Seminar 11
(not required for the final exam)

Please discuss a short Introduction in C#

C# References

- C# Microsoft Programming Guide:
  http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx

- Any online C# tutorial or book

Fundamentals of .NET and C#

The .NET Framework consists of:

- a runtime called the *Common Language Runtime* (CLR)

- a set of libraries:
    - core libraries (*mscorlib.dll, System.dll, System.Xml.dll, System.Core.dll*),
    - applied libraries (Windows forms, ADO.NET, etc).

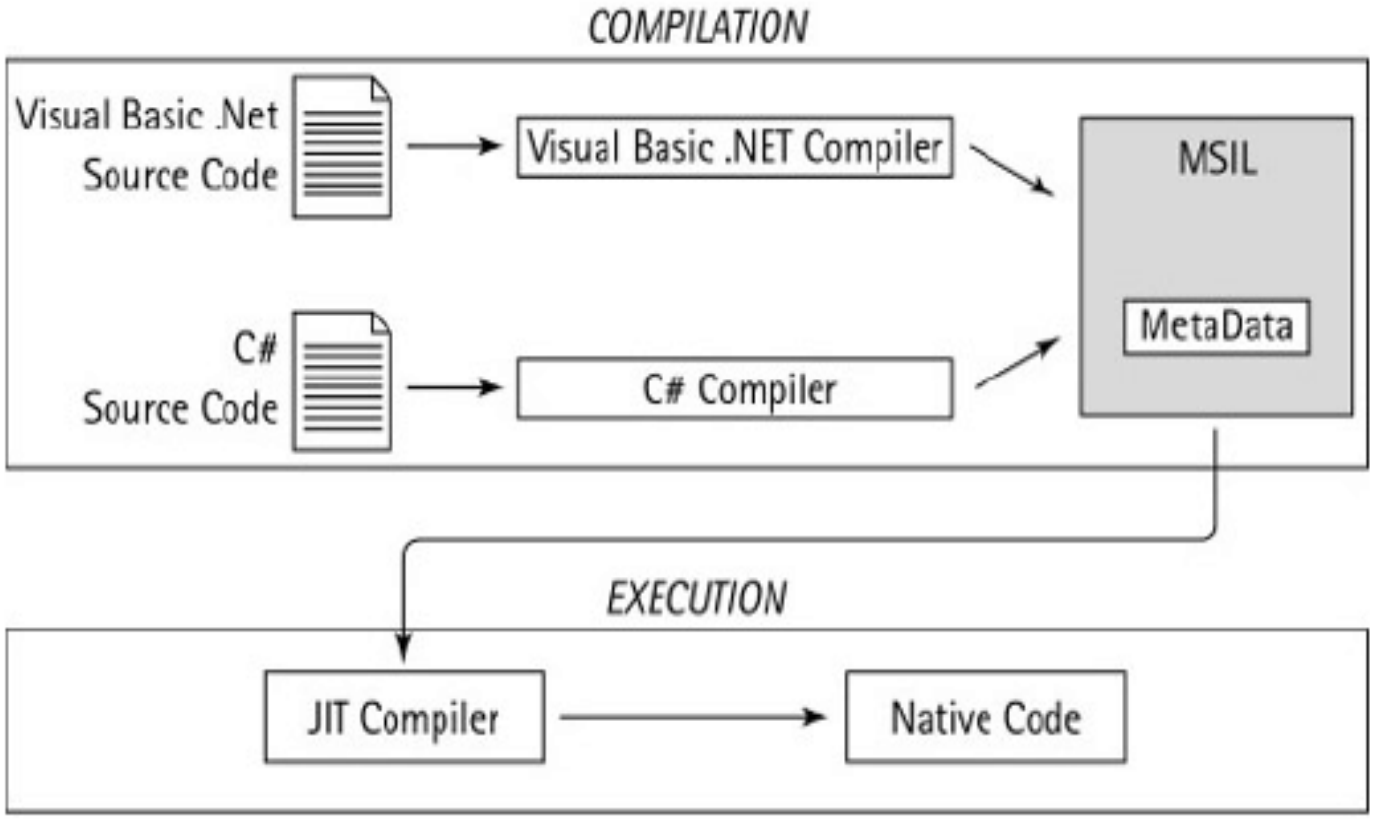
- The CLR is the runtime for executing *managed code*- it is a virtual machine

- Common Intermediate Language (CIL or MSIL)- .Net bytecode

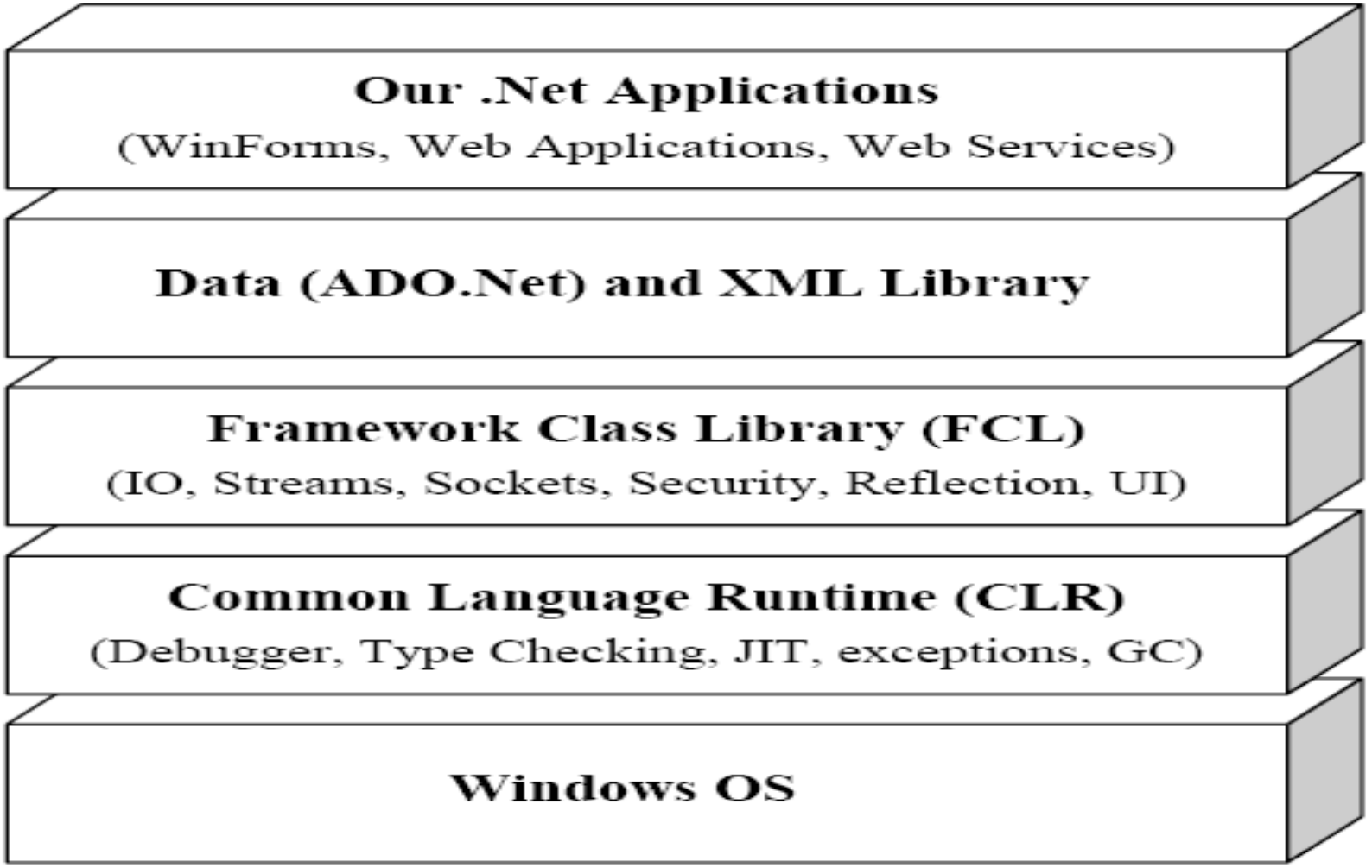
- C# is one of several *managed languages* that get compiled into managed code.

- Managed code is packaged into an *assembly* (an executable file (an *.exe*) or a library (a *.dll*)), along with type information, or *metadata*.

Fundamentals of .NET and C#



Fundamentals of .NET and C#



C# basics

//Java

```
class Test{
 public static void main(String args[]){
   System.out.println(“Hello”);
  }
}
```

//C#

```
using System;
class Test{
  public static void Main(String args[]){
   Console.WriteLine(“Hello”);
  }
}
```

4 different possible entry points:

```
static void Main(){}     static void Main(String[] args){}
static int Main(){}      static int Main(String[] args){}
```

C# compiler

- The C# compiler compiles source code, specified as a set of files with the .cs extension, into an assembly.
- A normal console or Windows *application* has a Main method and is an *.exe*.
- A library is a *.dll* and is equivalent to an *.exe* without an entry point.
- The name of the C# compiler is *csc.exe*

```
csc Test.cs
```

It produces an application named *Test.exe*

```
csc target:library /out:datastruct.dll List.cs Stack.cs
```

Constants

A *constant* is a field whose value can never change. A constant is evaluated statically at compile time and its value is literally substituted by the compiler whenever used.

It can be one of the following types: `sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal, bool, char, string`, or an `enum` type.

A constant is declared with the `const` keyword and must be initialized with a value.

```
const string Message = "Hello World";
```

Java

```
final String message;
…
message=“Hello World”;
```

Arrays

1-dimensional array declaration

```
<data type> [] <identifier>=new <data type>[<size>]
int[] arr=new int[20];
```

N-dimensional arrays

Rectangular arrays are declared using commas to separate each dimension

```
int [,] matrix = new int [2, 3];
```

The `GetLength` method of an array returns the length for a given dimension (starting at 0)

```
for (int i = 0; i < matrix.GetLength(0); i++)
    for (int j = 0; j < matrix.GetLength(1); j++)
        matrix[i, j] = i * 3 + j;
```

Initialization:

```
int[,] matrix = new int[,] { {0,1,2},
                             {3,4,5} };
```

Arrays

- *Jagged arrays* are declared using successive square braces to represent each dimension.

```
int [][] matrix = new int [2][];
```

The inner dimensions are not specified in the declaration. Each inner array can have an arbitrary length. Each inner array is implicitly initialized to null, so it must be created manually:

```
for (int i = 0; i < matrix.Length; i++) {
  matrix[i] = new int [i+1];     // create inner array
  for (int j = 0; j < matrix[i].Length; j++)
     matrix[i][j] = i * 3 + j;
}
```

Initialization:

```
int[][] matrix = new int[][] {
           new int[] {0,1,2},
           new int[] {3,4,5}
           };
```

Enum

An **enum** is a special value type that specifies a group of named numeric constants.

```
public enum BorderSide { Left, Right, Top, Bottom }
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);   // true
```

Each enum member has an underlying integral value. By default:

Underlying values are of type int.

The constants 0, 1, 2... are automatically assigned, in the declaration order of the enum members.

It is possible to specify an alternative integral type:

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

It is also possible to specify an explicit underlying value for each(or some) enum member:

```
public enum BorderSide : byte { Left=1, Right=2, Top=10,
  Bottom }
```

Classes

A class in C# is declared using the `class` keyword:

```
class Person{
 //fields
 //methods
 //properties
}
```

Object creation:

```
Person person=new Person();
```

Field declaration:

```
[static] [access modifier][new][readonly] Type field_name [=init_val];
```

The `readonly` modifier prevents a field from being modified after construction. A read-only field can be assigned only in its declaration or within the constructor.

The new modifier hides the inherited field with the same name as that declared with new.

Constructor /Destructor

Constructor

```
[access modifier] ClassName ([list of parameters]){…}
```

constructor can be overloaded

if no constructor is defined, the compiler automatically generates a default constructor

Destructor

```
~ClassName(){…}
```

The destructor is called automatically when the object is about to be destroyed (when the garbage collector is about to destroy the object).

Destructor/Finalize

- `Finalize ()` method

Each class inherits from the Object class the `Finalize()` method. This method is guaranteed to be called when the object is garbage collected. The `Finalize()` method can be overridden.

```
protected override void Finalize(){…}
```

C# compiler internally converts the destructor to the Finalize() method.

Destructors are not used very much in common C# programming practice.

Methods

```
class Test{
   static void swap(int a, int b){
    int c=a;
    a=b;
    b=c;
   }
   static void Main(){
    int x=23, y=45;
    Console.WriteLine("x={0} y={1}",x,y);
    swap(x,y);
    Console.WriteLine("After x={0} y={1}",x,y);
   }
}
```

x=23 y=45

After x=23 y=45

Methods

```
class Test{
   static void swapRef(ref int a,ref int b){
    int c=a;
    a=b;
    b=c;
   }
   static void Main(){
    int x=23, y=45;
    Console.WriteLine("x={0} y={1}",x,y);
    swapRef(ref x,ref y);
     Console.WriteLine("After x={0} y={1}",x,y);
   }
}
```

x=23 y=45

After x=45 y=23

Methods

```
class Test{
   static void testOut( int a,out int b){
    b=a*a;
   }
   static void Main(){
    int x=2, y;
    Console.WriteLine("x={0} y=",x);
    testOut(x,out y);
    Console.WriteLine("After x={0} y={1}",x, y);
   }
}
```

x=2 y=

After x=2 y=4

Access modifiers

private	Access only within the same class
protected internal	Access only from the same project or from subclasses
internal	Access only from the same project
protected	Access from subclasses
public	No restriction

Default modifier:

fields/methods: private

classes, interfaces: internal

Properties

- Properties look like fields from the outside but act like methods on the inside.
- A property is declared like a field, but with a **`{ get {} set {} }`** block added.
- **`get`** and **`set`** denote property *accessors*. The **`get`** accessor runs when the property is read. It must return a value of the property's type. The **`set`** accessor is run when the property is assigned. It has an implicit parameter named **`value`** of the property's type that is typically assigned to a field.

//Java

```
class Person{
   private String name;
   //…
   public String getName() {return name; }
   public void setName(String name) {this.name=name;}
}
```

Properties

//C#

```
class Person{
   private String name;
   //…
   public String Name {
       get{ return name; }
       set{ name=value;}
   }
}
```

- A property is *read-only* if it specifies only a get accessor, and it is *write-only* if it specifies only a set accessor.
- A property usually has a dedicated backing field to store the underlying data.
- A property can also be computed from other data.

Properties

```
public class Stock {
   string symbol;
   double price;
   long sharesOwned;
   public Stock (string symbol, double price, long shares) {
       this.symbol = symbol;
        this.price = price;
        this.sharesOwned = shares; }
   public double Price {
        get { return price; }
        set { price = value; } }
   public string Symbol {
         get { return symbol; } }
   public long SharesOwned {
         get { return sharesOwned; } }
   public double Worth { get { return price*SharesOwned; } }
}
```

Indexers

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are similar to properties, but are accessed via an index **argument** rather than a property name.

```
public class Portfolio {
    Stock[] stocks;
    public Portfolio (int numberOfStocks) {
        stocks = new Stock [numberOfStocks];
    }
    public int NumberOfStocks {
        get { return stocks.Length; }
     }
    public Stock this [int index]          // indexer
    {
        get { return stocks [index]; }
        set { stocks [index] = value; }
    }
 }
```

Indexers

A type may declare multiple indexers (overloaded indexers).

The Portfolio class can be extended to also allow a stock to be returned by a symbol:

```
public class Portfolio {
    ...
    public Stock this[string symbol] {
        get { foreach (Stock s in stocks)
            if (s.Symbol == symbol) return s;
              return null;
        }
    }
    //An indexer can take any number of parameters.
    public Stock this [string symbol, string exchange] {
        get { ... }
        set { ... }
     }

 }
```

Inheritance

In C# the : is used for inheritance.

```
class Person{ … }
class Student : Person { … }
```

Abstract classes and methods: `abstract` keyword

```
abstract class Element{
    public abstract int CompareTo(Element e);
}
```

Interfaces: `interface` keyword

```
public interface IComparable{
    int CompareTo(Object o);
}
```

Inheritance

C# supports single class inheritance, and multiple interface implementations.

```
class A { … }
interface B{…}
interface C : B {…}        //interface inheritance
interface CC {…}
class SA : A, B, CC  { … }  //class inheritance, interface
//implementation
```

Virtual methods

A function marked as `virtual` can be overridden by subclasses wanting to provide a specialized implementation. Methods, properties, indexers, and events can all be declared virtual:

```
public class Person{
        private String address;
        public virtual String getAddress(){…};
}
```

A subclass overrides a virtual method by applying the `override` modifier

```
public class Student:Person{
        public override String getAddress(){…}
}
```

It is possible to hide a member (field/method/property) deliberately, using the `new` modifier to the member in the subclass.

```
public class Student:Person{
        private new String address;
        public new String getAddress(){…}
}
```

`base` keyword

It is used to explicitly call the constructor of the base class.

```
class SubClass : BaseClass{
   SubClass(int id):base (){ //explicitly call the default
   constructor
   //…
   }
   SubClass(String n, int id): base(n){
   //…
   }
}
```

`sealed` keyword

A sealed method cannot be overridden by a subclass.

```
class AA{
   public virtual int f(){…}
}
class BB: AA{
   public sealed override int f(){…}
}
class CC: BB{
   public  override int f(){…}    // compile time error
}
```

A sealed class implicitly seals all the virtual functions and prevents subtyping.

```
sealed class DD{ … }
```

Static constructor

Static constructor

```
public class Set{
     static int capacity;
     static int instances=0;
     static Set(){
          capacity=20;
     }
     // …
}
```

A static constructor executes once per type, rather than once per instance.

A static constructor executes before any instances of the type are created and before any other static members are accessed.

A type can define only one static constructor.

The static constructor has no parameters and it has the same name as the class.

It can access only the static members from the class.

Static field assignments occur before the static constructor is called, in the declaration order in which the fields appear.

Static Classes

Static classes

A class can be marked `static`, indicating that it contains only static members (fields, methods, properties).

```
static class EncodingUtils{
 public static String encode(String txt){…}
      public static String decode(String txt){…}
}
```

A static class cannot be subclassed.

`System.Console`

`System.Math`

Partial Classes

Partial classes allow a class definition to be split—typically across multiple files.

A common usage is for a partial class to be auto-generated from some source, and for that class to be augmented with manually added methods.

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }
// PaymentForm.cs – manually added
partial class PaymentForm { ... }
```

Each participant must have the `partial` declaration.

Participants cannot have conflicting members.

Partial classes are resolved entirely by the compiler, which means that each participant must be available at compile time and must reside in the same assembly.

Partial Methods

A partial class may contain *partial methods*.

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ...
 partial void ValidatePayment(decimal amount);
}
// PaymentForm.cs - manually
partial class PaymentForm { ...
  partial void ValidatePayment(decimal amount) {
       if (amount > 100) ...
 }
}
```

A partial method consists of two parts: a *definition* and an *implementation*. The definition is typically written by a code generator, and the implementation is typically manually added. If an implementation is not provided, the definition of the partial method is compiled away.

`Partial` methods must be `void` and are implicitly `private`.

as and is Operators

The `as` operator performs a downcast that evaluates to null if the downcast fails:

```
Person pers=new Person();
Student st= pers as Student();    //st is null
Person pers1=new Student();
Student st1=pers1 as Student();   //st is not null
```

The `is` operator tests whether a downcast would succeed (i.e., whether an object derives from a specified class or implements an interface). It is often used to test before downcasting.

```
if ( pers1 is Student){ … }
```

Implementing interfaces

```
public interface ILog{
   void Write(String mess);
}
public interface IFile{
   void Write(String mess);
}
public class MyFile:ILog, Ifile{   //different than Java
   public void Write(String mess){
    Console.WriteLine(mess);
   }
   void ILog.Write(String mess){
    Console.WriteLine("ILog: {0}", mess);
   }
}
…
void Main(){
   ILog ilog=new MyFile();
   IFile ifile=new MyFile();
   ifile.Write("ana");        //ana
   ilog.Write("ana");     //ILog: ana
}
```

Object class

All classes inherits from Object class (from System namespace).

```
public class Object {
   public extern Type GetType( );
   public virtual bool Equals (object obj);
   public virtual int GetHashCode( );
   public virtual string ToString( );
   protected override void Finalize( );
    public static bool Equals (object objA, object objB);
    public static bool ReferenceEquals (object objA, object objB);
   //…
}
```

Structs

- A struct is similar to a class, with the following key differences:
  - A struct is a value type, whereas a class is a reference type.
  - A struct does not support inheritance
- A struct can have all the members a class can, except the following:
  - A default constructor
  - A finalizer
  - Virtual members

```
public struct Complex
{
  double re, im;
  public Complex (double re, double im) {this.re = re; this.im = im;}
}
...
Complex c1 = new Complex ( );       // c1.re and c1.im will be 0.0
Complex c2 = new Complex (1, 1);   // c2.re and c2.im will be 1.0
```

Struct

- A default constructor that cannot be overridden implicitly exists. It performs a bitwise-zeroing of its fields.
- In a struct constructor, every field must be explicitly initialized. There cannot be field initializers in a struct.

```
public struct Point
{
  int x = 1;
  int y;
  public Point( ) {}  //error

 }
```

Nested Types

A *nested type (class or struct)* is declared within the scope of another type.

```
class List{
    public class Node{…}
}
```

A nested type can access only the enclosing **`static`** members (even **`private`**).

It can be declared with the full range of access modifiers (not just **`public`** or **`internal`**). The default visibility for a nested type is **`private`**.

Accessing a nested type from outside the enclosing type requires qualification with the enclosing type's name.

```
List.Node n;
```

Operator Functions

An operator is overloaded by declaring an *operator function*. An operator function has the following rules:

- The name of the function is specified with the `operator` keyword followed by an operator symbol.
- The operator function must be marked `static`.
- The parameters of the operator function represent the operands.
- The return type of an operator function represents the result of an expression.
- At least one of the operands must be the type in which the operator function is declared.

Operator overloading

```
class Complex    {
  double re, im;
  public Complex(double re, double im){this.re=re; this.im = im;}
  public static Complex operator +(Complex a, Complex b) {
    return new Complex(a.re + b.re, a.im + b.im);
  }
  public  static bool operator ==(Complex a, Complex b) {
    return (a.re == b.re) && (a.im == b.im);
  }
  public static bool operator !=(Complex a, Complex b){
    return !(a==b);
  }
  //…
}
```

Operator overloading

The C# compiler enforces operators that are logical pairs to both be defined. These operators are (`==` `!=`), (`<` `>`), and (`<=` `>=`).

If you overload (`==`) and (`!=`), you need to override the `Equals` and `GetHashCode` methods defined on `Object`. The C# compiler will give a warning if you do not override them.

If you overload (`<` `>`) and (`<=` `>=`), you should implement `IComparable` and `IComparable<T>`.

Delegates

- Delegates are references to methods.
- They are similar to function pointers from C++.
- Delegates are similar to object references, but they are used to reference methods instead of objects.

- A delegate has three properties:
  - The type or signature of the method that the delegate can point to.
  - The delegate reference which can be used to reference a method.
  - The actual method referenced by the delegate.

- Delegates declaration, using `delegate` keyword:

```
delegate <return type> DelegateName(<list of parameters>);
```

Example:

```
delegate int ArithmeticMethod(int a, int b);
```

User defined delegates are subclasses of `System.Delegate` class. The class is automatically generated by the compiler, it cannot be explicitly created by the user.

Delegates

Initialization and usage:

```
delegate String StringEncoder(String text);
class Test{
 public void Main(string args[]){
      String text="Ana are mere";
      EncoderUtils se=new EncoderUtils();
    StringEncoder enc1=new StringEncoder(ToLower);
      StringEncoder enc2=new StringEncoder(se.encodeA);
      StringEncoder enc3=new StringEncoder(se.encodeB);
      Console.WriteLine("ToLower ={0}", enc1(text));
      Console.WriteLine("encodeA ={0}", enc2(text));
      //…
 }
 static String ToLower(String text){
      return text.ToLower();
 }
}
class EncoderUtils{
 public String encodeA(String text){ … }
 public String encodeB(String txt){ … }
 public int encodeC(String txt){ … }
}
```

Multicast Delegates

- All delegate instances have multicast capability.
- A delegate instance can reference not just a single target method, but also a list of target methods. The `+=` operator combines delegate instances, and `-=` operator removes delegates instances.

```
String text="Ana are mere";
EncoderUtils se=new EncoderUtils();
 StringEncoder enc=new StringEncoder(ToLower);
enc+=new StringEncoder(se.encodeA);
enc+=new StringEncoder(se.encodeB);
enc(text); //all three methods are called, in the
//order they were added
enc-=ToLower;
enc(text); //only two methods are called
```

- A multicast delegate inherits from `System.MulticastDelegate` (that inherits from `System.Delegate`)

Multicast Delegates

- If a multicast delegate has a non void return type, the caller receives the return value from the last method to be invoked. The preceding methods are still called, but their return values are discarded.

- C# compiles `+=` and `-=` operations made on a delegate to the static `Combine` and `Remove` methods of the `System.Delegate` class.

- When a delegate instance is assigned an instance method, the delegate instance must maintain a reference not only to the method, but also to the instance of that method. The `Target` property of the `System.Delegate` class represents this instance. If the method is `static`, the result is `null.`

Implicitly Typed Local Variables

- Starting with C# 3.0 it is possible to declare and initialize a local variable without explicitly specifying the type.
- If the compiler is able to infer the type from the initialization expression, the keyword **`var`** can be used in place of the type declaration.

```
var x = 5;
var y = "hello";
var z = new System.Text.StringBuilder();
```

It is equivalent to:

```
int x = 5;
String y = "hello";
System.Text.StringBuilder z = new System.Text.StringBuilder();
```

Implicitly Typed Local Variables

Implicitly typed variables are statically typed:

```
var x = 5;
x = "hello";    // Compile-time error; x is of type int
```

- `var` can decrease code readability when the type cannot be deduced just by looking at the variable declaration.

```
Random r = new Random();
var x = r.Next();      //int
```

Lambda Expressions

- A *lambda expression* is an unnamed method written in place of a delegate instance.
- They were introduced in C# 3.0.
- The compiler immediately converts the lambda expression to either:
  - A delegate instance.
  - An expression tree, of type Expression<TDelegate>, representing the code inside the lambda expression in a traversable object model. This allows the lambda expression to be interpreted later at runtime.

Example

```
delegate int Transformer (int i);

Transformer sqr = x => x * x;      //lambda expression
Console.WriteLine (sqr(3)); // 9
```

Lambda Expressions

- A lambda expression has the following form:

```
(parameters) => expression-or-statement-block
```

- The parentheses can be omitted if and only if there is exactly one parameter of an inferable type.
- Each parameter of the lambda expression corresponds to a delegate parameter, and the type of the expression (which may be `void`) corresponds to the return type of the delegate.
- A lambda expression's code can be a statement block instead of an expression.

```
x => { return x * x; };
```

- When the compiler cannot infer the type of the lambda parameter contextually, you must specify the type explicitly:

```
(int x) => x * x
```

Extension Methods

- *Extension methods* allow an existing type to be extended with new methods without altering the definition of the original type. They were added in C# 3.0.
- An extension method is a static method of a static class, where the **`this`** modifier is applied to the first parameter. The type of the first parameter will be the type that is extended.

```
public static class StringHelper
{
    public static bool IsCapitalized (this string s)
    {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
    }
}
```

Call:

```
    String location="Cluj";
    Console.WriteLine (location.IsCapitalized());
```

Extension Methods

- An extension method call, when compiled, is translated back into an ordinary static method call:

```
Console.WriteLine (StringHelper.IsCapitalized (location));
```

- The translation works as follows:

```
arg0.Method (arg1, arg2, ...); // Extension method call
StaticClass.Method (arg0, arg1, arg2, ...); // Static method
```

- Remarks:
  - An extension method cannot be accessed unless the namespace is in scope. You have to use the `using` directive.
  - Any compatible instance method (having the same signature) will always take precedence over an extension method. The extension method can still be called using its normal static syntax.
  - If two extension methods have the same signature, the extension method must be called as an ordinary static method to disambiguate the method to call.

C# Namespaces

- Classes can be grouped in namespaces
  - A hierarchical grouping of classes and other entities
  - Every source file defines a global namespace
    - possibly implicitly, if the user doesn't provide a name
- Affects visibility of various classes

- Unlike Java, there need not be any connection between namespaces and directory structure
- The following are allowed in C# and disallowed in (the official implementation of) Java:
  - multiple public classes in the same file
  - splitting the declaration of a class across multiple files

The Global Namespace

- The global namespace consists of:
  - » All top-level namespaces
  - » All types not declared in any namespace

```
class Utils  {}
namespace tests {
  namespace model  {
        class Person {…}
        class Question{…}
    }
}
```

The class `Utils` and the namespace `tests` belong to the global namespace.

Namespaces

All names present in outer namespaces are implicitly imported into inner namespaces.

If you want to refer to a type in a different branch of your namespace hierarchy, you can use a partially qualified name.

```
namespace tests{
 namespace model{
   class Person{}
 }
 namespace gui{
   class TestForm {
      model.Person p;
   }
 }
}
```

Names in inner namespaces hide names in outer namespaces.

Namespaces

- A namespace declaration can be repeated, as long as the type names within the namespaces do not conflict.

```
namespace tests.model{
 class Person{}
}
namespace tests.model{
 class Question{}
}
```

- The `using` directive can be nested within namespaces.

```
namespace N1 {
  class Class1 {}
}
namespace N2{
  using N1;
  class Class2 : Class1 {}
}
namespace N2 {
  class Class3 : Class1 {}   // compile error
}
```

Aliasing Types and Namespaces

The `using` directive can be used for declaring an alias for a type or for a namespace:

```
using person=tests.model.Person;   //alias for a type
using win=tests.gui;        //alias for a namespace
class Test{
 void Main(){
     person p=new person();
     win.TestForm tf=new win.TestForm();
 }
}
```

- `object` is an alias for `System.Object`
- `string` is an alias for `System.String`

BCL (Base Class Library)

`System`

(basic language functionality, fundamental types)

`System.Collections` (collections of data structures)

`System.IO` (streams and files)

`System.Net` (networking and sockets)

`System.Reflection` (reflection)

`System.Security`

(cryptography and management of permissions)

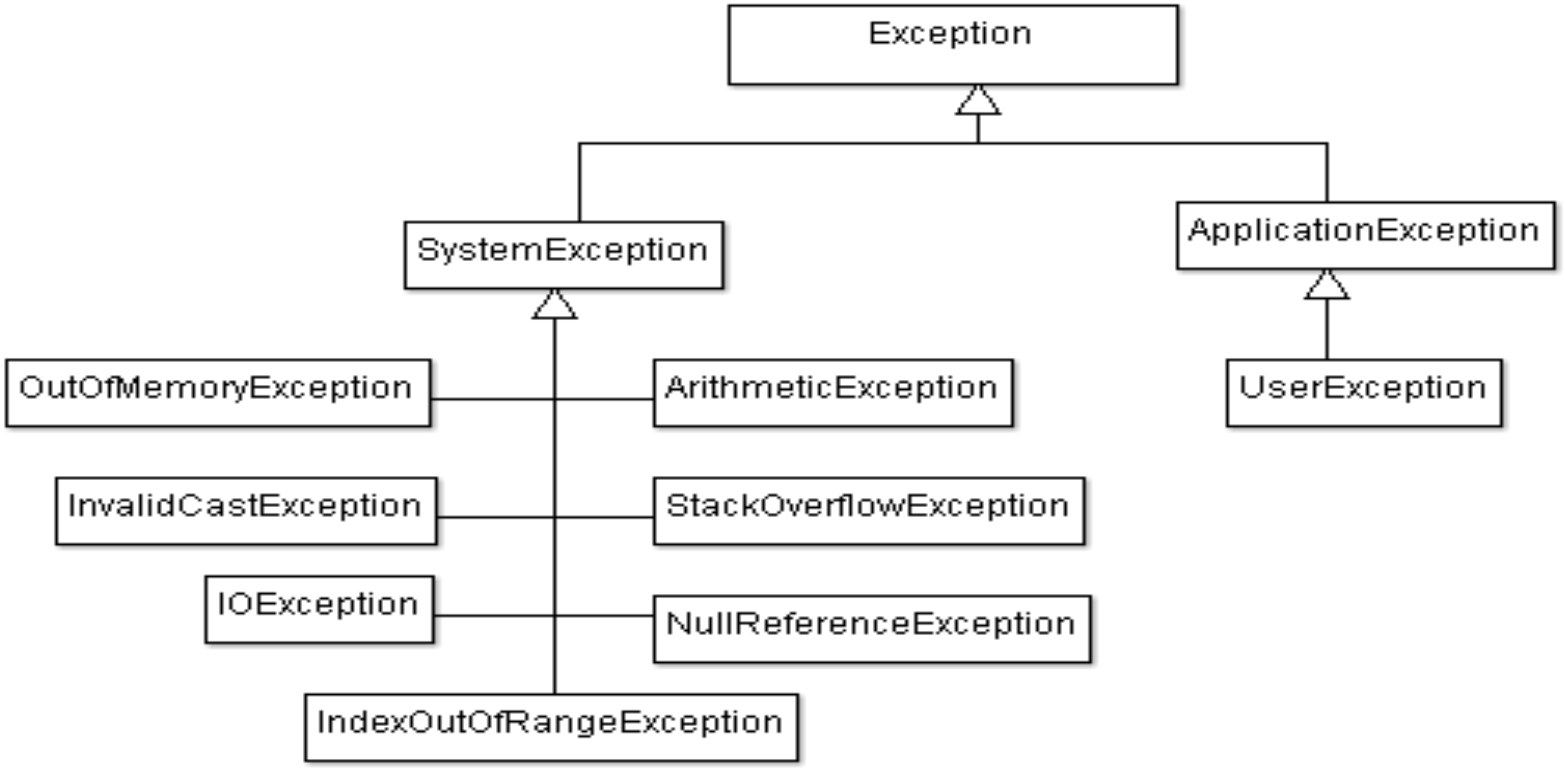
`System.Threading` (multithreading)

`System.Windows.Forms`

(GUI components, nonstandard, specific to the Windows platform)

C# Exceptions

`System` namespace



- All exceptions in C# are unchecked exceptions.
- There is no equivalent to Java's compile-time checked exceptions.

User Defined Exception

Inherits from **`ApplicationException`**

There are four constructors inherited from **`Exception`** that can be called:

- The default constructor.
- A constructor that takes a **`String`** as a message.
- A constructor that takes a message and an inner, lower-level **`Exception`**.
- **`Exception(SerializationInfo, StreamingContext).`**

```
class StockException : ApplicationException{
  public StockException(){ }
  public StockException(String message) : base(message) { }
  public StockException(String msg, Exception exp):base(msg,exp) { }
}
```

Try-catch block

```
try {
   // Code that might generate exceptions
} catch(Exception1 e1) {
   // Handle exceptions of type Exception1
}
// more catch block
catch(Exceptionn en) {
   // Handle exceptions of type Exceptionn
}finally{
   // code that is always executed
}
```

Example:

```
try{
   int a=10, b=0;
   int d=a/b;
}catch(Exception e){
   Console.WriteLine("Exception "+e);
}
```

Properties of `System.Exception`

- **`StackTrace`** A string representing all the methods that are called from the origin of the exception to the catch block.
- **`Message`** A string with a description of the error.
- **`InnerException`** The inner exception (if any) that caused the outer exception. This, itself, may have another InnerException.

```
try{
   …
}catch(AnException e){
   Console.WriteLine(“Error message {0}“,e.Message);
   Console.WriteLine("Stack Trace {0}", e.StackTrace);
}
```

Finally clause

- A **`finally`** block is **always** executed after the **`try`** block even if no exceptions are thrown
  - It may be used to free resources
- **`return`** statements cannot occur in **`finally`** blocks
- **`goto`**, **`break`**, and **`continue`** statements can occur in**`finally`** blocks only if they do not transfer control outside the **`finally`** block itself
- The above restrictions disallow tricky cases that are allowed inJava

Valid Java code but invalid C# code:

```
public int foo() {
try { return 1; }
finally { return 2; }
}

public void foo() {
int b = 1;
while (true) {
    try { b++; throw new Exception(); }
    finally { b++; break; }
}
b++;
}
```

Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

- Publisher(Subject)-Subscriber(Observer) relationship:

A publisher is one who publish data and notifies it to the list of subscribers who have subscribed for the same to that publisher.

Example:

A simple example is Newspaper.

Whenever a new edition is published by the publisher,it will be circulated among subscribers whom have subscribed to publisher.

Events

- Events are a language feature that formalizes the Publisher/Subscriber (Observer) pattern.
- An *event* is a wrapper for a delegate that exposes just the subset of delegate features required for the publisher/subscriber model.
- The main purpose of events is to prevent subscribers from interfering with each other.
- To declare an event member, the `event` keyword is put in front of a delegate member.

Events

1. Define a public delegate

```
public delegate void DelegateEvent(Object sender, EventArgs args);
```

The first parameter is usually the originator of the event, and the second parameter usually holds any additional data to be passed to the event handler.

2. Define a class that generates or raises the event (`Publisher`). Inside this class a public event is declared.

```
class Publisher{
 public event DelegateEvent eventName;
  …
 … someMethod(…){
    //code that raises an event
    EventArgsSubClass args=new EventArgsSubClass(<some data>);
    eventName(this, args);
    //or eventName(this, null);
 }
}
```

Events

3. Define the class(es) that handle the appearance of an event (`Observer`). The name of the event handler conventionally starts with “On”.

```
class Observer{
  //the methods that matches the delegate signature
  public void OnEventName(Object sender, EventArgs args){
  //event handling code
  } …}
```

- Configuration

```
class StartApp{
  … Main(){
  //create the publisher(subject)
  Publisher pub=new Publisher(…);
  //create the observers
  Observer obs1=new Observer(…);
  Observer obs2=new Observer(…);
  //subscribe the observers to the event
  pub.eventName+=new DelegateEvent(obs1.OnEventName);
  pub.eventName+=new DelegateEvent(obs2.OnEventName);
  pub.someMethod(…);//explicit call of the method that raises the event
  }
}
```

Events Example

```
public delegate void TimerEvent(object sender, EventArgs args);
class ClockTimer{
   public event TimerEvent timer;
   public void start(){
    for(int i=0;i<3;i++){
         timer(this, null);
         Thread.Sleep(1000);
    }
   }
}
class Test{
   static void Main(){
    ClockTimer clockTimer=new ClockTimer();
    clockTimer.timer+=new TimerEvent(OnClockTick);
    clockTimer.start();
   }
   public static void OnClockTick(object sender, EventArgs args){
    Console.WriteLine("Received a clock tick event!");
   }
}
```

Events

For reusability, the **`EventArgs`** is subclassed and it is named according to the information it contains. It typically exposes data as properties or as read-only fields.

The rules for choosing or defining a delegate for the event are:

- It must have a void return type.
- It must accept two arguments: the first of type **`object`**, and the second a subclass of **`EventArgs`**. The first argument indicates the event publisher, and the second argument contains the extra information to be passed.

Starting with .NET Framework 2.0 a generic delegate, called **`System.EventHandler<T>,`** is defined, that satisfies these rules.

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e) where TEventArgs : EventArgs;
//publisher class
public event EventHandler<TEventArgs> concreteEvent;
protected virtual void OnEvent (TEventArg e) {
    if (concreteEvent != null) concreteEvent (this, e);
  }
```