Introduction to C#

- Lecture 3 – Introduction to C# Part 2

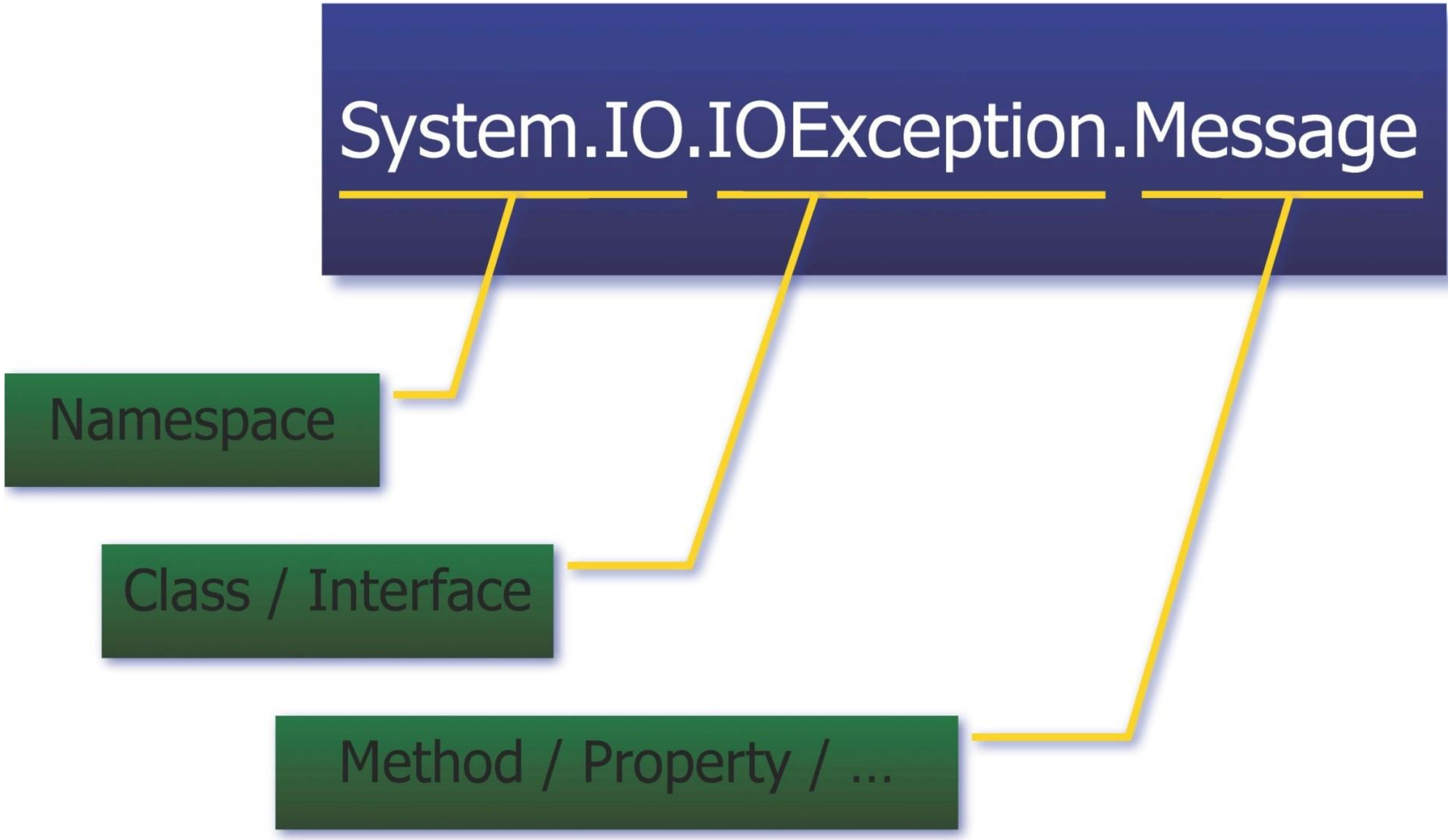
C# .NET Framework
Programming

Outline

We will look at more of the C# features

- Using namespaces (revision)
- Value and reference types (revision)
- More on classes and Interfaces in C#
- Simple Generics <T>
- Exceptions

Namespaces



System.IO.IOException.Message

Namespace

Class / Interface

Method / Property / ...

Using a namespace

```
namespace StreamUtils
{
       public class StreamReader
        {
         public static string ReadString(System.IO.Stream s)
                  {
                                 . . .
namespace StreamUtils
{ public class StreamPrinter
        {
        public static void DisplayData(System.IO.Stream s){
        string data;
                  while ((data = ReadString(s)) != null) {
                  System.Console.WriteLine("{0}", data);
                   }
       }

 }
```

Namespaces and Using

```
namespace StreamUtils
{
    public class StreamReader
    {
        public static string ReadString(System.IO.Stream s)
        {
            . . .
```

```
namespace StreamUtils
{
    public class Stream
    {
        public static vo
        {
            string data
            while ((data
            {
                System.
            }
        }
    }
}
```

```
namespace MyApplication
{
    using StreamUtils;


    public class MyClass
    {
        . . .


        StreamPrinter.DisplayData(stream);


        string s = StreamReader.ReadString(stream);

        . . .
    }
}
```

Value and reference variables

- Value
  - Data is stored and accessed directly
  - On the stack
  - **Structures, primitive types (int, float, double etc.)**
- Reference
  - Data is stored and accessed indirectly
  - On the heap
  - **Classes, arrays**

More on classes

- C# supports only single inheritance (just like java), so <u>does not</u> support mul) ple inheritance
- It does support **interfaces** however
- Classes can then implement one or more interfaces

Interfaces

- An **interface** looks like a class, but has no implementation
- The only thing it contains are **declarations** of *events*, *indexers*, *methods* and/or *properties* The reason **interfaces** only provide declarations is because they are inherited by *classes* and *structs*, which must provide an implementation for each interface member declared
- You can then say that a class **conforms** to an interface such *printable* or *serializable* etc.

Interfaces

- So, what are **interfaces** good for if they don't implement functionality?
- They're great for putting together plug-n-play like architectures where components can be interchanged at will
- Since all interchangeable components implement the same **interface**, they can be used without any extra programming
- The **interface** forces each component to expose specific public members that will be used in a certain way
- Interfaces remain the same whilst implementation can change

Interface Example

```
// Interface implementation
using System;
interface IMyInterface
{
    void MethodToImplement();
}
```

This *interface* has a single method named *MethodToImplement()*

Example

```
interface IPrintable
{
    String printDescription();
}

class DVD: Product , IPrintable
{
   String printDescription(){
}
    //my custom method implementation
    //to print description
}
```

Summary

- Only methods, properties, and events are allowed
- An interface is like a contract with a class
  - You must implement this method (**printDescription** in our example)
  - It is up to the you (the class) how you implement it
- When it is called you are talking to the interface so it is very consistent across all classes that implement the same interface

Working with any object

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

This is know as a generic

We might want to define an interfaces that does something with objects

This means I can be any object

Example of <T>

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        if (this.Make == car.Make &&
            this.Model == car.Model
            && this.Year == car.Year)
        {
            return true;
        }
        else
            return false;
    }
}
```

Remember
(T obj)

A bit more on generics

```
public struct Point<T>
{
   public T X;

   public T Y;
}
```

You can use the generic point for integer coordinates, for example:

```
Point<int>
point; point.X =
1;
point.Y = 2;
```

Or for coordinates that require floating point precision:

```
Point<double> point;
point.X = 1.2;
point.Y = 3.4;
```

Catching Exceptions

- C# allows you to call methods that throw exceptions
- Exceptions are a mechanism for error trapping
- They are essentially an object that has methods that detects an error or condi) on
- They also provide useful information about why the exception was thrown
- They are closely linked to the **try – catch** statement

Catching Exceptions

- C# allows you to call methods that throw exceptions without a try – catch
- Therefore your code could crash if you don’t catch them
- **Intellisense** will tell you if methods throw exceptions but some errors are less obvious such as divide by zero

try – catch – finally

```
try
{
  //your code to try here
  //note it must throw an exception
}
catch(Exception ex)
{
  //you exception handling code here
  //usually a note to the user or to
  //an error or both
}
```

The exception object contains details of the exception thrown

With ‘finally’

```
try
{
  //your code to try here
  //that throws exceptions
catch(Exception ex)
{
  //your exception handling code here
}
finally
{
  //we ALWAYS go here
  //your clean up code here – used for closing DB, file, sockets, etc.
}
```

Creating Custom Exceptions

```
public class MyException: Exception
{
 public MyException()
 {
 }

 public MyException(string message): base(message)
 {
 }
  public MyException(string message, Exception inner): base(message, inner) {
  }
}
```

More on these later as we will build one in a tutorial