C# Programming

29 October 2021

Atos | Syntel

Content

Content

Introduction to .NET Framework and C# Programming

1

Atos | Syntel

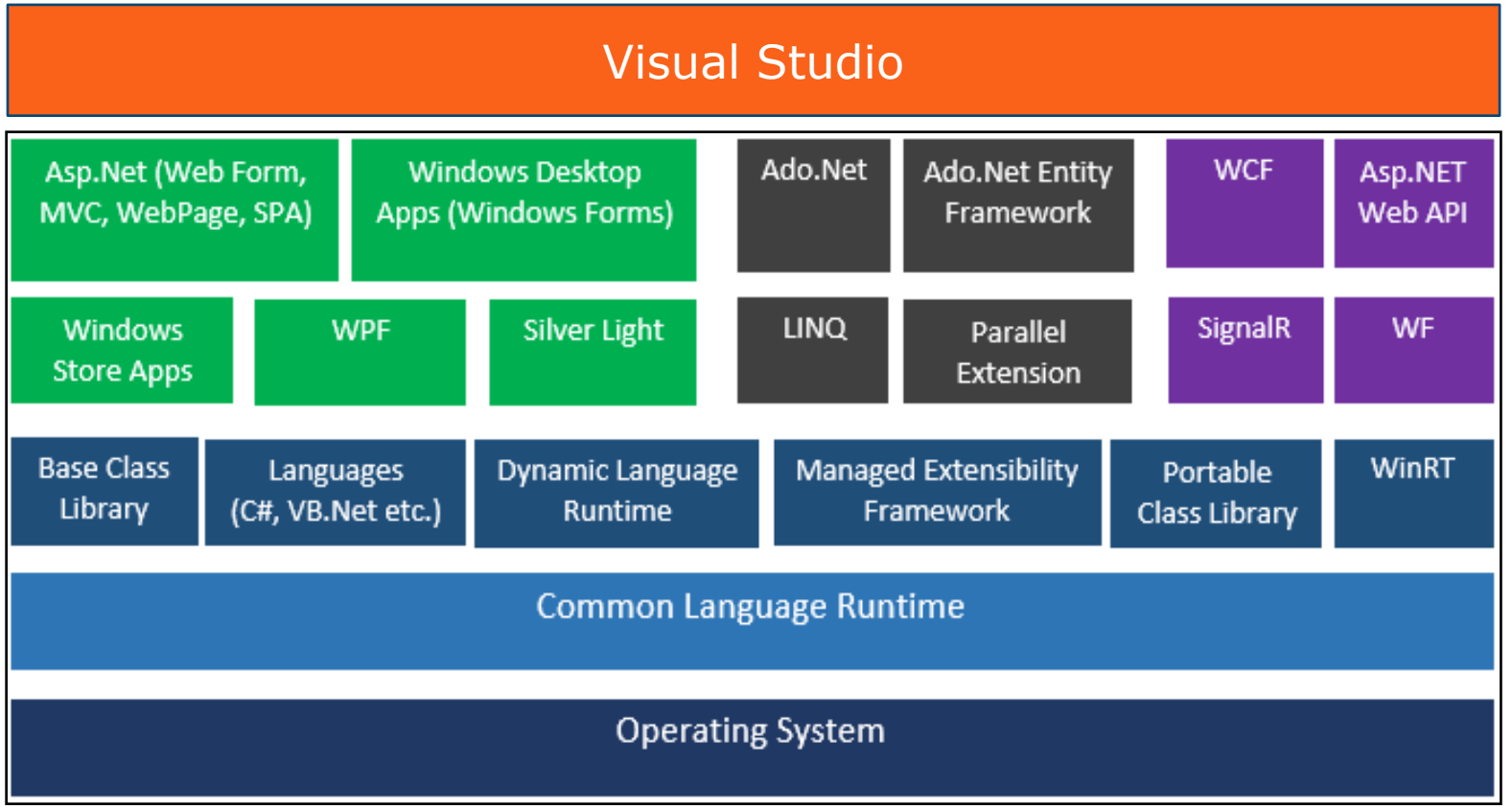
Introduction to .NET Framework

.NET Framework

- The .NET Framework is a technology that supports building, deploying and running the next generation of apps and XML Web services.
- The .NET framework is an execution and development platform for building apps for Windows, Windows Phone, Windows Server and Windows Azure.
- The .NET Framework is designed to fulfil the following objectives:
  - consistent object-oriented programming environment
  - code-execution environment that minimizes software deployment and versioning conflicts and safe execution of code.
  - build all communication on industry standards to ensure that code based on the .NET Framework integrates with any other code.

Introduction to .NET Framework

.NET Framework Architecture



Atos | Syntel

Introduction to .NET Framework

Component of .NET Framework – Common Language Runtime

- CLR is runtime environment to execute .NET Apps. Commercial implementation of CLI by Microsoft.
  - Code Execution
  - Exception Handling
  - Resource Management
  - Garbage Collection
  - Just In Time Compilation
  - Code Access Security
  - Language Interoperability
  - Application Isolation
- CLI – Common Language Infrastructure is an international standard that is the basis for creating execution and development environments in which languages and libraries work together seamlessly.

Introduction to .NET Framework

Component of .NET Framework – Common Language Runtime

- **JIT Just-in-Time** compiler compiles CIL to Machine Code. Compiles portion of code as it is called and it is cached.
- **CLS Common Language Specification** is a set of standards that all compilers targeting .NET must support. CLS work with CTS to ensure language interoperability.
- **CTS Common Type System** defines the set predefined data types available in it.
- **CAS Code Access Security** applies the permission to code, based on source of code and other identities.
- **GC Garbage Collection** manages allocation and deallocation of objects in memory.

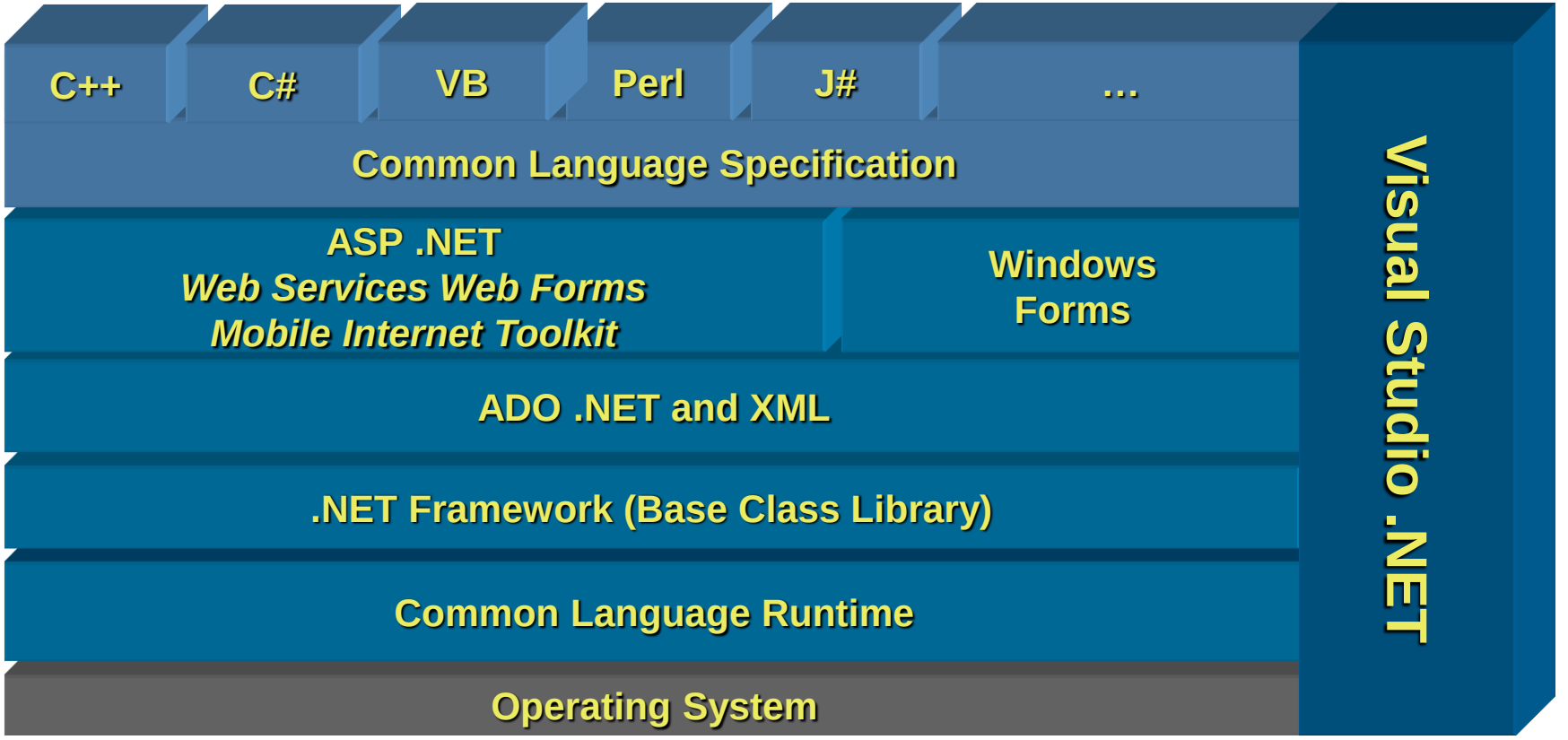
Introduction to .NET Framework

Component of .NET Framework – Framework Class Library

- The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime.
- The class library is object oriented, providing types from which your own managed code derives functionality
- Consists of classes, interfaces, and structures, delegates etc.
- Library of tested, reusable code that developers can use in own applications
- are categorized using namespaces.

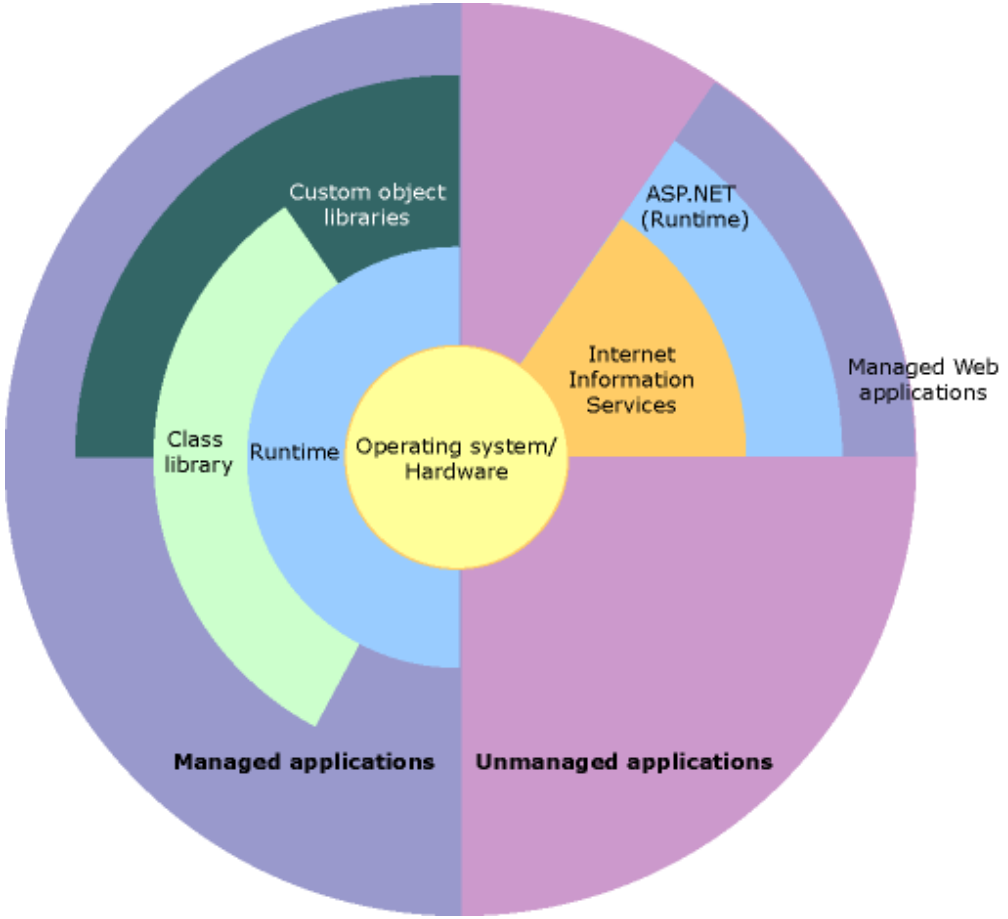
Introduction to .NET Framework

Component of .NET Framework



Introduction to .NET Framework

.NET Framework Overview



Introduction to .NET Framework

Assembly

- An assembly is the smallest deployable unit of execution.
- It is the basic building block of .NET Framework.
- It Consists of Meta-Information and CIL
- It is Self Descriptive
- Have unique version number. Contains IL code to be executed.
- There are two types of assemblies
  - Private: Can be used only in one application. Stored in the specific application's directory or sub directory.
  - Public / Shared: Can be used by multiple client application. Stored in centralized shared folder called GAC (Global Assembly Cache).

Introduction to C# Programming

C# Programming

- C# is a modern, object-oriented language that enables programmers to quickly build a wide range of applications for the new Microsoft .NET platform, which provides tools and services that fully exploit both computing and communications.
- C# supports the concepts of encapsulation, inheritance, and polymorphism.
- C# programming language can be used to develop different types of secured and robust applications.
  - Console Applications
  - Windows Applications
  - Web Applications
  - Distributed Applications
  - Web Service Applications

Introduction to C# Programming

C# Features

- Pointers are missing in C#.
- Unsafe operations such as direct memory manipulation are not allowed.
- Automatic Memory Management and Garbage Collection.
- Varying ranges of the primitive types like Integer, floats etc.
- Very powerful and simple for building interoperable, scalable, robust applications.
- Built in support to turn any component into a web service that can be invoked over the Internet from any application running on any platform.

Introduction to C# Programming

C# Features

- Object Oriented
  - Supports Data Encapsulation, inheritance, polymorphism, interfaces
  - C# introduces structures (structs) which enable the primitive types to become objects

```
int i = 1;
string a = i.Tostring(); //conversion (or) Boxing
```

- Type Safe
  - Cannot perform unsafe casts like convert double to a Boolean. Value types (primitive types) are initialized to zeroes and reference types (objects and classes are initialized to null by the compiler automatically.
  - Arrays are zero base indexed and are bound checked.
  - Overflow of types can be checked.

Introduction to C# Programming

C# Features

- Interoperable
  - It includes native support for the COM and windows based applications.
  - C# allows the users to use pointers as unsafe code blocks to manipulate your old code.
  - Components from VB NET and other managed code languages can directly be used in C#.
- Scalable and Versionable
  - NET has introduced assemblies, which are self-describing by means of their manifest. Manifest establishes the assembly identity, version, culture and digital signature etc. Assemblies need not to be registered anywhere.
  - No registering of DLL, and support versioning in the language.

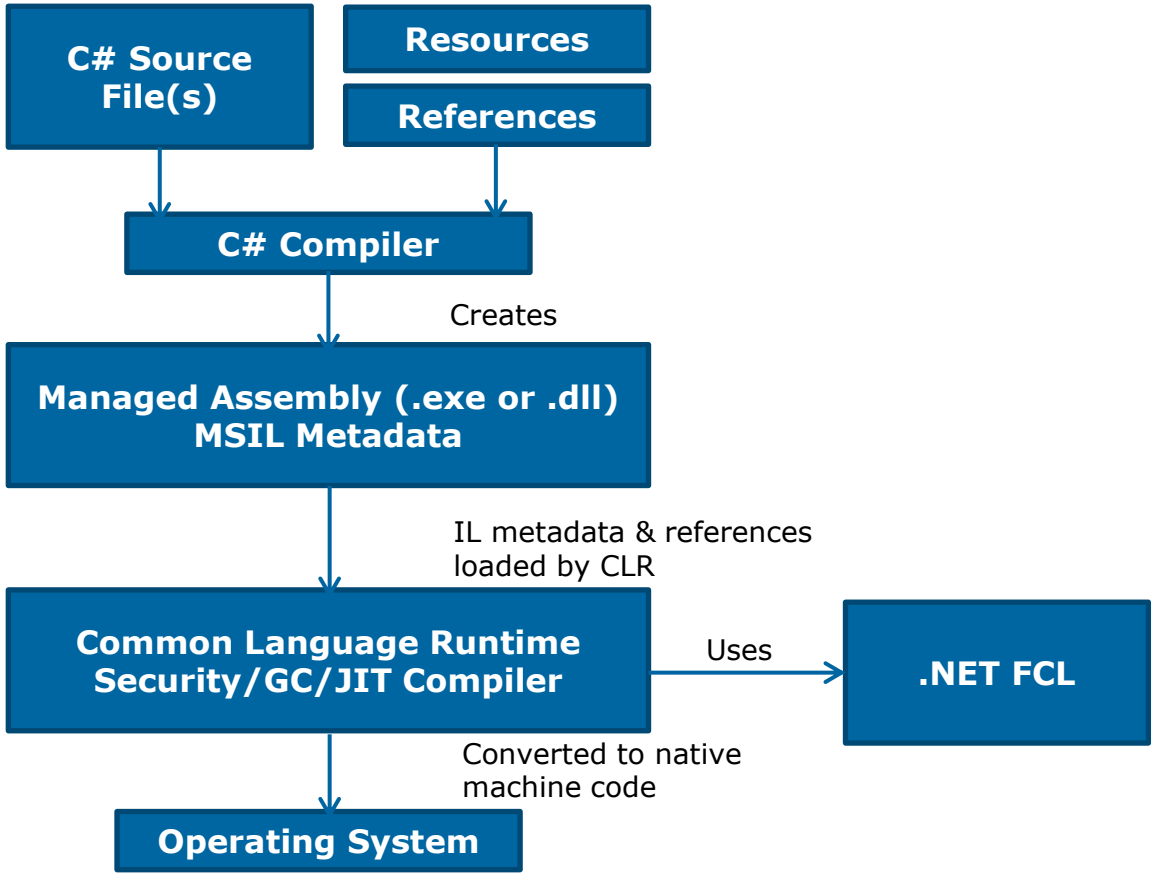
Introduction to C# Programming

C# Features

- Interoperable
  - It includes native support for the COM and windows based applications.
  - C# allows the users to use pointers as unsafe code blocks to manipulate your old code.
  - Components from VB NET and other managed code languages can directly be used in C#.
- Scalable and Versionable
  - NET has introduced assemblies, which are self-describing by means of their manifest. Manifest establishes the assembly identity, version, culture and digital signature etc. Assemblies need not to be registered anywhere.
  - No registering of DLL, and support versioning in the language.

Introduction to C# Programming

C# Program Compilation and Execution



- C# Programs can be compiled and executed using
  - **Command Prompt**

    Save the file as .cs extension:

    Hello.cs

    Compile the file using csc compiler:

    csc Hello.cs

    Execute the file:

    Hello.exe or Hello
  - **Visual Studio IDE**

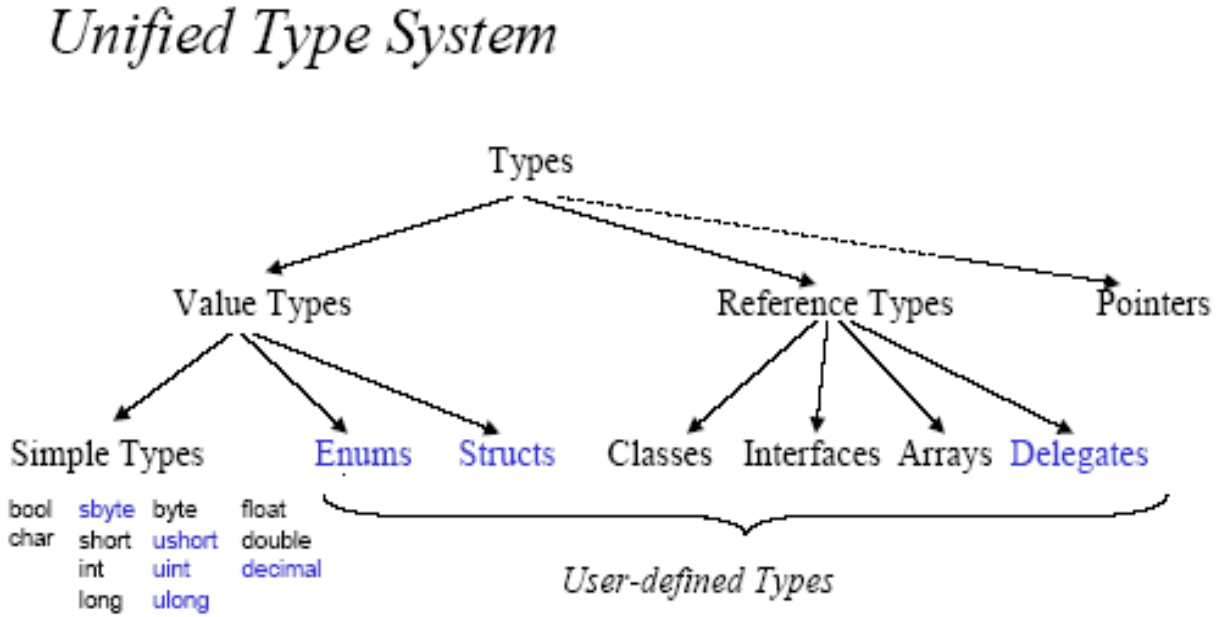
Introduction to C# Programming

Dynamic Link Library (DLL)

- A Dynamic Link library (DLL) is a library that contains functions and codes that can be used by more than one program at a time.
- Both EXE and DLL files are executable program modules, but DLL cannot runs on its own. DLL needs to be taken in EXE to run it.
- DLL files cannot be executed directly.
- Once a DLL is created, it can be used in many applications.
- To use DLL
  - Add the reference/import the DLL file

Introduction to C# Programming

Data Types



All types are compatible with *object*

- can be assigned to variables of type *object*
- all operations of type *object* are applicable to them

C# Type	CLR Type
Bool	System.Boolean
Byte	System.Byte
SByte	System.SByte
Char	System.Char
Decimal	System.Decimal
Double	System.Double
Float	System.Single
Int	System.Int32
Uint	System.UInt32
Long	System.Int64
Ulong	System.UInt64
Object	System.Object
Short	System.Int16
Ushort	System.UInt16
String	System.String

Introduction to C# Programming

Value Type and Reference Type

- Value Type: A variable of a value type always contains a value of that type. The assignment to a variable of a value type creates a copy of the assigned value
- Two Categories of value types:
  - Struct type: user-defined struct types, Numeric types, Integral types, Floating-point types, decimal, bool
  - Enumeration type
- Reference Type: Variables of reference types, referred to as objects, store references to the actual data. Assignment to a variable of a reference type creates a copy of the reference but not of the referenced object.
  - Examples of reference types: class, interface, delegate
  - Examples of built-in reference types: object, string

Introduction to C# Programming

Value Type vs Reference Type

Value Types	Reference Types
The variable contains the value directly	The variable contains a reference to the data (data is stored in separate memory area)
Allocated on stack	Allocated on heap using the new keyword
Assigned as copies	Assigned as references
Default behavior is pass by value	Passed by reference
== and != compare values	== and != compare the references, not the values
simple types, structs, enums	classes

Introduction to C# Programming

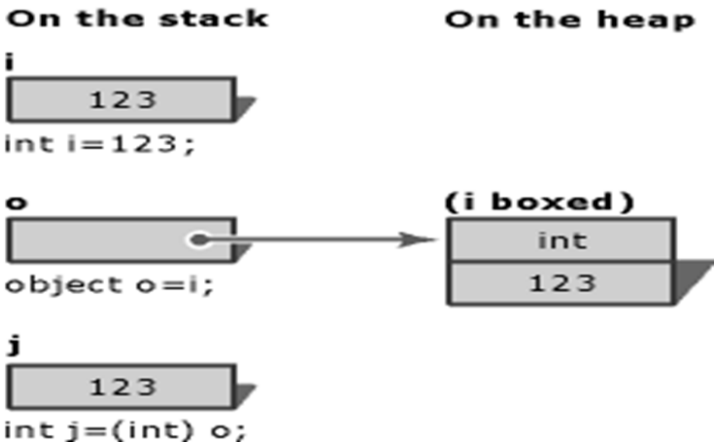
Boxing and Unboxing

- Boxing and unboxing enable value types to be treated as objects. Value types, including both struct types and built-in types, such as int, can be converted to and from the type object.
- Boxing Example:

```
int nFunny = 2000;
object oFunny = nFunny;
```

- Now both the integer variable and the object variable exist on the stack, but the value of the object resides on the heap.
- Unboxing Conversions

```
int nFunny = 2000;
object oFunny = nFunny;
int nNotSoFunny = (int)oFunny
```



Introduction to C# Programming

Arrays

- An array is a data structure that contains a number of variables called the elements of the array.
- C# arrays are zero indexed; that is, the array indexes start at zero.
- All of the array elements must be of the same type, which is called the element type of the array.
- Array elements can be of any type, including an array type.
- An array can be a single-dimensional array, a multidimensional array or a Jagged Array (Array of Arrays).
- Array types are reference type derived from the abstract base type System.Array.

Introduction to C# Programming

Arrays

- Single Dimensional Array
  - int[] array1 = new int[5];
  - string[] stringArray = new string[6];
  - int[] array2 = new int[] { 1, 3, 5, 7, 9 };
  - int[] array3 = { 1, 2, 3, 4, 5, 6 };

- Multi Dimensional Array
  - int[,] multiDimensionalArray1 = new int[2, 3];
  - int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

Introduction to C# Programming

Arrays

- Jagged Array
  - Jagged array is an array whose elements are arrays.
  - The elements of a jagged array can be of different dimensions and sizes.
  - A jagged array is sometimes called an "array of arrays."
  - The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] jaggedArray = new int[2][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Introduction to C# Programming

Arrays

- Each of the elements is a single-dimensional array of integers
- The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.
- It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size. For example:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

Introduction to C# Programming

Arrays

- Single-Dimensional Array
  - int[] numbers = new int[5] {1, 2, 3, 4, 5};
  - string[] names = new string[3] {"Matt", "Joanne", "Robert"};
  - int[] numbers = {1, 2, 3, 4, 5};
  - string[] names = {"Matt", "Joanne", "Robert"};

- Multidimensional Array
  - int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
  - string[,] siblings = new string[2, 2] { {"Mike","Amy"}, {"Mary","Albert"} };
  - int[,] numbers = new int[,] { {1, 2}, {3, 4}, {5, 6} };
  - string[,] siblings = new string[,] { {"Mike","Amy"}, {"Mary","Ray"} };
  - int[,] numbers = { {1, 2}, {3, 4}, {5, 6} };
  - string[,] siblings = { {"Mike", "Amy"}, {"Mary", "Albert"} };

Introduction to C# Programming

Nullable Types

- Nullable types represent value-type variables that can be assigned the value of null.
- You cannot create a nullable type based on a reference type.
- Reference types already support the null value
- A nullable type can represent the normal range of values for its
- underlying value type, plus an additional null value
- A Nullable<Int32>, pronounced "Nullable of Int32," can be assigned any value from -2147483648 to 2147483647, or it can be assigned the null value.
- A Nullable<bool> can be assigned the values true or false, or null.
- The ability to assign null to numeric and Boolean types is particularly useful when dealing with databases and other data types containing elements that may not be assigned a value
- Example: A Boolean field in a database can store the values true or false, or it may be undefined

OOPS

2

Atos | Syntel

OOPS

Classes

- A class is a user-defined type (UDT) that is composed of field data (member variables) and methods (member functions) that act on this data.
- C# Classes can contain the following
  - Constructors and destructors
  - Fields and constants
  - Methods
  - Properties
  - Indexers
  - Events
  - Overloaded operators
  - Nested types (classes, structs, interfaces, enumerations and delegates)

OOPS

Class Constructor

- A constructor is called automatically right after the creation of an object to initialize it.
- Constructor have the same name as the class name.
- Constructor does not have return type.
- Default constructor: if no constructor is declared, a parameterless constructor can be declared.
- A class can contain default constructor and overloaded constructors to provide multiple ways to initialise objects.
- Static constructor: similar to static method. It must be parameterless and must not have an access modifier (private, public).

OOPS

Class Destructor

- Destructors are used to destruct instances of classes.
- The Destructor is called implicitly by the .NET Framework's Garbage collector.
- An instance variable or an object is eligible for destruction when it is no longer reachable.
- Characteristics
  - A Destructor has no return type and has exactly the same name as the class name.
  - Destructors (~) are only used with classes. Cannot be inherited or overloaded.
  - Destructors (~) cannot be defined in Structs.
  - Destructor does not take modifiers or have parameters.
  - Destructor is called when program exits.
  - Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction.
  - Destructor implicitly calls Finalize on the base class of object.

OOPS

Garbage Collector (GC)

- GC is a technique in .NET Framework to free the unmanaged code objects in the memory and free the space to the process.
- When the activities of the class object in the heap memory are completed, then that object will be there as an unused space in the memory. GC now handles this object clearing in the memory implicitly.
- The heap memory is divided into the following number of generations
  - Gen 0: is for short lived objects (e.g. temp variables)
  - Gen 1: medium lived objects which are moved from gen 0.
  - Gen 2: for stable objects.
- The GC algorithm collects all unused objects that are dead objects in the generation. If the live objects running for longer time then based on the life time it will be moved to next generation.
- The object cleaning in the generation will not take place exactly after the life time over of the particular object. It takes own time to implement the sweeping algorithm to free the space to the process.

OOPS

Garbage Collector (GC)

- GC marks object as reachable and unreachable and basis of this it creates tree of reachable objects except the objects that are freed from the memory.
- Dispose and Finalize methods
  - GC only releases memory of managed resources.
  - For unmanaged resources like file handlers, network sockets and database connections etc., it is the program's responsibility to release the resources.
  - For streams, both Close() and Dispose() method will serve same purpose. Dispose() method calls Close() method internally.
  - GC Finalize() method just clean the memory used by unmanaged resource. Memory used by managed resource remains in heap as accessible reference. The memory will release whenever GC runs next time.
  - Due to Finalize() method GC will not clear entire memory associated with object in first attempt.

OOPS

Access Modifiers

Modifier	Description
public	There is no restrictions on accessing public members. Accessible from outside the assembly.
private	Access is limited to within the class definition. This is the default access modifier for member variables and methods.
protected	Access is limited to within the class definition and any class that inherits from the class.
internal	Access is limited to classes defined within the current project assembly.
protected internal	Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables and methods.
private protected	Access is limited to the containing class or types derived from the containing class within the current assembly.

OOPS

Structures

- Structs are similar to classes in that they represent data structures that can contain data members and function members.
- Unlike classes, structs are value types and do not require heap allocation.
- Structs are particularly useful for small data structures that have value semantics. The simple types provided by C#, such as int, double, and bool, are in fact all struct types
- Classes and Structs Similarities:
  - Both are user-defined types
  - Both can implement multiple interfaces
  - Both can contain
  - Data
    - Fields, constants, events, arrays
  - Functions
    - Methods, properties, indexers, operators, constructors
  - Type definitions
    - Classes, structs, enums, interfaces, delegates

OOPS

Class Vs. Structure

Class	Structure
Reference type	Value type
Can inherit from any non-sealed reference type	No inheritance (inherits only from System.ValueType)
Can have a destructor	No destructor
Can have user-defined parameterless constructor	No user-defined parameterless constructor

OOPS

Enums

- C# Enum is a value type with a set of related named constants. C# enum keyword is used to declare an enumeration. Enums are value types and are created on the stack.
- It is a user defined primitive data type. Enums type can be an integer (float, int, byte, double etc.).
- The default underlying type of the enumeration elements is int.
- By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. The default index value can be changed.
- Enum is used to create enumerated list of numeric constants having symbolic names.
- Enums are strongly typed constants. They are strongly typed, i.e., an enum of one type may not be implicitly assigned to an enum of another type even if underlying value of their members are the same.
- enum values are fixed. enum can be displayed as a string and processed as an integer.
- Every enum type automatically derives from System.Enum and thus System.Enum methods can be implemented on enums.

OOPS
Properties

- Properties can be used protect a field in a class by reading and writing to it through the property.
- In other languages, this is often accomplished by programs implementing specialized getter and setter methods.
- A property without a set accessor is considered read-only.
- A property without a get accessor is considered write-only.
- A property that has both assessors is read-write.
- Properties have many uses:
  - They can validate data before allowing a change.
  - they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database;
  - they can take an action when data is changed, such as raising an event, or changing the value of other fields.

OOPS

Method and Method Parameters

- A method is a member that implements a computation or action that can be performed by an object or class. Methods are declared using method-declaration:

  [attributes] [method-modifiers] return-type method-name-identifier ([formal-parameter-list]){ [statements]}

- There are 3 kinds of parameters:
  - out
  - ref
  - Params
- ref and out keywords in C# are used to pass arguments within a method or function. Both indicate that an argument/parameter is passed by reference. By default parameters are passed to a method by value. By using these keywords (ref and out) parameters are passed by reference.
- params are used for passing variable length of arguments to a method.

OOPS

Ref Vs. Out

- Both ref and out are treated differently at run time and they are treated the same at compile time.

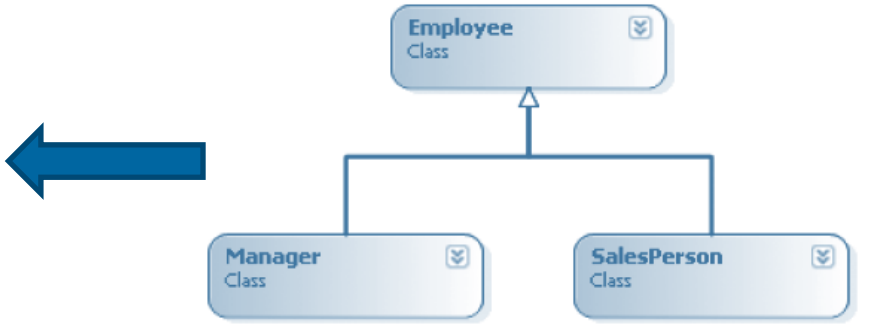
Ref	Out
Passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter.	Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method.
It is not compulsory to initialize a parameter value before using it in a calling method.	A parameter value must be initialized within the calling method before its use.
The parameter or argument must be initialized first before it is passed to ref.	It is not compulsory to initialize a parameter or argument before it is passed to an out.
In ref, data can be passed bi-directionally.	In out, data is passed in a unidirectional way (from the called method to the caller method).

OOPS

Inheritance

- Inheritance is a form of software reusability in which classes are created by reusing data and behaviours of an existing class with new capabilities.
- A class inheritance hierarchy begins with a base class that defines a set of common attributes and operations that it shares with derived classes.
- A derived class inherits the resources of the base class and overrides or enhances their functionality with new capabilities.
- The classes are separate, but related. Inheritance is also called “is a” relationship

- A SalesPerson “is-a” Employee (as is a Manager)
- Base classes (such as Employee) are used to define general characteristics that are common to all descendents
- Subclasses (such as SalesPerson and Manager) extend this general functionality while adding more specific behaviours.



OOPS

Method Overloading

- Method overloading allows programmers to use multiple methods with the same name. The methods are differentiated with their number and type of method arguments.
- Method overloading is an example of the polymorphism feature of an object oriented programming language.
- Method overriding is one of the ways by which C# achieve Compile Time Polymorphism(Static Polymorphism).
- Method overloading can be achieved by the following:
  - By using different data types for parameters.
  - By changing number of parameters in a method.
  - By changing the order of parameters in a method.

OOPS

Method Overriding

- Method Overriding is a technique that allows the invoking of functions from base class in the derived class. Creating a method in the derived class with the same signature as a method in the base class is called as method overriding.
- Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class. Method overriding is one of the ways by which C# achieve Run Time Polymorphism(Dynamic Polymorphism).
- The method that is overridden by an override declaration is called the overridden base method. An override method is a new implementation of a member that is inherited from a base class. The overridden base method must be virtual, abstract, or override.

OOPS

Abstract Class and Method

- An abstract class is one that cannot be instantiated.
- It is intended to be used as a base class.
- May contain abstract and non-abstract function members.
- It cannot be sealed.
- Abstract methods do not have an implementation in the abstract base class and every concrete derived class must override all base-class abstract methods and properties using keyword override.
- Must belong to an abstract class
- Intended to be implemented in a derived class
- When a class has been defined as an abstract base class, it may define any number of abstract members (which is analogous to a C++ pure virtual function).
- Abstract methods can be used whenever you wish to define a method that does not supply a default implementation.

OOPS

Sealed Class

- To prevent inheritance a sealed modifier is used to define a class.
- A sealed class is one that cannot be used as a base class.
- Sealed classes can’t be abstract.
- All structs are implicitly sealed.
- Many .NET Framework classes are sealed: String, StringBuilder, and so on
- Why seal a class?
  - To prevent unintended derivation.
  - Code optimization.
  - Virtual function calls can be resolved at compile-time.

OOPS

Interfaces

- An interface defines a contract. Interface is a purely abstract class; has only signatures, no implementation. May contain methods, properties, indexers and events (no fields, constants, constructors, destructors, operators, nested types).
- Interface members are implicitly public abstract (virtual). Interface members must not be static.
- Classes and structs may implement multiple interfaces.
- Interfaces can extend other interfaces.
- A class can inherit from a single base class, but can implement multiple interfaces.
- A struct cannot inherit from any type, but can implement multiple interfaces.
- Every interface member (method, property, indexer) must be implemented or inherited from a base class.
- Implemented interface methods must not be declared as override. Implemented interface methods can be declared as virtual or abstract (i.e. an interface can be implemented by an abstract class).
- If two interfaces have the same method name, then explicit implementation is procedure is used.

OOPS

Abstract class and Interface

- An abstract class doesn't provide full abstraction but an interface does provide full abstraction. Abstract classes typically do far more than define a group of abstract methods. They are free to define public, private and protected state data, as well as any number of concrete methods that can be accessed by the subclasses. Using abstract we cannot achieve multiple inheritance but using an Interface we can achieve multiple inheritance.
- Interfaces on the other hand, are pure protocol. Interfaces never define data types, and never provide a default implementation of the methods. Every member of an interface (whether is method or property) is automatically abstract. Given that C# support single inheritance , the interface-based protocol allows a given type to implement multiple interfaces and all implemented methods has to be public . We can not declare a member field in an Interface. We can not use any access modifier i.e. public, private, protected, internal etc. because within an interface by default everything is public. An Interface member cannot be defined using the keyword static, virtual, abstract or sealed.

OOPS

Partial Classes

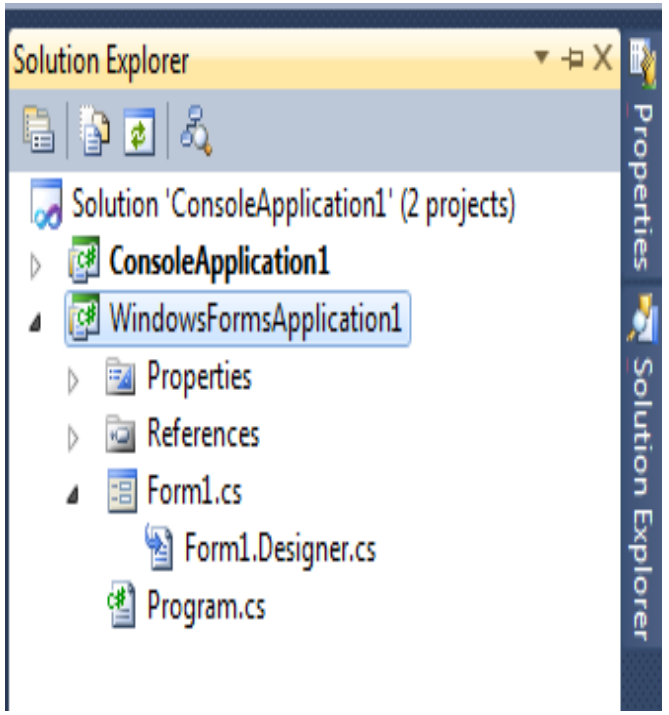
- Partial classes give you the ability to split a single class into more than one source code (.cs) file. Here's what a partial class looks like when it is split over two files:

```
//stored in file MyClass1.cs
public partial class MyClass
{
    public MethodA()
    {...}
}
```

```
//stored in file MyClass2.cs
public partial class MyClass
{
    public MethodB()
    {...}
}
```

- When the application is build, Visual Studio .NET tracks down each piece of MyClass and assembles it into a complete, compiled class with two methods, MethodA() and MethodB().

OOPS
Partial Classes




```
partial class Form1
{
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer components

    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    /// <param name="disposing">true if managed resourc
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    Windows Form Designer generated code
}
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

Delegates and Events

Atos | Syntel

Delegates and Events

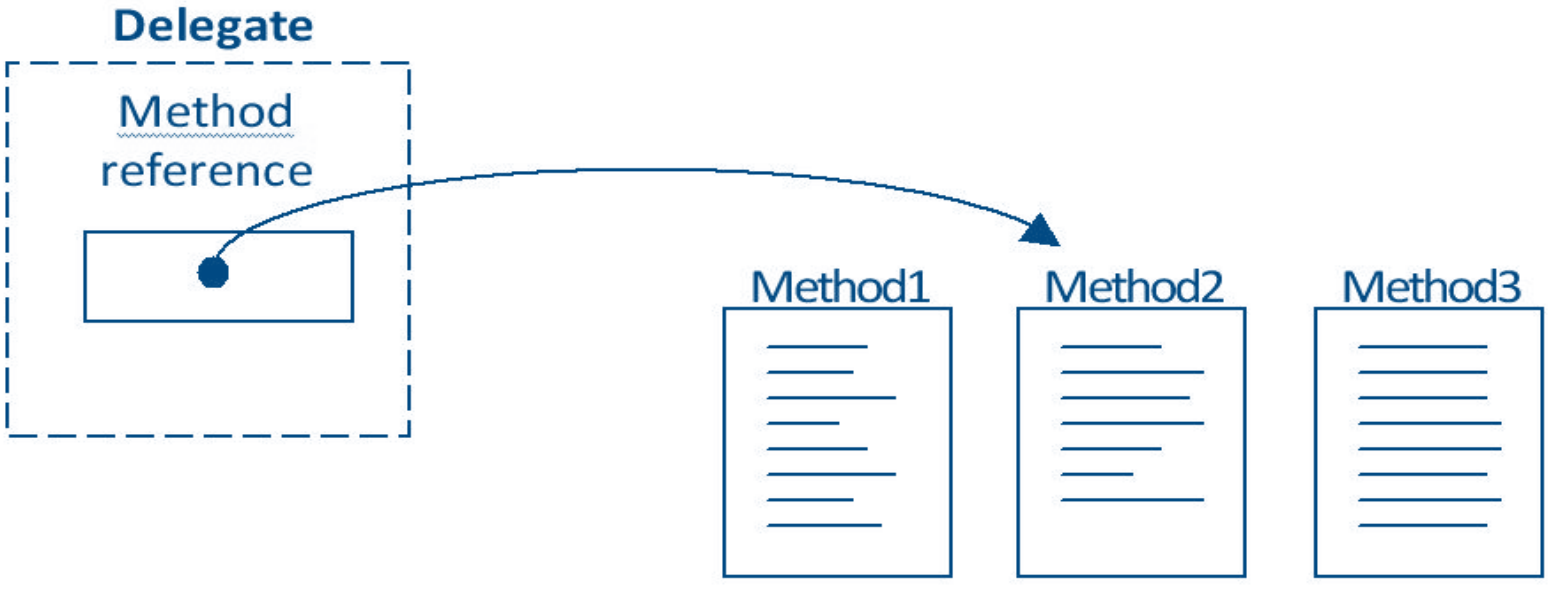
Delegates

- A delegate is a reference type that defines a method signature
- A delegate instance holds one or more methods
  - Essentially an “object-oriented function pointer”
  - Methods can be static or non-static
  - Methods can return a value
- Provides polymorphism for individual functions
- Foundation for event handling
- Two types of delegates are:
  - Singlecast delegate
  - Multicast delegate

Delegates and Events

Singlecast Delegate

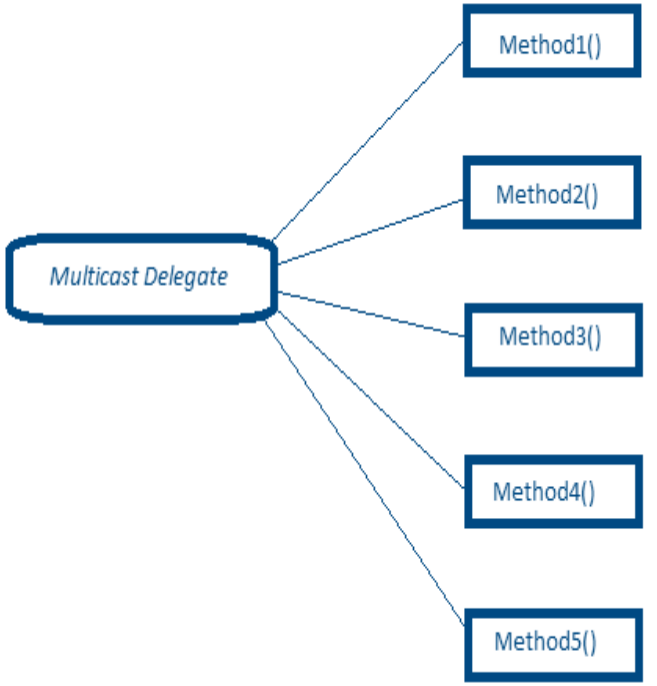
- Singlecast delegate had a reference of a single method.



Delegates and Events

Multicast Delegate

- Multicast delegate can hold and invoke multiple methods.
- Multicast delegates must contain only methods that return void.
- Each delegate has an invocation list.
- Methods are invoked sequentially, in the order added.
- The += and -= operators are used to add and remove delegates, respectively.
- += and -= operators are thread-safe.



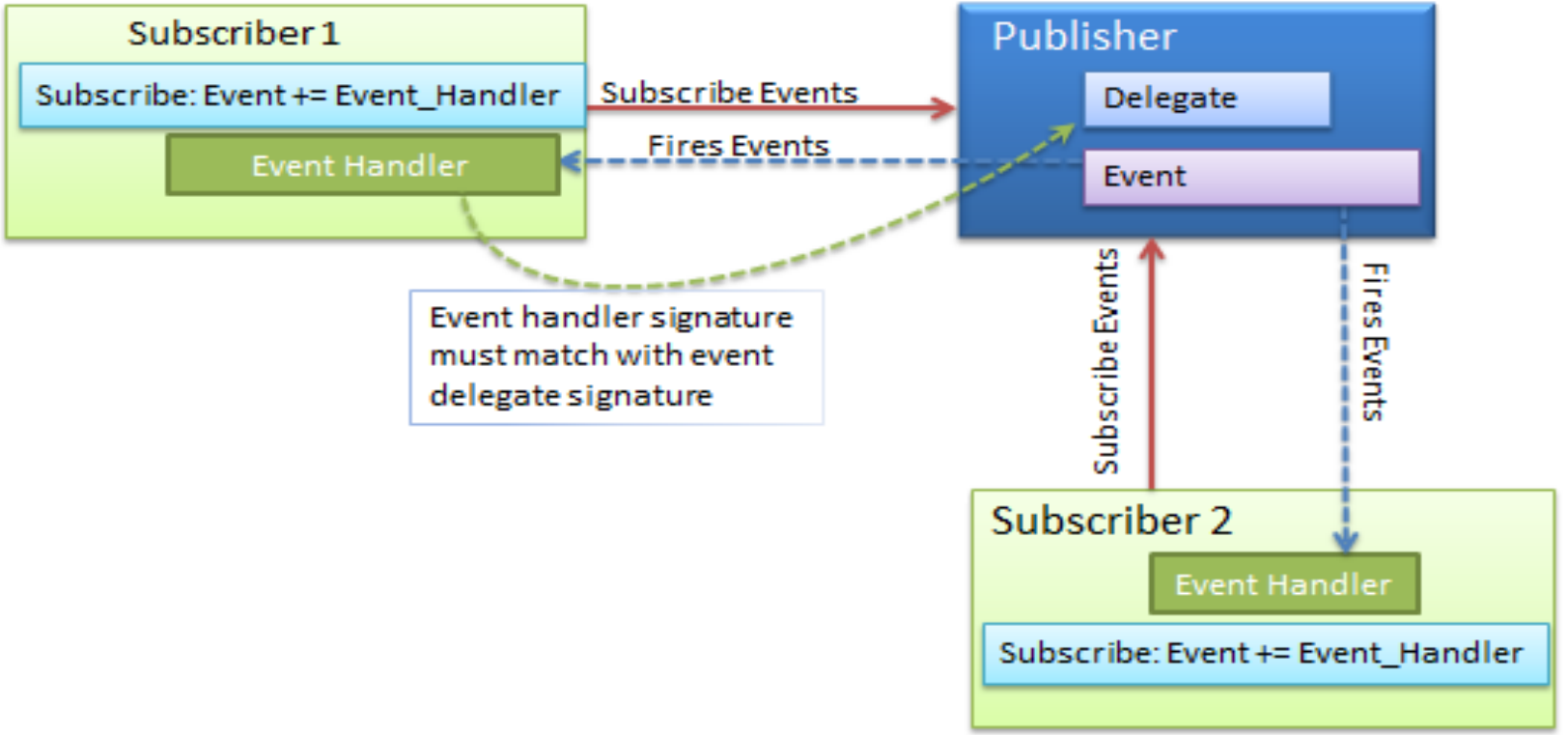
Delegates and Events

Events

- Event handling is a style of programming where one object notifies another that something of interest has occurred
- A publish-subscribe programming model.
- Events allow you to tie your own code into the functioning of an independently created component. Events are a type of “call back” mechanism
- Events are well suited for user-interfaces. E.g. the user does something (clicks a button, moves a mouse, changes a value, etc.) and the program reacts in response
- Many other uses, e.g.
  - Time-based events
  - Asynchronous operation completed
  - Email message has arrived
  - A web session has begun

Delegates and Events

Events



Delegates and Events

Events

- C# has native support for events
- Based upon delegates
- An event is essentially a field holding a delegate
- However, public users of the class can only register delegates
  - They can only call += and -=
  - They can’t invoke the event’s delegate
- Multicast delegates allow multiple objects to register with the same event.
- Anonymous Method
  - Even though only a single statement is executed in response to the button’s Click event, that statement must be extracted into a separate method with a full parameter list, and an EventHandler delegate referencing that method must be manually created.
  - Using an anonymous method, the event handling code becomes significantly more succinct.

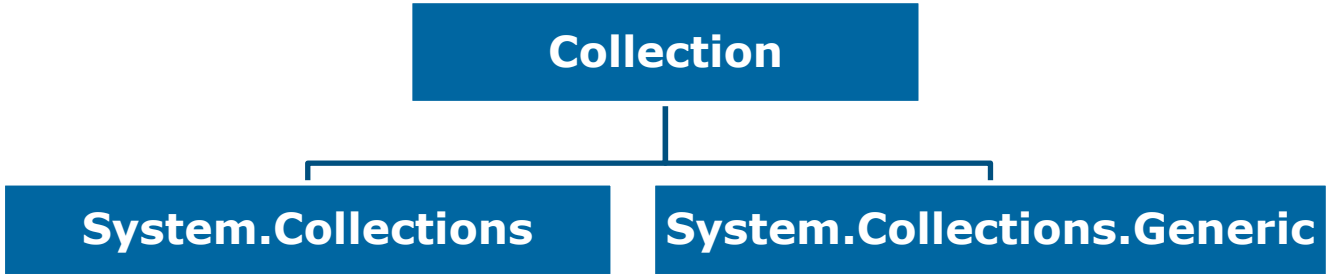
Collections and Generics

Atos | Syntel

Collections and Generics

Collections and Generics

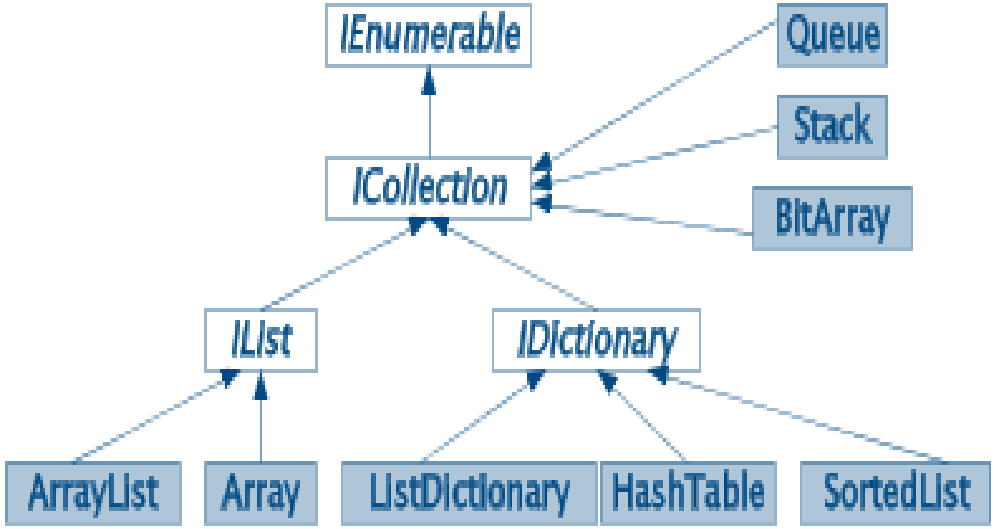
- The .NET Framework has powerful support for Collections.
- Collections are enumerable data structures that can be assessed using indexes or keys.
- Collections standardize the way in which the objects are handled by the program.
- It contains a set of classes to contain elements in a generalized manner.
- With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search, sort etc.
- There are two types of collections available in C#:



Collections and Generics

System.Collections

- Non-Generic collection in C# is defined in System.Collections namespace.
- It is a general-purpose data structure that works on object reference, but not in a safe-type manner.
- Non-generic collections are defined by the set of interfaces and classes.
- Non-generic requires type casting.



Collections and Generics

System.Collections

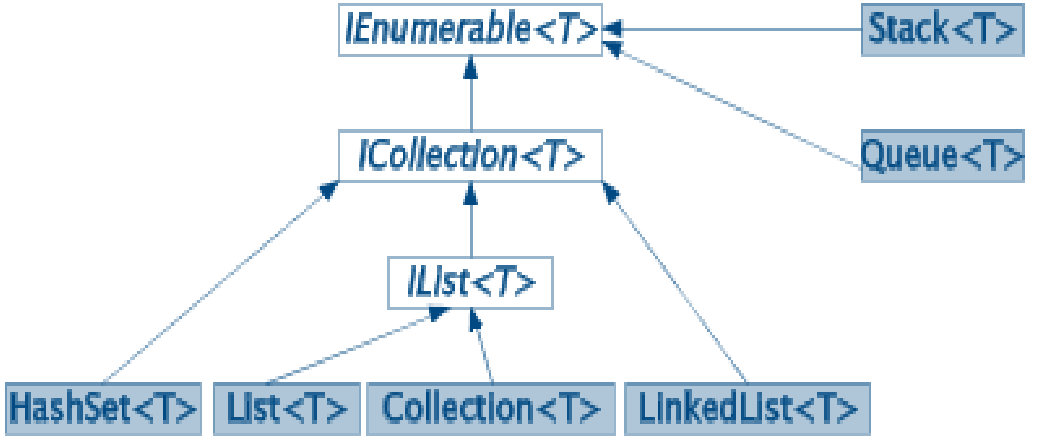
- Below table contains the frequently used classes of the System.Collections namespace:

Class	Description
ArrayList	ArrayList stores objects of any type like an array. There is no need to specify the size of the ArrayList because it grows dynamically.
Hashtable	Hashtable stores key and value pairs. It retrieves the values by comparing the hash value of the keys.
Queue	C# includes generic and non-generic Queue. Queue stores the values as FIFO (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection
Stack	C# includes generic and non-generic Stack. Stack stores the values in LIFO style (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values.
SortedList	C# includes generic and non-generic SortedList collection. SortedList stores key and value pairs. It automatically arranges elements in ascending order of key by default.

Collections and Generics

System.Collections.Generic

- Generic collection in C# is defined in System.Collections.Generic namespace.
- It provides a generic implementation of standard data structure like linked lists, stacks, queues, and dictionaries.
- These collections are type-safe because they are generic means only those items that are type-compatible with the type of the collection can be stored in a generic collection.
- Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time.
- A generic class can be defined using angle brackets <>.
- Generics eliminates type casting.
- Generic delegates allows to define in a type-safe manner, a generalized form that can then be matched to any compatible method.



Collections and Generics

System.Collections.Generic

- Below table contains the frequently used classes of the System.Collections.Generic namespace:

Class	Description
List<T>	Dynamic array that provides functionality similar to that found in the non-generic ArrayList class.
Queue<T>	FIFO list provides functionality similar to the non-generic Queue class.
Stack<T>	LIFO list provides functionality similar to the non-generic Stack class.
HashSet<T>	An unordered collection of the unique elements. It prevent duplicates from being inserted in the collection.
LinkedList<T>	It allows fast inserting and removing of elements. It implements a classic linked list.
Dictionary<TKey,TValue>	It stores key/value pairs and provides functionality similar to that found in the non-generic Hashtable class.
SortedList<TKey,TValue>	It is a sorted list of key/value pairs and provides functionality similar to that found in the non-generic SortedList class.

Collections and Generics

Collection Interfaces and Iterators

- IEnumerable:
  - An enumerator is an object that provides a forward, read-only cursor for a set of items.
  - The IEnumerable interface has one method called the GetEnumerator() method.
  - Classes that implement this method must return a class that implements the IEnumerator interface.
- IEnumerator
  - It defines the notion of a cursor that moves over the elements of a collection. It has three members for moving the cursor and retrieving elements from the collection.
  - Properties
    - Current : It returns the element at the position of the cursor.
  - Methods
    - MoveNext() : MoveNext method advances the cursor returning true if the cursor was successfully advanced to the next element and false if the cursor has moved past the last element.

Collections and Generics

Collection Interfaces and Iterators

- An iterator is a method, get accessor or operator that support foreach iteration in a class or struct without having to implement the entire IEnumerable interface. When the compiler detects the iterator, it automatically generate the Current, MoveNext and Dispose methods of the IEnumerable or IEnumerable<T> interface.
- An iterator is a section of code that returns an ordered sequence of values of the same type.
- The iterator code uses the yield return statement to return each element in turn, yield break ends the iteration.
- The return type of an iterator must be IEnumerable, IEnumerator, IEnumerable<T>, or IEnumerator<T>
- Iterators are especially useful with collection classes.

Multithreading

Multithreading

Multithreading Basics

- Multithreading is a specialized form of Multitasking.
- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.
- There are two distinct types of multitasking:
  - Process-Based
    - Is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking allows you to run a word processor at the same time you are using a spreadsheet or browsing the Internet.
    - In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
  - Thread-Based
    - Deals with concurrent execution of pieces of the same program.

Multithreading

Types of Threads

- Types of threads:
  - Foreground Thread: Foreground threads are threads which will continue to run until the last foreground thread is terminated. The application is closed when all the foreground threads are stopped. By default, the threads are foreground threads. Default value of IsBackground property of thread is false.

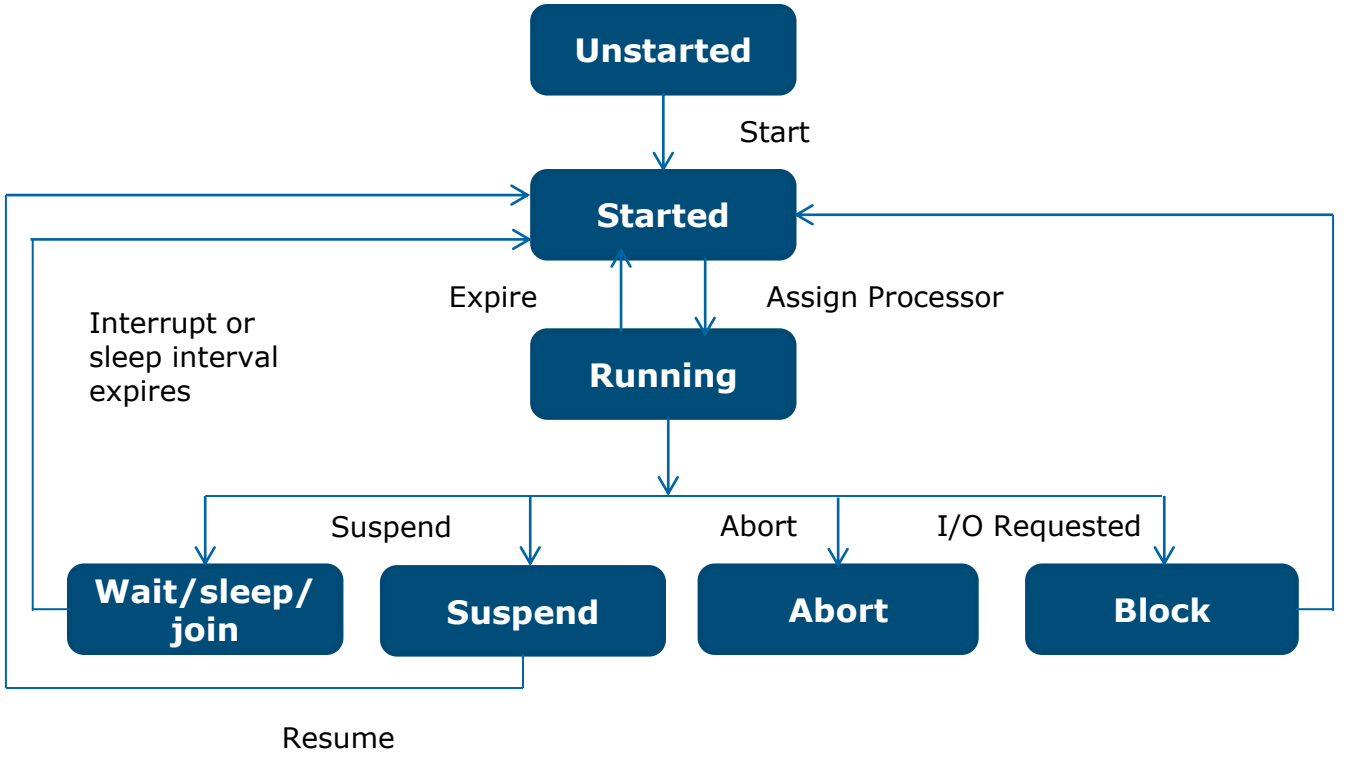
  - Background Thread: Background threads are threads which will get terminated when all foreground threads are closed. The application does not wait for them to be completed.

- Advantages of Multithreading:
  - Maximize the utilization of CPU resources.
  - It executes multiple process simultaneously.
  - Time sharing between multiple threads.

Multithreading

Thread Life Cycle

- At any given point of time, thread in c# will be in any one of the following states:
  - Unstarted
  - Runnable
  - Running
  - Not Runnable
  - Dead



Multithreading

Thread Life Cycle

- Unstarted State: When an instance of a Thread class is created, thread is in the unstarted state.
- Runnable State: A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give time to the thread to run i.e. when the start method is called.
- Running State: Thread gets the processor i.e. thread is running.
- Not Runnable State: Thread is not running because
  - Sleep() method is called.
  - Wait() method is called.
  - Due to I/O request.
  - Suspend() method is called.
- Dead State: When the thread completes its task, then thread enters into dead, terminated or abort state. Thread cannot enter the runnable state after the dead state.

Multithreading

Thread Properties

- System.Threading namespace contains Thread class.

Property	Description
CurrentThread	returns the instance of currently running thread.
IsAlive	checks whether the current thread is alive or not. Used to find the execution status of the thread.
ManagedThreadId	is used to get unique id for the current managed thread.
Name	is used to get or set the name of the current thread.
Priority	is used to get or set the priority of the current thread.
ThreadState	is used to return a value representing the thread state.

Multithreading

Thread Methods

- System.Threading namespace contains Thread class.

Method	Description
Abort()	is used to terminate the thread. It raises ThreadAbortException.
Interrupt()	is used to interrupt a thread which is in WaitSleepJoin state.
Join()	is used to block all the calling threads until this thread terminates.
Pulse()	is used to resume the suspended thread.
Sleep(milliseconds)	is used to suspend the current thread for the specified milliseconds.
Start()	changes the current state of the thread to Runnable.
Wait()	suspends the current thread if it is not suspended.
Yield()	is used to yield the execution of current thread to another thread.

Multithreading

ThreadStart and ParameterizedThreadStart Delegates

- Thread(ThreadStart) constructor is used to initialize a new instance of a Thread class. This constructor will throw ArgumentNullException if the value of the parameter is null.
- Syntax:

  public Thread(ThreadStart threadStart);
- Here, threadStart is a delegate which represents a method to be invoked when thread begins executing.
- Thread(ParameterizedThreadStart) constructor is used to initialize a new instance of the Thread class. It defined a delegate which allows an object to pass to the thread when the thread starts. This constructor throw ArgumentNullException if the parameter of constructor is null.
- Syntax:

  public Thread(ParameterizedThreadStart paramthreadStart);
- Here, paramthreadStart is a delegate which represents a method to be invoked when thread begins executing.

Multithreading

Thread Synchronization

- Synchronization is a technique that allows only one thread to access the resource for the particular time.
- No other thread can interrupt until the assigned thread finishes its task.
- In multithreading program, threads are allowed to access any resource for the required execution time. Threads share resources and executes asynchronously. Accessing shared resources (data) is critical task that sometimes may halt the system.
- Thread synchronization prevents the problems in multithreading.
- Synchronization in implemented using
  - Using lock keyword
  - Using Monitor class
  - The monitor class have two static method Wait() and Pulse().
  - The purpose of Wait and Pulse is to provide a simple signaling mechanism:
  - Wait blocks until it receives notification from another thread (Pulse provides that notification).

Multithreading
Thread Synchronization

- Wait(v) and Pulse(v) may only be called in a statement sequence that is protected with lock(v).
- Between Pulse(v) and the continuation of the awakened thread other threads may run, which in the meantime have tried to obtain the lock (i.e. the condition signalled by Pulse need not be true any more when the awakened thread resumes after Wait!)

  Therefor Wait should be enclosed in a loop that check for the continuation condition:

```
while (condition false) Monitor.Wait(v);
...
make condition true;
Monitor.Pulse(v);
```

- PulseAll(v) wakes up all threads that wait for v, but only one of them is allowed to continue. The others must wait until the previous one has released the lock. Then the next thread may enter the critical region.

Exception and Debugging

Atos | Syntel

Exception Handling

Why Exception Handling

```
readFile
{
        open the file;
        find its size;
        allocate memory;
        read file into memory;
        close the file;
}
```

ABNORMALITIES

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

Exception Handling

Exception Handling in C#

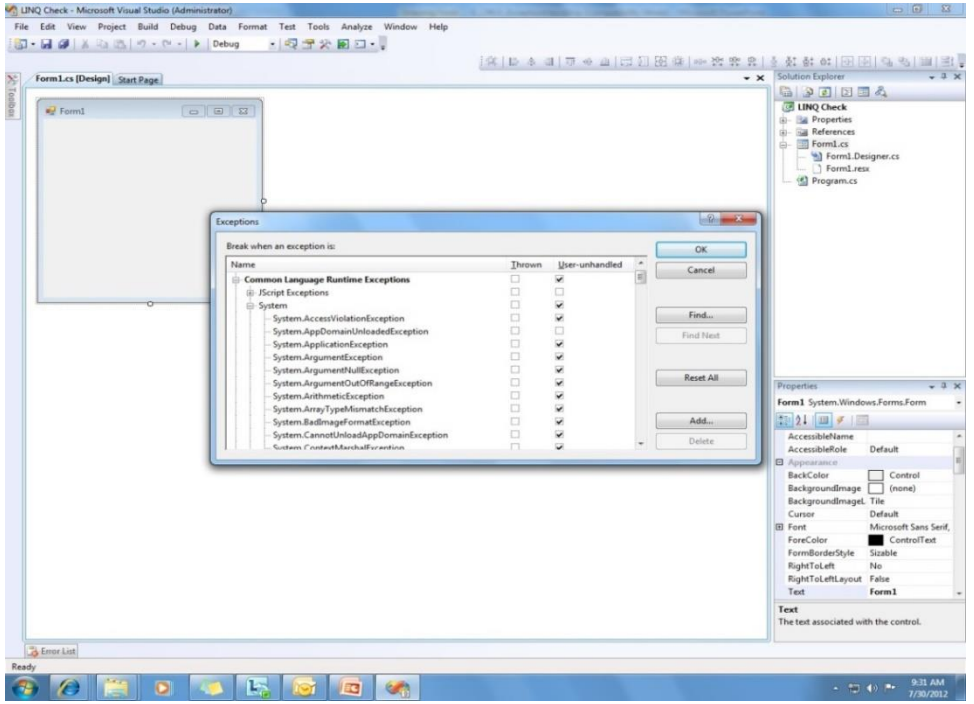
- When an application encounters an exceptional circumstance, such as a division by zero or low memory warning, an exception is generated
- Once an exception occurs, the flow of control immediately jumps to an associated exception handler, if one is present
- If no exception handler for a given exception is present, the program stops executing with an error message
- Actions that may result in an exception are executed with the try keyword.
- An exception handler is a block of code that is executed when an exception occurs. In C#, the catch keyword is used to define an exception handler.
- Exceptions can be explicitly generated by a program using the throw keyword.
- Exception objects contain detailed information about the error, including the state of the call stack and a text description of the error
- Code in a finally block is executed even if an exception is thrown, thus allowing a program to release resources.

Exception Handling

Exception Classes

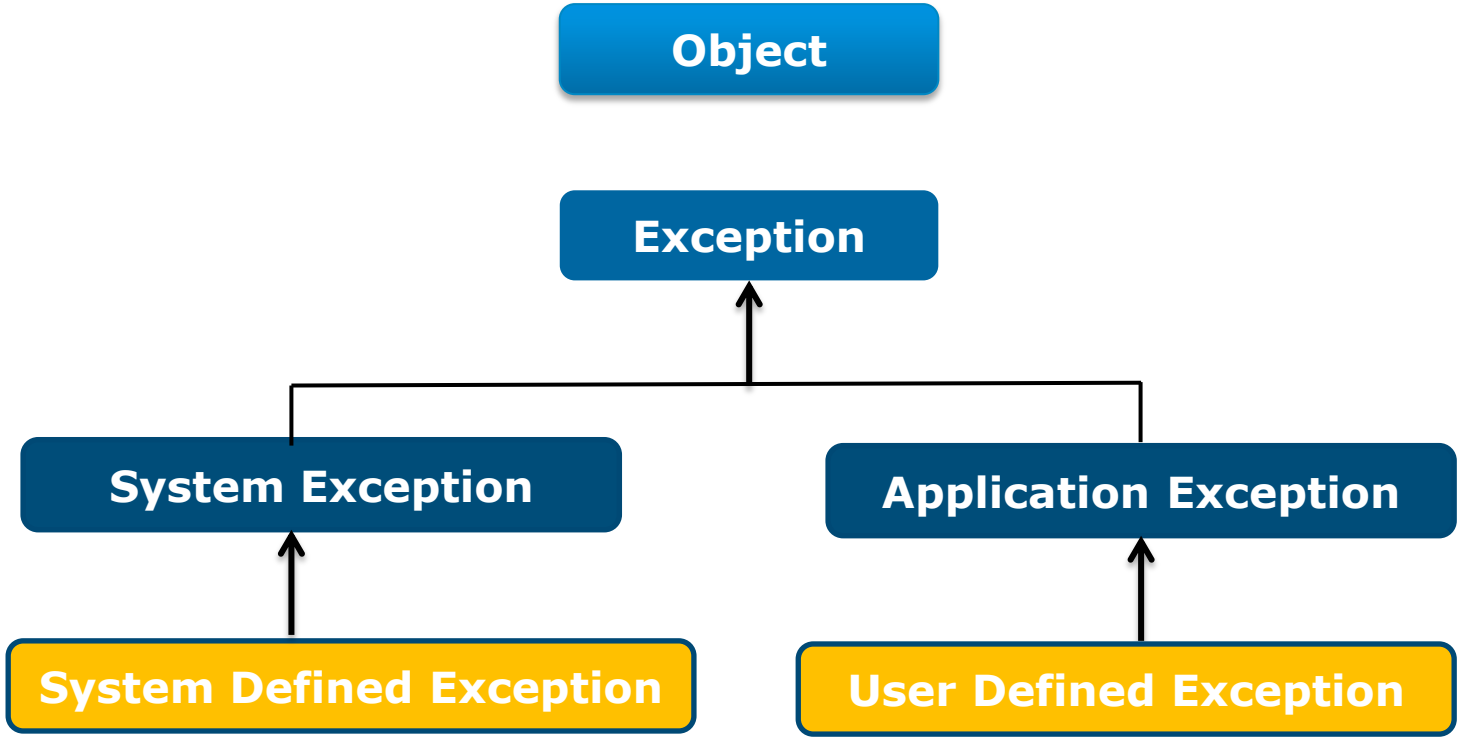
- To View Exception Class Hierarchy

  CTRL +ALT + E ➔ Common Language Runtime Exception ➔ System ➔ All Exception Classes can be viewed



Exception Handling

Exception Hierarchy



Exception Handling

System.Exception Classes

Exception	Description
System.DivideByZeroException	handles the error generated by dividing a number with zero.
System.NullReferenceException	handles the error generated by referencing the null object.
System.InvalidCastException	handles the error generated by invalid typecasting.
System.IO.IOException	handles the Input Output errors.
System.FieldAccessException	handles the error generated by invalid private or protected field access.
System.IndexOutOfRangeException	handles errors generated when a method refers to an array index out of range.
System.OutOfMemoryException	handles errors generated from insufficient free memory.

Exception Handling

Exception Handling Keywords

- try

  A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.

- catch

  A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

- finally

  The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown.

- throw

  A program throws an exception when a problem shows up. This is done using a throw keyword.

Exception Handling

Custom Exceptions

- There is a need to raise an exception when the business rule of the application gets violated.
- Hence custom exception class can be created by deriving from Exception or ApplicationException base class.

Debugging

- Debugging is the process of finding errors during application execution.
- Debugging is the routine process of locating and removing computer program bugs, errors or abnormalities, which is methodically handled by software programmers via debugging tools.
- Debugging checks, detects and corrects errors or bugs to allow proper program operation according to set specifications.
- Microsoft Visual Studio .NET includes an integrated debugger that can be used to debug Visual C# projects.
- Because the debugger is integrated into the same tool used to edit and compile your Visual C# code, the compile/test/debug cycle is greatly simplified.
- When Visual C# project is created, it has two configurations:
- Debug and Release. The Debug configuration generates symbolic information that's used by the debugger; this information isn't included when building a project with the default Release configuration.

Debugging

- Setting Breakpoints
  - Breakpoints are code locations that cause the debugger to pause (or break) execution of your program.
  - A breakpoint is typically used to halt the debugger in a location.
  - For example, let's say your application isn't responding correctly after receiving user input. By setting a breakpoint at the location in your code that handles input from the user, you cause the debugger to pause execution at the specified point.
  - While stopped at a breakpoint, you can inspect the program variables, change the path of program execution, or step through your program's instructions.
  - Breakpoints can be added on any line of your Visual C# source code that contains a program statement.
  - While your program is stopped at a breakpoint, you can inspect and modify variables in several ways.
  - If you hover the mouse pointer over a variable, the type and value of the variable will be displayed in a ToolTip.

Debugging

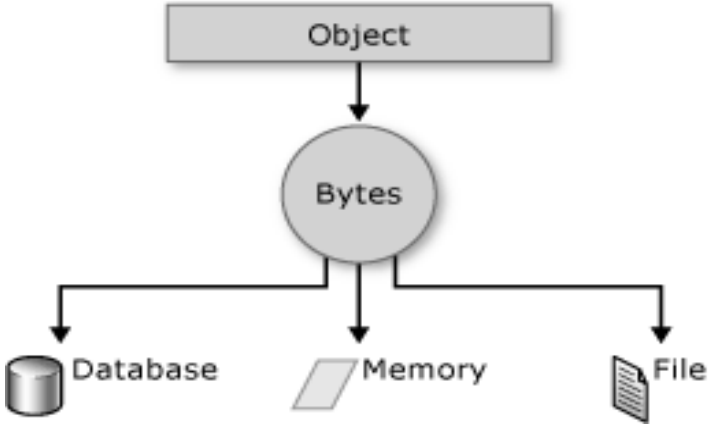
- Alternatively, following windows can also display the variables while debugging
  - Autos Contains variables referenced on the current or previous line.
  - Locals Contains variables defined in the current method..
  - Watch Contains variable names that you define while debugging. Up to four Watch windows can be open at one time.

- Debugging Process
  - Add the breakpoint
  - Start the debugger and hit breakpoints.
  - Learn commands to step through code in the debugger
  - Inspect variables in data tips and debugger windows
  - Examine the call stack

Serialization

Serialization

- Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file.
- Its main purpose is to save the state of an object in order to recreate it when needed.
- The reverse process of converting bytes into object is called deserialization.



Serialization

- Different formats into which an object can be serialized.

Serialization Type	Description
JSON serialization	JSON is an open standard used for sharing data across the web. System.Text.Json namespace contains classes for JavaScript Object Notation (JSON) serialization and deserialization.
Binary serialization	Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams. System.Runtime.Serialization namespace contains classes for binary serialization and deserialization.
XML serialization	XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. System.Runtime.Serialization namespace contains classes for XML serialization and deserialization.

Parallel Extensions and Asynchronous Programming

Atos | Syntel

Parallel Extensions

- Parallel Programming is breaking a large problem into smaller ones and executing each chunk simultaneously.
- Parallel Extensions are a set of APIs provided by Microsoft, designed to take advantage of systems that have multiple core and multiple processors using data parallelism, task parallelism, and the coordination on the parallel hardware.
- Parallel Extensions and Enhancements can be divided into
  - Task Parallel Library (TPL)
  - Parallel LINQ (PLINQ)

Parallel Extensions

- Task Parallel Library (TPL)
  - Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces.
  - The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available.
  - TPL handles the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details.
  - Using TPL, performance of the code is maximize.
  - TPL is the preferred way to write multithreaded and parallel code, starting from .NET Framework 4.0
  - Parallel loop construct is used to parallelize the application
    - Parallel.For()
    - Parallel.ForEach()

Parallel Extensions

- Parallel LINQ (PLINQ)
  - PLINQ is the implementation of LINQ to Objects that executes queries in parallel, scaled to utilize the number of processors available.
  - Executing queries with PLINQ is similar to executing queries with LINQ.
  - PLINQ supports all query operators supported by System.Collection.Enumerable.
  - LINQ to SQL and LINQ to Entity are not supported as these are executed by the database or the query operators.
  - AsParallel extension method is used for PLINQ query.
  - Using the AsParallel method makes sure that the compiler uses ParallelEnumerable<T> instead of Enumerable<T> of query interfaces.

Asynchronous Programming

- Asynchronous Programming allows writing asynchronous code without having to wait for the other task to complete.
- It follows Task-based Asynchronous Pattern (TAP).
- The core of async programming is the Task<> and Task<T> objects, which models asynchronous operations. They are supported by async and await keyword.
- Async programming model:
  - For I/O-bound code, await an operation that returns a Task or Task<T> inside of an async method.
  - For CPU-bound code, await an operation that is started on a background thread with the Task.Run method.
- The await keyword yields control to the caller of the method that performed await, and it ultimately allows a UI to be responsive or a service to be elastic.
- Increases the performance and responsiveness of the application.

Asynchronous Programming

- Key points to understand:
  - await can only be used inside an async method.
  - async code is used differently for both I/O bound and CPU bound code.
  - async code uses Task<T> and Task, which are constructs used to model work being done in the background.
  - The async keyword turns a method into an async method, which allow to use the await keyword in its body.
  - When the await keyword is applied, it suspends the calling method and yields control back to its caller until the awaited task is complete.
  - Task.WhenAll and Task.WhenAny, that allow you to write asynchronous code that performs a non-blocking wait on multiple background jobs.
  - async void should only be used for event handlers.

Asynchronous Programming

- Structure of async/await method:
  - To make a method asynchronous with async/await keywords, following needs to be implemented
    - Method definition should include the keyword async, this keyword by itself doesn't do anything except enabling us to use the keyword await within the method.
    - Method return type should change to return either void or Task or Task<T> , where T is the return data type.
    - Example

      public async Task<String> GetUserDataAsync(){ }
    - According to the naming convention, an asynchronous method name should end with the word 'Async`. Example, If the method name is GetUserData() then the async method will be

      GetUserDataAsync(){}

Lambda Expressions

Atos | Syntel

Lambda Expressions

- Lambda expressions are used like anonymous functions. It is a concise way to represent an anonymous method.
- => operator is used in all lambda expressions
- Lambda expressions does not require to specify the type of value that is input making it more flexible.
- Lambda expression is divided into two parts, the left side is the input and the right is the expression.
- Lambda expressions can be of two types:
  - Expression Lambda: Consists of input and expression.
    - input => expression
  - Statement Lambda: Consists of input and a statement block to be executed.
    - input => { statements };
- Input parameters of lambda expressions are given in parenthesis and zero input parameters with empty parenthesis.

Lambda Expressions

- Async lambda incorporate asynchronous processing by using the async and await keywords.
- Lambda expressions can also be assigned to delegates.
  - E.g. Func<in T, out TResult> type delegate and Action<> delegate.
- Lambda expressions can be invoked in a similar way to delegate.
- Lambda expressions are also used in LINQ queries.
- Lambda expressions are used in the construction of expression tree. An expression tree is similar to a data structure resembling a tree in which every node is itself an expression like a method call, or it can be a binary operation .
- Unsafe code inside any lambda expression cannot be used.

Code Optimization and Performance

Code Optimization and Performance

- Optimization is a program transformation technique to improve the code for consuming less resources (CPU, Memory) and deliver high speed.
- In optimization, high level general programming constructs are replaced by efficient low-level programming codes.
- Rules for code optimization
  - Optimization must increase the speed of the program.
  - The output code must not change the meaning of the program.
  - Optimization should itself be fast and should not delay the overall compiling process.
- Optimization can be made at various levels
  - At the beginning
  - After generating intermediate code
  - While producing the target machine code.

Code Optimization and Performance

- **-optimize**
  - C# compiler option. Enables or disables optimizations performed by the compiler to make the output file smaller, faster and more efficient. Optimizes the code at runtime.
- Avoid value types where they must be boxed a high number of times. E.g. instead of using non-generic class like System.Collections.ArrayList use System.Collections.Generic.List<T>. Boxing and unboxing are computationally expensive processes. When a value type is boxed, a new object must be created. This can take up to 20 times longer than a simple reference assignment. In unboxing, the casting process can take four times as long as an assignment.
- Use System.Text.StringBuilder instead of C# + operator for concatenating large number of string variables.
- Empty destructors should not be used. When the destructor is called, the garbage collector is invoked to process the queue. If the destructor is empty this results in loss of performance.

Code Optimization and Performance

- **Exception optimization**
  - Throwing exceptions can be very expensive. Applications must not throw many exceptions. Runtime can throw exceptions on its own even if explicit exceptions are not thrown.
- **Loop optimization**
  - Use for loop instead of for-each loop for string iteration. String are simple character arrays, they can be walked using much less overhead than other structures.
- **Array Optimization**
  - Use jagged arrays. JIT optimizes jagged array more efficiently than rectangular arrays.
- **Async Optimization**
  - Using asynchronous IO operation gives ten times better performance.

Code Optimization and Performance

- **Arithmetic Expression optimization**
  - Instead of computing arithmetic results using a series of local variables, using a compound expression and storing it in a single variable is faster.
- **Inline optimization**
  - Inlining a method body affects performance.
- **Parameter optimization**
  - Excess parameters reduce method call performance. Removing unneeded parameters will reduce the memory usage on the stack and improve performance.

Code Optimization and Performance

- Visual Studio Code Analysis feature performs static code analysis to identify interoperability, security, performance and other categories of potential problems.
- It is used to improve the code quality.
- Code metrics is a set of important software tools for measuring the quality, code maintainability, and complexity of code written.
- These metrics also provide better insight into the code by understanding the types of classes/methods/modules that should be reworked and thoroughly tested, identify potential risks, current state of code, track progress of development etc.
- Code metrics is used to validate code, improve code quality and make the application maintainable and less complex.

Thank you

For more information please contact:
T+ 33 1 98765432
M+ 33 6 44445678
firstname.lastname@atos.net

Atos | Syntel