**DOT NET PROGRAMMING**

CHAPTER - 3 : Exception & object life time and Interface and Collections

SESSION – 22 : The Basics of Object Life Time, The Role of Application Roots ,Understanding Object Generations,

# The Basics of Object Life Time

Object lifetime is the time when a block of memory is allocated to this object during some process of execution and that block of memory is released when the process ends. Once the object is allocated with memory, it is necessary to release that memory so that it is used for further processing, otherwise, it would result in memory leaks. We have a class in .Net that releases memory automatically for us when the object is no longer used. We will try to understand the entire scenario thoroughly of how objects are created and allocated memory and then deallocated when the object is out of scope.

The class is a blueprint that describes how an instance of this type will look and feel in memory. This instance is the object of that class type. A block of memory is allocated when the **new** keyword is used to instantiate the new object and the constructor is called. This block of memory is big enough to hold the object. When we declare a class variable it is allocated on the stack and the time it hits a new keyword and then it is allocated on the **heap**. In other words, when an object of a class is created it is allocated on the heap with the C# **new** keyword operator. However, a new keyword returns a reference to the object on the heap, not the actual object itself. This reference variable is stored on the stack for further use in applications.

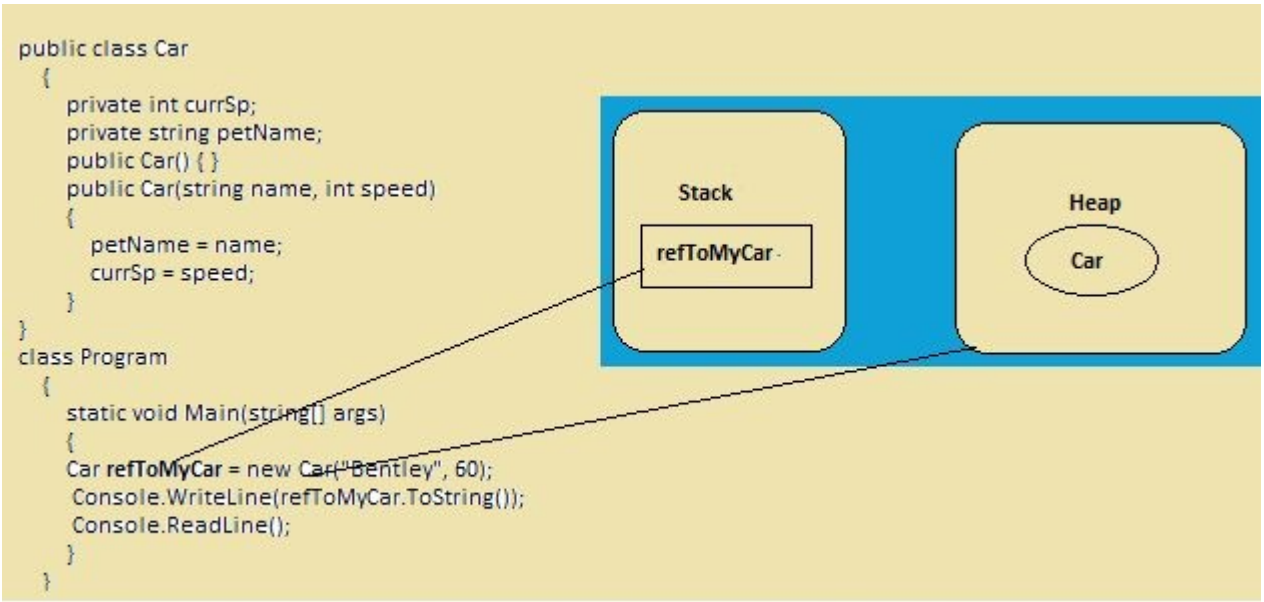



Diagram - new keyword returns a reference to the object on the heap and the actual reference variable is stored on stack.

When the new operator is used to create an object, memory is taken from the managed heap for this object and the managed heap is more than just a

random chunk of memory accessed by the CLR. When the object is no longer used then it is de-allocated from the memory so that this memory can be reused.

The key pillar of the .NET Framework is the automatic garbage collection that manages memory for all .NET applications. When an object is instantiated the garbage collector will destroy the object when it is no longer needed. There is no explicit memory deallocation since the garbage collector monitors unused objects and does a collection to free up memory that is an automatic process. The Garbage Collector removes objects from the heap when they are unreachable by any part of your codebase. The .Net garbage collector will compact empty blocks of memory for the purpose of optimization.

The heap is categorized into three generations so it can handle long-lived and short-lived objects. Garbage collection primarily occurs with the reclamation of short-lived objects that typically occupy only a small part of the heap.

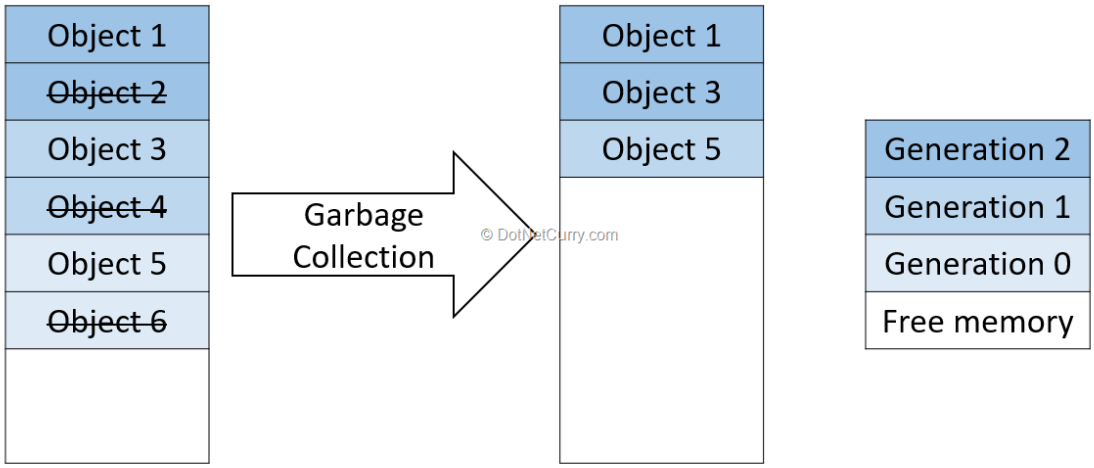
## Generations

There are the following three generations of objects on the heap:

- **Generation 0:** Newly created objects are in Generation 0. These objects on Generation 0 are collected frequently to ensure that short-lived objects are quickly collected and the memory is released. Objects that survive Generation 0, the collections are promoted to Generation 1. Most objects are reclaimed for garbage collection in Generation 0 and do not survive to the next generation.

- **Generation 1:** Objects that are collected less frequently than Generation 0 and contains longer-lived objects that were promoted from Generation 0. Objects that survive Generation 1, collection are promoted to Generation 2.

**Generation 2:** Objects promoted from Generation 1 that are the longest-lived objects and collected infrequently. The overall strategy of the garbage collector is to collect and move longer-lived objects less frequently.




The garbage collector cleans up **managed resources** automatically since the managed code is directly targeted by the CLR. But when the object uses **unmanaged resources** like database connections or file manipulation, that needs to be released manually and this can be done by a finalize method.

We use the destructor method using the (~) sign in our code to destroy the objects and this destructor is converted into a finalize method (check in the compiled code). This is known as a finalization process. If we are implementing Finalize(), we do not have control since when this method should be called the garbage collector takes care of this on its own. In the finalization process, there are two collection cycles to completely release the object's memory. During the first collection pass, the object is flagged for finalization. After the finalization occurs, the garbage collector can reclaim the object's memory and the memory is released.

There is another method, Dispose(), that releases managed and unmanaged resources explicitly. This method is the single method in an IDisposable interface and can be used to release unmanaged resources manually.

I have written another article on the IDisposable pattern in which we would have a clear picture of the difference between Finalize() and Dispose ().

---

# The Role of Application Roots

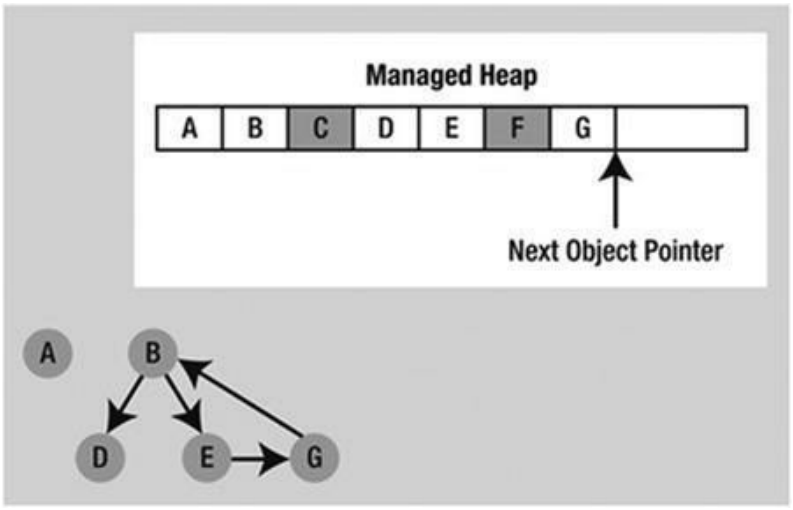
how the garbage collector determines when an object is no longer needed. To understand the details, you need to be aware of the notion of application roots. Simply put, a root is a storage location containing a reference to an object on the managed heap. Strictly speaking, a root can fall into any of the following categories:

- References to global objects (though these are not allowed in C#, CIL code does permit allocation of global objects)
- References to any static objects/static fields
- References to local objects within an application's code base
- References to object parameters passed into a method
- References to objects waiting to be finalized
- Any CPU register that references an object

During a garbage collection process, the runtime will investigate objects on the managed heap to determine whether they are still reachable (i.e., rooted) by the application. To do so, the CLR will build an object graph, which represents each reachable object on the heap.

Note that the garbage collector will never graph the same object twice, thus avoiding the nasty circular reference count found in COM programming.

Assume the managed heap contains a set of objects named A, B, C, D, E, F, and G. During a garbage collection, these objects (as well as any internal object references they may contain) are examined for active roots. After the graph has been constructed, unreachable objects (which you canassume are objects C and F) are marked as garbage. Figure ,diagrams a possible object graph for



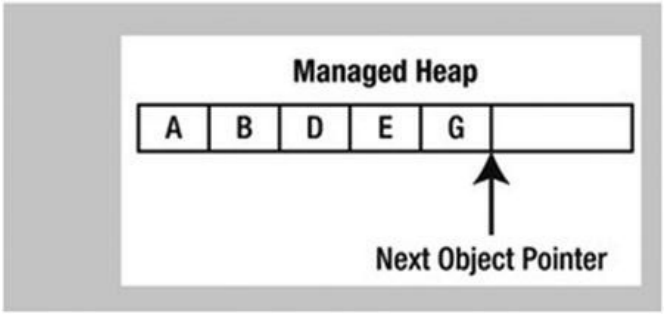

the scenario just described

Figure , Object graphs are constructed to determine which objects are reachable by application roots

After objects have been marked for termination (C and F in this case—as they are not accounted for in the object graph), they are swept from memory. At this point, the remaining space on the heap is compacted, which in turn causes the CLR to modify the set of active application roots (and the underlying pointers) to refer to the correct memory location (this is done automatically and transparently). Last but not least, the next object pointer is readjusted to point to the next available slot. Figure illustrates the resulting readjustment.

Figure , A clean and compacted heapNote Strictly speaking, the garbage collector uses two distinct heaps, one of which is specificallyused to store large objects. This heap is less frequently consulted

during the collection cycle, given possible performance penalties involved with relocating large objects. Regardless, it is safe to consider the managed heap as a single region of memory.



---

## Understanding Object Generations

When the CLR is attempting to locate unreachable objects, it does not literally examine every object placed on the managed heap. Doing so, obviously, would involve considerable time, especially in larger (i.e., real-world) applications.

To help optimize the process, each object on the heap is assigned to a specific “generation.” The idea behind generations is simple: the longer an object has existed on the heap, the more likely it is to stay there. For example, the class that defined the main window of a desktop application will be in memory until the program terminates. Conversely, objects that have only recently been placed on the heap (such as an object allocated within a method scope) are likely to be unreachable rather quickly.

Given these assumptions, each object on the heap belongs to one of the following generations:

Generation 0: Identifies a newly allocated object that has never been marked for

collection

Generation 1: Identifies an object that has survived a garbage collection (i.e., it was marked for collection but was not removed because the sufficient heap spacewas acquired)

Generation 2: Identifies an object that has survived more than one sweep of the garbage collector

The garbage collector will investigate all generation 0 objects first. If marking and sweeping (or said more plainly, getting rid of) these objects results in the required amount of free memory, any surviving objects are promoted to generation 1. To see how an object’s generation affects the collection process, ponder

in Figure , which diagrams how a set of surviving generation 0 objects (A, B, and E) are promoted once the required memory has been reclaimed.

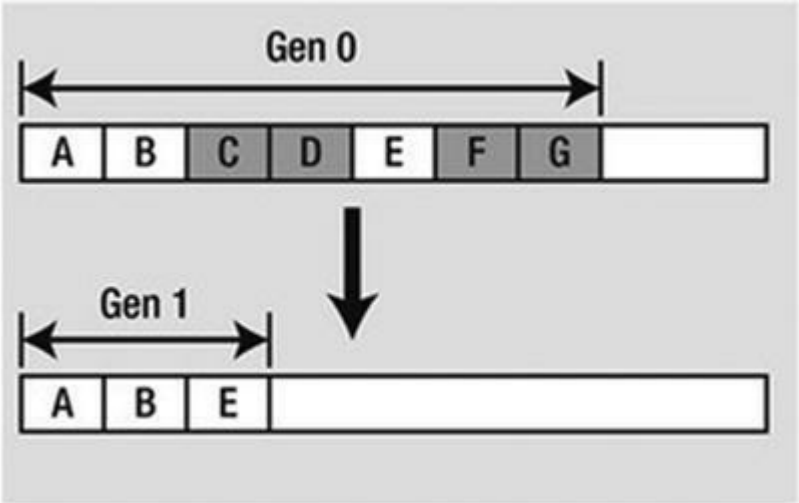


Figure 13-5. Generation 0 objects that survive a garbage collection are promoted to generation 1If all generation 0 objects have been evaluated but additional memory is still required, generation

1 objects are then investigated for reachability and collected accordingly. Surviving generation 1

objects are then promoted to generation 2. If the garbage collector still requires additional memory,

generation 2 objects are evaluated. At this point, if a generation 2 object survives a garbage

collection, it remains a generation 2 object, given the predefined upper limit of object generations.

The bottom line is that by assigning a generational value to objects on the heap, newer objects (such as local variables) will be removed quickly, while older objects (such as a program's main window) are not "bothered" as ofte

# The Role of .NET Exception hanlding

Prior to .NET, error handling under the windows operating system was a confused mishmash of techniques. Many Programmers rolled their own error handling logic within the context of a given application. For example, a development team may define a set of numerical constants that represent known error conditions, and make use of them as return values. In addition to a developer's ad-hoc techniques, the Windows API defines hundreds of error codes that comes by the way of various methods and techniques.

The obvious problem with these techniques is lack of symmetry. Each approach is more or less tailored to a given technology, a given language, and perhaps even a given project. In order to put up an end, .NET platform provides a standard technique to send and trap run time errors: Structured Exception Handling [SEH].
The beauty of this approach is that developers now have a unified approach to error handling, which is common to all languages targeting the .NET Universe, As an added bonus, the syntax used to throw and catch exceptions across assemblies and machine boundaries is identical. Another bonus of .NET exceptions is that fact that rather than receiving a cryptic numerical value that identifies the problem at hand, exception are objects that contain a human readable descriptions

### The System.Exception Base Class

All user- and system-defined exceptions ultimately derive from the System.Exception base class (which in turn derives from System.Object). Note that some of these members are virtual and may thus be overridden by derived types:

```
public class Exception : ISerializable, _Exception {

        public virtual IDictionary Data { get; }
        protected Exception(SerializationInfo info, StreamingContext context);
        public Exception(string message, Exception innerException);
        public Exception(string message);
        public Exception();

public virtual Exception GetBaseException();
public virtual void GetObjectData(SerializationInfo info, StreamingContext context); public System.
Type GetType();
protected int HResult { get; set; }
public virtual string HelpLink { get; set; }
public System.Exception InnerException { get; }
 public virtual string Message { get; }
public virtual string Source { get; set; }
public virtual string StackTrace { get; }
 public MethodBase TargetSite { get; }
 public override string ToString(); }
```

## Core Members of the System.Exception Type

| System.Exception Property | Meaning in Life |
|---|---|
| Data | Data This read-only property retrieves a collection of key/value pairs (represented by an object implementing IDictionary) that provide additional, programmer-defined information about the exception. By default, this collection is empty. |
| HelpLink | This property gets or sets a URL to a help file or web site describing the error in full deta |
| InnerException | This read-only property can be used to obtain information about the previous exception(s) that<br>caused the current exception to occur. The previous exception(s) are recorded by passing them into the constructor of the most current exception. |
| Message | This read-only property returns the textual description of a given error. The error message itself is set as a constructor parameter. |

| | |
|---|---|
| Source | This property gets or sets the name of the assembly, or the object, that threw the current exception. |
| StackTrace | This read-only property contains a string that identifies the sequence of calls that triggered the exception. As you might guess, this property is useful during debugging or if you want to dump the error to an external error log. |
| TargetSite | This read-only property returns a MethodBase object, which describes numerous details about the method that threw the exception (invoking ToString() will identify the method by name) |

**Properties of the Exception Class:** The Exception class has many properties which help the user to get information about the exception during the exception.

- **Message:** This property helps to provide the details about the main cause of the exception occurrence.
- **InnerException:** This property helps to provide the information about the series of exceptions that might have occurred.

- **Data:** This property helps to get the information about the arbitrary data which is held by the property in the key-value pairs.
- The Data property of System.Exception allows you to fill an exception object with relevant
- auxiliary information (such as a timestamp).
- The Data property returns an object implementing an interface named IDictionary, defined in the System.Collections namespace

- **TargetSite:** This property helps to get the name of the method where the exception will throw.
    - The System.Exception.TargetSite property allows you to determine various details about the method that threw a given exception. As shown in the previous Main() method, printing the value of TargetSite will display the return type, name, and parameter types of the method that threw the exception. However, TargetSite does not return just a vanilla-flavored string but rather a strongly typed System.Reflection.MethodBase object. This type can be used to gather numerous details regarding the offending method, as well as the class that defines the offending method.

- **HelpLink:** This property helps to hold the URL for a particular exception.

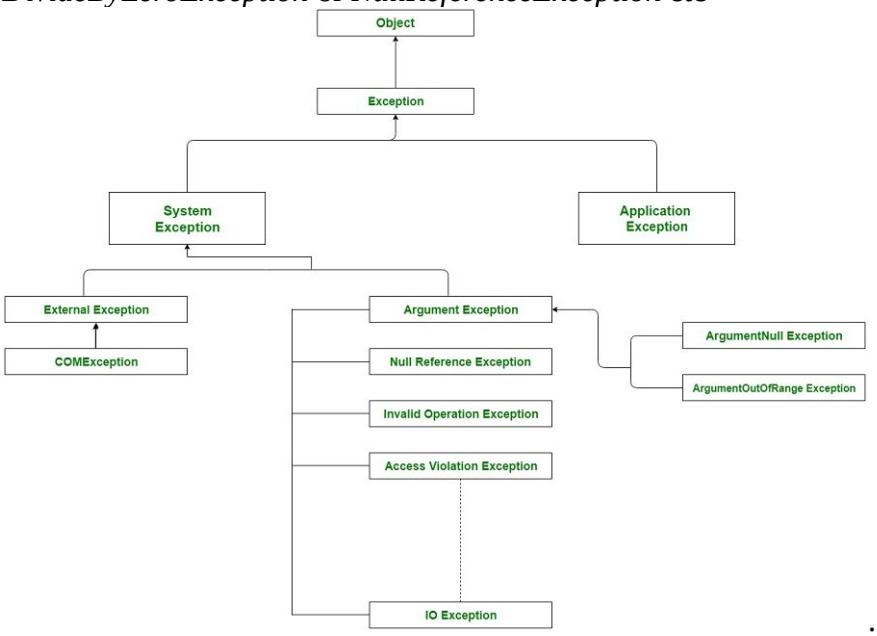
- While the TargetSite and StackTrace properties allow programmers to gain an understanding
- of a given exception, this information is of little use to the end user. As you have already seen, the System.Exception.Message property can be used to obtain human-readable information that can be displayed to the current user. In addition, the HelpLink property can be set to point the user to a specific URL or standard Windows help file that contains more

detailed information. By default, the value managed by the HelpLink property is an empty string.

- **StackTrace:** This property helps to provide the information about where the error occurred.
- The System.Exception.StackTrace property allows you to identify the series of calls that
- resulted in the exception. Be aware that you never set the value of StackTrace, as it is established automatically at the time the exception is created.

## Exception Hierarchy

In C#, all the exceptions are derived from the base class **Exception** which gets further divided into two branches as **ApplicationException** and another one is **SystemException**. *SystemException* is a base class for all CLR or program code generated errors. *ApplicationException* is a base class for all application related exceptions. All the exception classes are directly or indirectly derived from the Exception class. In case of *ApplicationException*, the user may create its own exception types and classes. But SystemException contains all the known exception types such as *DivideByZeroException* or *NullReferenceException* etc




**Different Exception Classes:** There are different kinds of exceptions which can be generated in C# program:

- **Divide By Zero exception**: It occurs when the user attempts to divide by zero
- **Out of Memory exceptions**: It occurs when then the program tries to use excessive memory
- **Index out of bound Exception**: Accessing the array element or index which is not present in it.
- **Stackoverflow Exception**: Mainly caused due to infinite recursion process
    - **Null Reference Exception** : Occurs when the user attempts to reference an object which is of NULL type.
    -

# Exceptions

Exceptions in the application must be handled to prevent crashing of the program and unexpected result, log exceptions and continue with other functionalities. C# provides built-in support to handle the exception using `try`, `catch` & `finally` blocks.

Syntax:

```
try
{
   // put the code here that may raise exceptions
}
catch
{
   // handle exception here
}
finally
{
   // final cleanup code
}
```

**try block:** Any suspected code that may raise exceptions should be put inside a `try{ }` block. During the execution, if an exception occurs, the flow of the control jumps to the first matching `catch` block.

**catch block:** The `catch` block is an exception handler block where you can perform some action such as logging and auditing an exception. The `catch` block takes a parameter of an exception type using which you can get the details of an exception.

**finally block:** The `finally` block will always be executed whether an exception raised or not. Usually, a `finally` block should be used to release resources, e.g., to close any stream or file objects that were opened in the `try` block.

The following may throw an exception if you enter a non-numeric character.

## Example: C# Program

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Enter a number: ");

    var num = int.Parse(Console.ReadLine());

    Console.WriteLine($"Squre of {num} is {num * num}");
  }
}
```

To handle the possible exceptions in the above example, wrap the code inside a `try` block and handle the exception in the `catch` block, as shown below.

## Example: Exception handling using try-catch blocks

```
class Program
{
  static void Main(string[] args)
  {
    try
    {
      Console.WriteLine("Enter a number: ");

      var num = int.parse(Console.ReadLine());

      Console.WriteLine($"Squre of {num} is {num * num}");
    }
    catch
    {
      Console.Write("Error occurred.");
    }
    finally
    {
      Console.Write("Re-try with a different number.");
    }
  }
}
```

---

# Throwing a Generic Exception

The current implementation of Accelerate() displays an error message if the caller attempts to speed up the Car beyond its upper limit. To retrofit this method to throw an exception if the user attempts to speed up the automobile after it has met its maker, you want to create and configure a new instance of the System.Exception class, setting the value of the read-only Message property via the class constructor. When you wish to send the error object back to the caller, make use of the C# throw keyword. Here is the relevant code update to the Accelerate() method:

```
public void Accelerate(int delta)
{
if (carIsDead)
        Console.WriteLine("{0} is out of order...", petName);
else
        {
        currSpeed += delta;
        if (currSpeed >= maxSpeed)
                {
                carIsDead = true;
                currSpeed = 0;
                // Use "throw" keyword to raise an exception.
                throw new Exception(string.Format("{0} has overheated!", petName));
                }
        else
        Console.WriteLine("=> CurrSpeed = {0}", currSpeed);
        }
}
```

First of all, when you are throwing an exception, it is always up to you to decide exactly what constitutes the error in question, and when it should be thrown. Here, you are making the assumption that if the program attempts to increase the speed of a car that has expired, a System.Exception type should be thrown to indicate the Accelerate() method cannot continue. Alternatively, you could implement Accelerate() to recover automatically without needing to throw an exception in the first place. By and large, exceptions should be thrown only when a more terminal condition has been met Deciding exactly what constitutes throwing an exception is a design issue you must always contend with.

# Catching Exceptions

In .NET from a Java background, understand that type members are not prototyped with the set of exceptions they may throw (in other words, .NET does not support checked exceptions). its not required to handle every exception thrown from a given member.

Because the Accelerate() method now throws an exception, the caller needs to be ready to handle the exception, should it occur. When you are invoking a method that may throw an exception, you make use of a try/catch block. After you have caught the exception object, you are able to invoke the members of the exception object to extract the details of the problem. What you do with this data is largely up to you. You might want to log this information to a report file, write the data to the Windows event log, e-mail a system administrator, or display the problem to the end user. Here, as to dump the contents to the console window:

---

```
// Handle the thrown exception.
static void Main(string[] args)
{
                Console.WriteLine("***** Simple Exception Example *****");
                Console.WriteLine("=> Creating a car and stepping on it!");
                Car myCar = new Car("Zippy", 20);
                myCar.CrankTunes(true);
                // Speed up past the car's max speed to
                // trigger the exception.
try
{
        for(int i = 0; i < 10; i++)
        myCar. Accelerate(10);
}
catch(Exception e)
{
        Console.WriteLine("\n*** Error! ***");
        Console.WriteLine("Method: {0}", e.TargetSite);Console.WriteLine("Message: {0}", e
Message);
        Console.WriteLine("Source: {0}", e.Source);
}
        // The error has been handled, processing continues with the next
        statement.
        Console.WriteLine("\n***** Out of exception logic *****");
        Console.ReadLine();
}
```

---

As you can see, after an exception has been handled, the application is free to continue on from the point after the catch block. In some circumstances, a given exception could be critical enough to warrant the termination of the application. However, in a good number of cases, the logic within the exception handler will ensure the application can continue on its merry way (although it could be slightly less functional, such as not being able to connect to a remote data source).

# Finally Block

A try/catch scope may also define an optional finally block. The purpose of a finally block is to ensure that a set of code statements will always execute, exception (of any type) or not. To illustrate, assume you want to always power down the car's radio before exiting Main(), regardless of any handled exception. The `finally` statement lets you execute code, after `try...catch`, regardless of the result

---

```
try{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);}catch (Exception e){
  Console.WriteLine("Something went wrong.");
}
catch(Exception e)
{}
finally{
  Console.WriteLine("The 'try catch' is finished.");
}
```

## Interfaces vs. Abstract Classes

The interface type seems very similar to an abstract base class. When a class is market as abstract, it **may** define any number of abstract members to provide a polymorphic interface to all derived types. But even when a class type does define a set of abstract members, it may define any number of constructors, field data, nonabstract members, and so on. In contrast, interfaces **only** contain abstract members.

The polymorphic interface established by an abstract parent class suffers from one major limitation in that **only derived types** support the members defined by the abstract parent. But in larger

software, it is very common to develop multiple class hierarchies that have no common parent beyond **System.Object**. Because abstract members in an abstract base class only apply to derived types, we have no way to configure types in different hierarchies to support same polymorphic interface.

## C# - Interface

In the human world, a contract between the two or more humans binds them to act as per the contract. In the same way, an interface includes the declarations of related functionalities. The entities that implement the interface must provide the implementation of declared functionalities.

In C#, an interface can be defined using the `interface` keyword. An interface can contain declarations of methods, properties, indexers, and events. However, it cannot contain fields, auto-implemented properties.

The following interface declares some basic functionalities for the file operations.

Example: C# Interface

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}
```

You cannot apply access modifiers to interface members. All the members are public by default. If you use an access modifier in an interface, then the C# compiler will give a compile-time error "The modifier 'public/private/protected' is not valid for this item.". (Visual Studio will show an error immediately without compilation.)

Example: Invalid Interface with Access Modifiers

```
interface IFile
{
   protected void ReadFile(); //compile-time error
   private void WriteFile(string text);//compile-time error
}
```

An interface can only contain declarations but not implementations. The following will give a compile-time error.

Example: Invalid Interface with Implementation

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text){
      Console.Write(text);  //error: cannot implement method
   }
}
```

# Implementing an Interface

A class or a Struct can implement one or more interfaces using colon (:).

Syntax: `<Class or Struct Name> : <Interface Name>`

For example, the following class implements the `IFile` interface implicitly.

Example: Interface Implementation

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}

class FileInfo : IFile
{
   public void ReadFile()
   {
      Console.WriteLine("Reading File");
   }

   public void WriteFile(string text)
   {
      Console.WriteLine("Writing to file");
   }
}
```

In the above example, the `FileInfo` class implements the `IFile` interface. It defines all the members of the `IFile` interface with public access modifier. The `FileInfo` class can also contain members other than interface members.

Note:
Interface members must be implemented with the `public` modifier; otherwise, the compiler will give compile-time errors. You can create an object of the class and assign it to a variable of an interface type, as shown below.

Example: Interface Implementation

```
public class Program
{
   public static void Main()
   {
      IFile file1 = new FileInfo();
      FileInfo file2 = new FileInfo();

      file1.ReadFile();
      file1.WriteFile("content");

      file2.ReadFile();
      file2.WriteFile("content");
```

```
    }
}
```

Above, we created objects of the `FileInfo` class and assign it to `IFile` type variable and `FileInfo` type variable. When interface implemented implicitly, you can access `IFile` members with the `IFile` type variables as well as `FileInfo` type variable.

# Explicit Implementation

An interface can be implemented explicitly using `<InterfaceName>.<MemberName>`. Explicit implementation is useful when class is implementing multiple interfaces; thereby, it is more readable and eliminates the confusion. It is also useful if interfaces have the same method name coincidently.

Note: Do not use *public* modifier with an explicit implementation. It will give a compile-time error.
Example: Explicit Implementation

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}

class FileInfo : IFile
{
  void IFile.ReadFile()
   {
      Console.WriteLine("Reading File");
   }

   void IFile.WriteFile(string text)
   {
      Console.WriteLine("Writing to file");
   }}
```

When you implement an interface explicitly, you can access interface members only through the instance of an interface type.

Example: Explicit Implementation

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}

class FileInfo : IFile
{
   void IFile.ReadFile()
   {
      Console.WriteLine("Reading File");
   }
```

```
    void IFile.WriteFile(string text)
    {
        Console.WriteLine("Writing to file");
    }

    public void Search(string text)
    {
        Console.WriteLine("Searching in file");
    }
}

public class Program
{
    public static void Main()
    {
        IFile file1 = new FileInfo();
        FileInfo file2 = new FileInfo();

        file1.ReadFile();
        file1.WriteFile("content");
        //file1.Search("text to be searched")//compile-time error

        file2.Search("text to be searched");
        //file2.ReadFile(); //compile-time error
        //file2.WriteFile("content"); //compile-time error
    }
}
```

In the above example, `file1` object can only access members of `IFile`, and `file2` can only access members of `FileInfo` class. This is the limitation of explicit implementation.

# Implementing Multiple Interfaces

A class or struct can implement multiple interfaces. It must provide the implementation of all the members of all interfaces.

Example: Implement Multiple Interfaces

```
interface IFile
{
    void ReadFile();
}

interface IBinaryFile
{
```

```
    void OpenBinaryFile();
    void ReadFile();
}

class FileInfo : IFile, IBinaryFile
{
    void IFile.ReadFile()
    {
        Console.WriteLine("Reading Text File");
    }

    void IBinaryFile.OpenBinaryFile()
    {
        Console.WriteLine("Opening Binary File");
    }

    void IBinaryFile.ReadFile()
    {
        Console.WriteLine("Reading Binary File");
    }

    public void Search(string text)
    {
        Console.WriteLine("Searching in File");
    }
}

public class Program
{
    public static void Main()
    {
        IFile file1 = new FileInfo();
        IBinaryFile file2 = new FileInfo();
        FileInfo file3 = new FileInfo();

        file1.ReadFile();
        //file1.OpenBinaryFile(); //compile-time error
        //file1.SearchFile("text to be searched"); //compile-time error

        file2.OpenBinaryFile();
        file2.ReadFile();
        //file2.SearchFile("text to be searched"); //compile-time error

        file3.Search("text to be searched");
        //file3.ReadFile(); //compile-time error
        //file3.OpenBinaryFile(); //compile-time error
    }
}
```

Above, the `FileInfo` implements two interfaces `IFile` and `IBinaryFile` explicitly. It is recommended to implement interfaces explicitly when implementing multiple interfaces to avoid confusion and more readability.

Points to Remember :

1. Interface can contain declarations of method, properties, indexers, and events.
2. Interface cannot include private, protected, or internal members. All the members are public by default.

3. Interface cannot contain fields, and auto-implemented properties.
4. A class or a struct can implement one or more interfaces implicitly or explicitly. Use public modifier when implementing interface implicitly, whereas don't use it in case of explicit implementation.
5. Implement interface explicitly using `InterfaceName.MemberName`.
6. An interface can inherit one or more interfaces.

---

## Invoking Interface Members at the Object Level

The most straightforward way to interact with functionality supplied by a given interface is to invoke the members directly from the object level

For example, consider the following Main() method:

Explicit

```
class FileInfo : IFile
{
 void IFile.ReadFile()
  {
    Console.WriteLine("Reading File");
  }

}
```

invoking :

```
            IFile file1 = new FileInfo();
    FileInfo file2 = new FileInfo();

    file1.ReadFile();

    file2.Search("text to be searched");
    //file2.ReadFile(); //compile-time error
```

implicit:

```
class FileInfo : IFile
{
 public void ReadFile()
  {
    Console.WriteLine("Reading File");
  }

}
```

invoking:

```
            IFile file1 = new FileInfo();
    FileInfo file2 = new FileInfo();

    file1.ReadFile();

    file2.ReadFile();
```

| | |
|---|---|

One way to determine at runtime whether a type supports a specific interface is to use an explicit cast. If the type does not support the requested interface, you receive an InvalidCastException. To handle this possibility gracefully, use structured exception handling as in the following

example:

```
static void Main(string[] args)
{
...
// Catch a possible InvalidCastException.
Circle c = new Circle("Lisa");
IPointy itfPt = null;
try
{
itfPt = (IPointy)c;
Console.WriteLine(itfPt.Points);
}
catch (InvalidCastException e)
{
Console.WriteLine(e.Message);
}Console.ReadLine();
}
```

While you could use try/catch logic and hope for the best, it would be ideal to determine which interfaces are supported before invoking the interface members in the first place. Let’s see two ways of doing so.

**Obtaining Interface References: The as Keyword**

determine whether a given type supports an interface by using the as keyword. If the object can be treated as the specified interface, it will returned a reference to the interface in question. If not, receive a null reference. Therefore, be sure to check against a null value before proceeding.

```
static void Main(string[] args)
{
// Can we treat hex2 as IPointy?
Hexagon hex2 = new Hexagon("Peter");
IPointy itfPt2 = hex2 as Ipointy;

if(itfPt2 != null)
Console.WriteLine("Points: {0}", itfPt2.Points);
else
Console.WriteLine("OOPS! Not pointy...");
Console.ReadLine();
}
```

## Obtaining Interface References:The is Keyword

check for an implemented interface using the is keyword . If the object in question is not compatible with the specified interface, you are returned the value false. On the other hand, if the type is compatible with the interface in question, you can safely call the members without needing to use try/catch logic.

To illustrate, assume you have an array of Shape types containing some members that implement IPointy. Notice how you are able to determine which items in the array support this interface using the is keyword, as shown in this retrofitted Main() method:

```
static void Main(string[] args)
{
Console.WriteLine("***** Fun with Interfaces *****\n");// Make an array of Shapes.
Shape[] myShapes = { new Hexagon(), new Circle(),
new Triangle("Joe"), new Circle("JoJo")} ;
for(int i = 0; i < myShapes.Length; i++)
{
// Recall the Shape base class defines an abstract Draw()
// member, so all shapes know how to draw themselves.
myShapes[i].Draw();
// Who's pointy?
if(myShapes[i] is IPointy)
Console.WriteLine("-> Points: {0}", ((IPointy)myShapes[i]).Points);
else
Console.WriteLine("-> {0}\'s not pointy!",myShapes[i].PetName);
Console.WriteLine();
}
Console.ReadLine();
}
```

The output is as follows:
***** Fun with Interfaces *****
Drawing NoName the Hexagon
-> Points: 6
Drawing NoName the Circle
-> NoName's not pointy!
Drawing Joe the Triangle
-> Points: 3
Drawing JoJo the Circle
-> JoJo's not pointy!

---

## Interfaces As Parameters

Given that interfaces are valid .NET types, it is possible construct methods that take interfaces as parameters, as illustrated by the CloneMe() method earlier in this chapter. For the current example, assume you have defined another interface named Idraw3D.

```
// Models the ability to render a type in stunning 3D.public interface IDraw3D
{
void Draw3D();
}
```

Next, assume that two of your three shapes (ThreeDCircle and Hexagon) have been configured to support this new behavior.

```
// Circle supports IDraw3D.
class ThreeDCircle : Circle, IDraw3D
{
...
public void Draw3D()
{ Console.WriteLine("Drawing Circle in 3D!"); }
}
// Hexagon supports IPointy and IDraw3D.


class Hexagon : Shape, IPointy, IDraw3D
{
...
public void Draw3D()
{ Console.WriteLine("Drawing Hexagon in 3D!"); }
}
```

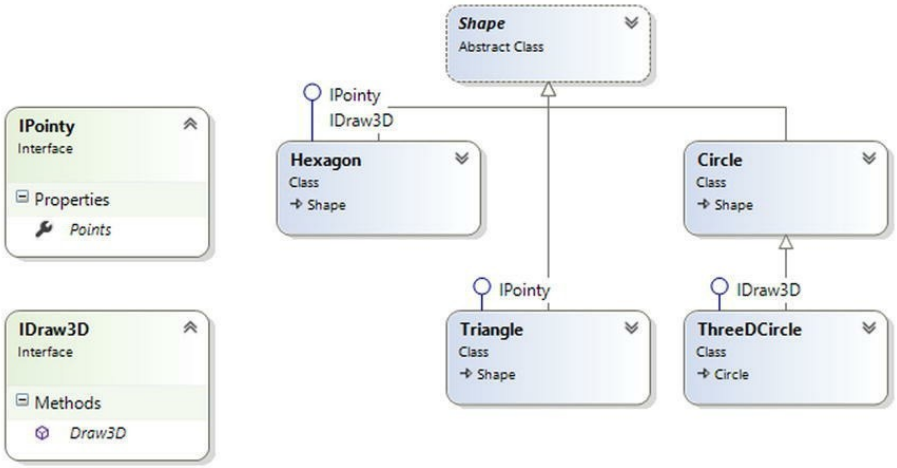



Figure -presents the updated Visual Studio class diagram.

In Figure The updated shapes hierarchyIf you now define a method taking an IDraw3D interface as a parameter, you can effectively send in any object implementing IDraw3D.

Consider the following method defined within Program class:

```
// I'll draw anyone supporting IDraw3D.
static void DrawIn3D(IDraw3D itf3d)
{
        Console.WriteLine("-> Drawing IDraw3D compatible type");
        itf3d.Draw3D();
}

static void Main(string[] args)
{
Console.WriteLine("***** Fun with Interfaces *****\n");
Shape[] myShapes = { new Hexagon(), new Circle(),new Triangle("Joe"), new Circle("JoJo") } ;
for(int i = 0; i < myShapes.Length; i++)
{
...
// Can I draw you in 3D?
        if(myShapes[i] is IDraw3D)
        DrawIn3D((IDraw3D)myShapes[i]);
}
}
```

Here is the output of the updated application. Notice that only the Hexagon object prints out in 3D, as the other members of the Shape array do not implement the IDraw3D interface.

***** Fun with Interfaces *****

```
Drawing NoName the Hexagon
-> Points: 6
-> Drawing IDraw3D compatible type
Drawing Hexagon in 3D!
Drawing NoName the Circle
-> NoName's not pointy!
Drawing Joe the Triangle
-> Points: 3
Drawing JoJo the Circle
-> JoJo's not pointy!
```

---

## Interfaces As Return Values

Interfaces can also be used as method return values. For example, you could write a method that takes an array of Shape objects and returns a reference to the first item that supports Ipointy.

```
// This method returns the first object in the
// array that implements IPointy.
static IPointy FindFirstPointyShape(Shape[] shapes)
{
        foreach (Shape s in shapes)
        {
                if (s is IPointy)
                return s as IPointy;
        }
                return null;
}
```

You could interact with this method as follows:

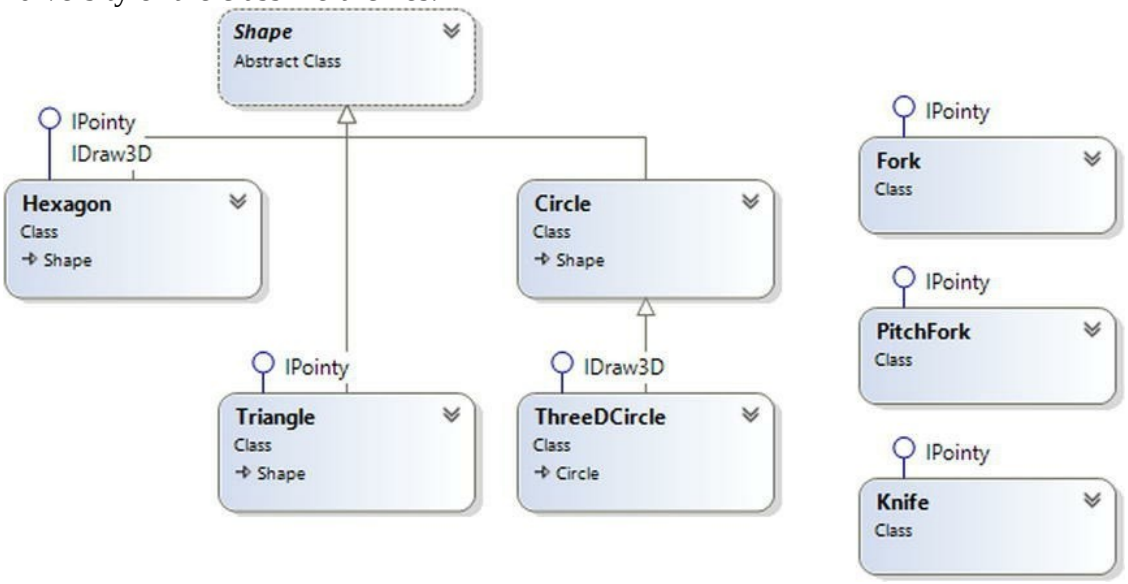
```
static void Main(string[] args)
{

Console.WriteLine("***** Fun with Interfaces *****\n");
// Make an array of Shapes.
Shape[] myShapes = { new Hexagon(), new Circle(),
new Triangle("Joe"), new Circle("JoJo")};
// Get first pointy item.
// To be safe, you'd want to check firstPointyItem for null before proceeding.
1IPointy firstPointyItem = FindFirstPointyShape(myShapes);
1Console.WriteLine("The item has {0} points", firstPointyItem.Points);
...
}
```

---

## Arrays of Interface Types

same interface can be implemented by numerous types, even if they are not within the same class hierarchy and do not have a common parent class beyond System.Object. This can yield some powerful programming constructs. For example, assume you have developed three new class types within your current project that model kitchen utensils (via Knife and Fork classes) and another modeling gardening equipment (à la PitchFork).

Consider below Figure Recall that interfaces can be “plugged into” any type in any part of a class hierarchy If you defined the PitchFork, Fork, and Knife types, you could now define an array of IPointy-compatible objects. Given that these members all support the same interface, you can iterate through the array and treat each item as an IPointy-compatible object, regardless of the overall diversity of the class hierarchies.




```
static void Main(string[] args)
{
...
// This array can only contain types that
// implement the IPointy interface.
IPointy[] myPointyObjects = {new Hexagon(), new Knife(), new Triangle(), new Fork(), new Pitch
Fork()};
foreach(IPointy i in myPointyObjects)
        Console.WriteLine("Object has {0} points.", i.Points);
        Console.ReadLine();
}
```

Just to highlight the importance of this example, remember this: when you have an array of a given interface, the array can contain any class or structure that implements that interface.

---

## polymorphic interface

The polymorphic interface established by an abstract parent class suffers from one major limitation in that **only derived types** support the members defined by the abstract parent. But in larger

software, it is very common to develop multiple class hierarchies that have no common parent beyond **System.Object**. Because abstract members in an abstract base class only apply to derived types, we have no way to configure types in different hierarchies to support same polymorphic interface.

Explicit

```
class FileInfo : IFile
{
 void IFile.ReadFile()
  {
    Console.WriteLine("Reading File");
  }

}
```

invoking :

```
IFile file1 = new FileInfo();
FileInfo file2 = new FileInfo();

file1.ReadFile();

file2.Search("text to be searched");
//file2.ReadFile(); //compile-time error
```

implicit:

```
class FileInfo : IFile
{
 public void ReadFile()
  {
    Console.WriteLine("Reading File");
  }

}
```

invoking:

```
IFile file1 = new FileInfo();
FileInfo file2 = new FileInfo();

file1.ReadFile();

file2.ReadFile();
```

```
class FileInfo : IFile
{
 public void ReadFile()
  {
    Console.WriteLine("Reading File");
  }

}
```

```
class PHOTO : IFile
{
 public void  ReadFile()
   {
      Console.WriteLine("Reading File");
   }

}
```

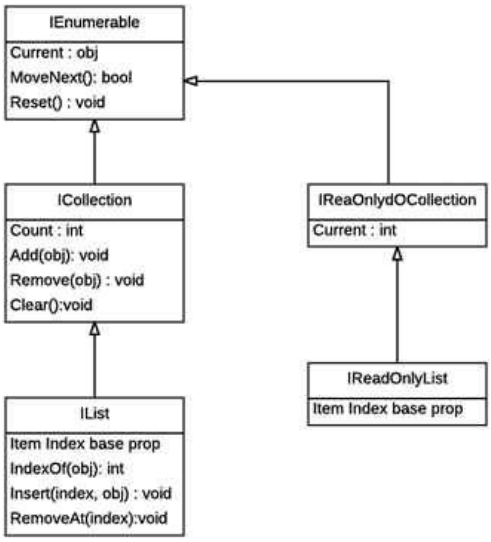
invoking:

```
        IFile  file1  =  new  FileInfo();
        IFile  filep  =  new  PHOTO();

      FileInfo file2 = new FileInfo();

      file1.ReadFile();

      file2.ReadFile();
```

## Collection Interfaces

A collection is a set of related objects. Unlike arrays, a collection can grow and shrink dynamically as the number of objects added or deleted. A collection is a class, so you must declare a new collection before you can add elements to that collection.

All of the collection types use some common interfaces. These common interfaces define the basic functionality for each collection class. The key collections interfaces are – IEnumerable, ICollection, IDictionary and IList.



IEnumerable acts as a base interface for all the collection types that is extended by ICollection. ICollection is further extended by IDictionary and IList.

**IEnumerab**le :Provides an enumerator which supports a simple iteration over a non-generic collection.
**ICollection** :Defines size, enumerators and synchronization methods for all nongeneric collections.
**IDictionary** :Represents a nongeneric collection of key/value pairs.
**IList** :Represents a non-generic collection of objects that can be individually accessed by index.
All collections interfaces are not implemented by all the collections. It depends on collection nature.

For example, IDictionary interface would be implemented by only those collection classes which support key/value pairs, like HasTable and SortedList etc.

# Features of Collections

Though all the collections have the ability to add, remove or find items in a collection they also have some additional features as mentioned below,

1. Ability to enumerate collections.
2. Ability to copy collection contents to an array.
3. Capacity and Count properties.
4. Consistent lower bound.
5. Synchronization for access from multiple threads.

# Interfaces

Interfaces

| | |
|---|---|
| IAsyncEnumerable<T> | Exposes an enumerator that provides asynchronous iteration over values of a specified type. |
| IAsyncEnumerator<T> | Supports a simple asynchronous iteration over a generic collection. |
| ICollection<T> | Defines methods to manipulate generic collections. |
| IComparer<T> | Defines a method that a type implements to compare two objects. |

| | |
|---|---|
| IDictionary<TKey,TValue> | Represents a generic collection of key/value pairs. |
| IEnumerable<T> | Exposes the enumerator, which supports a simple iteration over a collection of a specified type. |
| IEnumerator<T> | Supports a simple iteration over a generic collection. |
| IEqualityComparer<T> | Defines methods to support the comparison of objects for equality. |
| IList<T> | Represents a collection of objects that can be individually accessed by index. |
| IReadOnlyCollection<T> | Represents a strongly-typed, read-only collection of elements. |
| IReadOnlyDictionary<TKey,TValue> | Represents a generic read-only collection of key/value pairs. |
| IReadOnlyList<T> | Represents a read-only collection of elements that can be accessed by index. |
| IReadOnlySet<T> | Provides a readonly abstraction of a set. |
| ISet<T> | Provides the base interface for the abstraction of sets. |