous methods	(−)	2.0	−	−
Metadata (attributes/annotations)	5.0	+	−	−

Questions?

Literature

- Peter Sestoft, Henrik Hansen, C# Precisely, 2nd ed., MIT Press, 2011
- Andrew Troelsen and Philip Japikse. C# 6.0 and the .NET 4.6 Framework. 7th ed., Apress, 2016

Next lecture

Tutorial on Tools