Exception, interfaces and collections (C# 7,8,9)

Exceptions

An application can have:

- bugs
  - Bad programming job
    - memory leaks, null pointer
- Errors
  - End user
    - making errors
- Exceptions
  - runtime errors
    - File is not found
    - database not available

- .NET has built-in exceptions
  - IndexOutOfRangeException
  - FileNotFoundException
  - and a lot more...

- In C programming numerical constants and return values from functions

- Exception handling in .NET works across languages and assemblies

.NET exception handling has four elements

- A type representing an exception (an Exception class)
- A method that throws an exception (throw keyword)
- A block of code which invoke an exception (try section)
- A block of code which handles the exception (catch section)

- Plus finally

In another word:

- try
- catch
- throw
- finally (java like)

System.Exception base class

***Table 7-1**. Core Members of the System.Exception Type*

System.Exception Property	Meaning in Life
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 detail.
InnerException	This read-only property can be used to obtain information about the previous exception(s) that 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 very useful during debugging or if you wish 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).

Throwing an exception

```
using System;

public class Excp
{
    public void Kast() {

      throw new Exception("Exception");
    }


}


public class User
{

  public static void Main()

  {

        Excp f = new Excp();
        f.Kast();   // OH NO...

        Console.ReadLine();
  }
}
```

- **throw** must ALWAYS be caught
  - A throw propagates up to Main and close the application down
  - Use (always) try catch

```
using System;

public class Excp
{
    public void Kast() {

      throw new Exception("Exception");
    }


}
```

```
public class User
{


  public static void Main()

  {
        try{
          Excp f = new Excp();
          f.Kast();

        }catch(Exception e)
        {
          Console.WriteLine("Message: {0}", e.Message);
          Console.WriteLine("Method: {0}",e.TargetSite);


         }

        Console.ReadLine();
  }



}
```

- The application will survive!
- Exceptions should only be thrown when a terminal condition happen!

StackTrace and TargetSite

- StackTrace shows the sequence of call before the exception
- TargetSite shows type, name and parameter type of the method causing the exception

```
...
public static void Main()

  {
```

```
        try{
         Excp f = new Excp();
         f.Kast();
        }
          catch(Exception e)
        {
          Console.WriteLine("Meddelse: {0}",e.Message);
          Console.WriteLine("Metode: {0}",e.TargetSite);
          Console.WriteLine("Trace: {0}",e.StackTrace);
         }

        Console.ReadLine();
  }
...
```

Output:

Trace: at Excp.Kast()
  at User.Main()

HelpLink – Help to the user on run-time

- A guide to the user
- example:
    - Help: https://www.gedogko.com

```
using System;

public class Excp
{
    public void Kast() {

      Exception ex =  new Exception("En Exception");
      ex.HelpLink = "https://www.gedogko.com";
      throw ex;
    }
```

```
}

public class User
{


  public static void Main()

  {
        try
        {
         Excp f = new Excp();
         f.Kast();
        }catch(Exception e)
        {
          Console.WriteLine("Meddelse: {0}",e.Message);
          Console.WriteLine("Metode: {0}",e.TargetSite);
          Console.WriteLine("Trace: {0}",e.StackTrace);
          Console.WriteLine("Help: {0}",e.HelpLink);
         }

        Console.ReadLine();
  }

}
```

e.TargetSite.DeclaringType shows which class that caused an exception
e.TargetSite.MemberType shows the type that throws an exception for example a method
e.TargetSite alone gives the methods name

- Data Property
  - Can for instance contain a TimeStamp
  - Data property returns an object implementing the IDictionary interface

More about .NET and exceptions

- System
  - IndexOutOfRangeException
  - StackOverFlowException
  - And many more
- More specific namespaces have their own exceptions

  - System.IO
  - And many more

- If an exception is thrown in the built-in classes' methods
  - A *System exception* is thrown
  - System.Exception inherit from Exception
  - Exceptions are thrown by CLR

- Use the **is** keyword to test if an exception is a SystemException

System.Exceptions – how to know?

- How to know which exceptions are thrown fra base library classes?
  - Check online help

- As in C++
  - AND DIFFERENT FROMJava
    - You don't have to use try-catch

Example: **not working in java (without handling):**

```
using System;
using System.IO;

public class IOLeg
{
  public static void Main()

  {

         StreamReader sr = File.OpenText("l4_2.cs");
         string input = null;
         while((input =sr.ReadLine()) != null)
                  Console.WriteLine(input);
         sr.Close();
         Console.ReadLine();
  }

}
```

Building own exceptions

Example:

```
using System;
using System.IO;


public class AvException :System.Exception

{
   private string msg;
   private string className;

   public AvException(string className)
   {
        this.className = className;
   }

   public override string Message
   {
     get{
         msg = base.Message;
         msg += "Class Name: "+className;

         return msg;

     }
   }


}


public class Excp
{
    public void Kast() {

      throw new AvException(this.ToString());
    }


}



public class EgenExcp
{
```

```
  public static void Main()

  {
        try{
            Excp f  = new Excp();
            f.Kast();

        }catch(Exception e)
        {

          Console.WriteLine("Meddelse: {0}",e.Message);
        }

         Console.ReadLine();

  }

}
```

output:

Meddelse: Exception of type AvException was thrown.Class Name: Excp

- Notice inheritance from System.Exception
- override string Message

Better – Use ApplicationException

Note In practice, few .NET developers build custom exceptions that extend ApplicationException. Rather, it is more common to simply subclass System.Exception; however, either approach is technically valid.

Now inheriting ApplicationException (Application level exceptions):

```
...

public class AvException :ApplicationException

{
...
```

It is possible to pass information to the Message property. Just pass the incoming message to the parent's constructor

…

public AvException(string message, string cause, DateTime time): base(message)
{
}
…

Serialization

It is possible to serialize exceptions objects

More about applications and System exception – multiple exceptions

- Possible to have more catch sections
  - Here is one at application level
  - And one on System level

Example:

```
using System;
using System.IO;


public class AvException :Exception

{
   private string msg;
   private string className;

   public AvException(string className)
   {
        this.className = className;
   }

   public override string Message
   {
     get{
         msg = base.Message;
         msg += "Class Name: "+className;

         return msg;
```

```
     }
   }


}


public class Excp
{
    public void Kast(int i) {

     if(i==0)
      throw new AvException(this.ToString());
     if(i==1)  // System level
      throw new
  ArgumentOutOfRangeException("Error caught in System");
    }


}

public class EgenExcp
{


  public static void Main()

  {
        try{
            Excp f  = new Excp();
            //f.Kast(0);
            f.Kast(1);

        }catch(AvException e)
        {
          Console.WriteLine("AvExcp: {0}",e.TargetSite);
          Console.WriteLine("AvExcp: {0}",e.Message);
        }
        catch(ArgumentOutOfRangeException a)
        {
         Console.WriteLine("ArgExcp: {0}",
                                a.TargetSite);
         Console.WriteLine("ArgExcp: {0}",a.Message);

        }

         Console.ReadLine();
  }
```

```
}
```

If: `f.Kast(0);`

Output:

AvExcp: Void Kast(Int32)
AvExcp: Exception of type AvException was thrown.Class Name: Excp

If: `f.Kast(1);`

Output:

ArgExcp: Void Kast(Int32)
ArgExcp: Specified argument was out of the range of valid values.
Parameter name: `Error caught in System`

The order of the catch sections

- Most specific catch should come first
- The last one is the most general
    - analog with C++: “catch all”

The generic catch block – “catch all” in C#

```
...

catch {

   Console.WriteLine("Something went wrong ????");
}
...
```

Rethrowing exceptions

- To rethrow an exception (rethrow the exception up the call stack to the previous caller)

```
using System;
using System.IO;


public class AvException :Exception

{
   private string msg;
   private string className;

   public AvException(string className)
   {
        this.className = className;
   }

   public override string Message
   {
     get{
         msg = base.Message;
         msg += "Class Name: "+className;

         return msg;

     }
   }


}


public class Excp
{
    public void Kast(int i) {

     if(i==0)
      throw new AvException(this.ToString());
     if(i==1)  // System level
      throw new ArgumentOutOfRangeException("Error..");
    }


}
```

```
public class EgenExcp
{


  public static void Main()

  {

 try{ // outer try
      try{ // inner try
            Excp f  = new Excp();

            f.Kast(0); // AvException

        }catch(AvException e)
        {
          // make a rethrow
          throw e;
        }
        catch(ArgumentOutOfRangeException a)
        {
          Console.WriteLine("ArgExcp: {0}",
                                a.TargetSite);
         Console.WriteLine("ArgExcp: {0}",a.Message);

        }

 }catch
 {
     Console.WriteLine("Got it...");
 }
     Console.ReadLine();


  }




}
```

Output:

```
Got it...
```

finally – will always be executed

```
...
  public static void Main()
  {

      try{
            Excp f  = new Excp();

            f.Kast(0);

        }catch(AvException e)
        {
          Console.WriteLine("AvExcp: {0}",e.TargetSite);
          Console.WriteLine("AvExcp: {0}",e.Message);
        }
        catch(ArgumentOutOfRangeException a)
        {
         Console.WriteLine("ArgExcp: {0}",a.TargetSite);
         Console.WriteLine("ArgExcp: {0}",a.Message);

        }

        finally
        {
          Console.WriteLine("close db connection..");
        }
        Console.ReadLine();


  }
...
```

Expression Filter (C# 6)

```
…
catch( CarIsDeadException e) when (e.ErrorTimeStamp.DayOfWeek != DayOfWeek.Friday)
{
   // This line will be executed if the when clause evaluates to true
  Console.WriteLine(“Catching car is dead!”);

   Console.WriteLine(e.Message);
}
```

…

Which method is throwing what?

- .NET exceptions should be handled
    - Can easily be identified

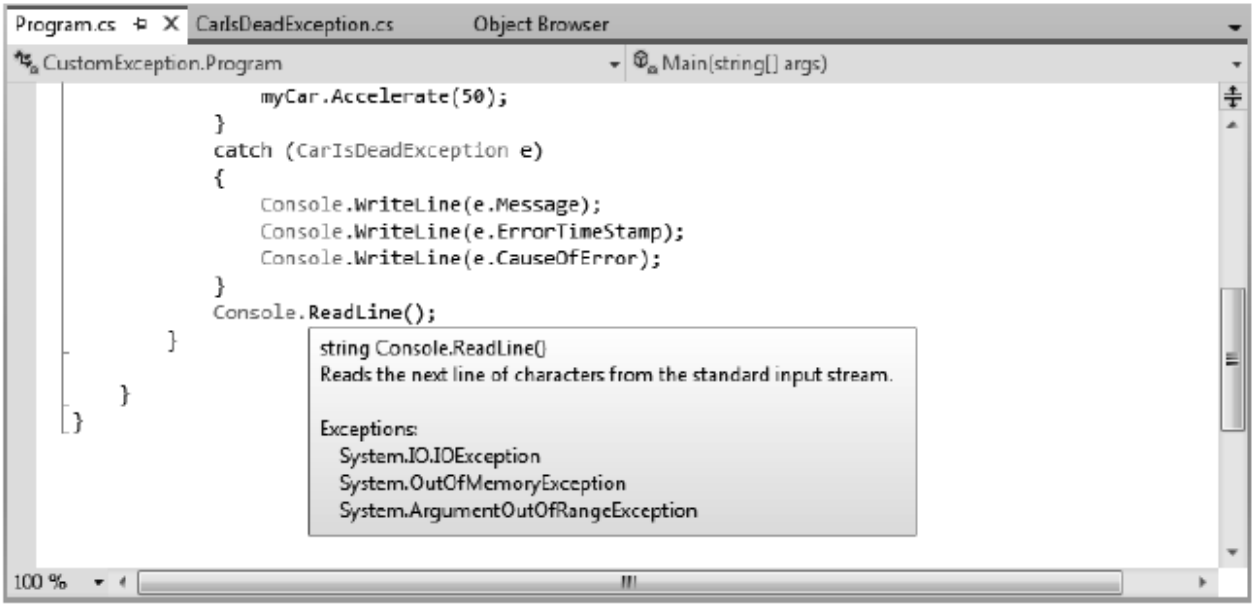


Figure 7-2. *Identifying the exceptions thrown from a given method*

Visual studio .Net and exceptions

- Possible to set Visual Studio to stop executing if an exception happen
    - without manual breakpoints

Lifetime of objects and garbage collection (short intro)

Rule:

Allocate an object with ”new” and forget the rest!

- CLR removes the object when it is not in use anymore
  - CLR keeps an eye with objects who are candidates for Garbage Collection
  - If the object has no reference

- When the object is created (**new**)
  - CLR calculates the need for memory
  - find ”has-a” and ”is-a” relations
  - Check the memory space
  - Place the object pointer to the next location in memory.
  - returns a reference

Garbage collection in C#

- If managed heap does not have enough memory the garbage collector is called

- Garbage Collector checks if the object is Ok to terminates
- Is thread based

To take over garbage collection: finalize

- Uses C++ syntax
- override System.Object.Finalize()

- Use GC.Collect() for **manual** garbage collection

Example:

```
Using System;

public class Mega
{
  private static int antal = 0;
  private static int objNo=0;

  public Mega()

  {
       antal++;
       objNo++;
       Console.WriteLine("obj no. {0} created", objNo);



  }
  public static void VisAntalObjekter()
  {


        Console.WriteLine("No of objects: {0}",antal);

  }


   ~Mega() // overrides Finalize()
   {
       // add extra work here..
       Console.WriteLine("objNo {0} destruktor",objNo);
       antal--;

   }

}


public class UserClass
{


  public static void Main()

  {
       Mega stor1 = new Mega();
       GC.Collect();
       Mega.VisAntalObjekter();
```

```
        Mega stor2 = new Mega();
        GC.Collect();
        Mega.VisAntalObjekter();
        Mega stor3 = new Mega();
        GC.Collect();
        Mega.VisAntalObjekter();
        Mega stor4 = new Mega();
        GC.Collect();
        Mega.VisAntalObjekter();
        Mega stor5 = new Mega();
        GC.Collect();
        Mega.VisAntalObjekter();

        Console.ReadLine();

   }


}
```

- Finalizing is time consuming
- **Only** do this if you have arguments for it

IDisposable Interface in .NET class Lib (Dispose method)

```
public interface IDisposable
{
      public void Dispose();
}
```

Dispose "belongs to" finally

```
...

 X x = new X();
...
// try / catch

finally
{
   x.Dispose();
```

```
}
...
```

Another way of doing this:

```
...
public void EnMetode()
{
      using ( X x = new X())
      {
         // Use the object
         //Dispose() is called automatically
         // when the block is leaved!

      }

}

...
```

Object generations – optimization of the GC

- Generations
  - Filosofy: Oldest objects stay longest (ex: object containing Main())
- three generations:
  - Generation 0
    - all new objects (Shortest life)
  - Generation 1
    - Objects marked for GC
  - Generation 2
    - Have survived several times

Interfaces

- interfaces have no base classes
- A class using an interface must implement the method
- Cannot have fields (variables)
- Cannot have constructors
- Cannot have implementation

Example:

```
using System;


public interface IPyth
{

   double Udregn(double k1, double k2);

}

public class UserClass:IPyth
{


  public static void Main()

  {



  }


}
```

Will give **compiler error:**

l4_7.cs(11,14): error CS0535: 'UserClass' does not implement interface member 'IPyth.Udregn(double, double)'

- A base class is placed before interfaces
- Many interfaces (comma separated) can be implemented
- All methods defined must be implemented

Why interfaces?

- A base class can contain abstract methods
  - interfaces are pure protocols
    - has no implementation itself
  - You will get polymorphy
    - (A class can implement in its own way)

example:

```
using System;


public interface IArea

{
    double UdregnArea(double g, double h);

}


public interface IPyth
{


   double UdregnHypo(double k1, double k2);

}
```

```
public class Triangle:IPyth , IArea // 2 interfaces
{

   public double UdregnHypo(double k1, double k2)
   {
     return ( Math.Sqrt((k1*k1)+(k2*k2)) );
   }

   public double UdregnArea(double h, double a)
   {

        return ((0.5*h) * a);

   }

}

public class UserClass
{


  public static void Main()

  {

    Triangle tri = new Triangle();

    double res = 0.0;
    double ka1 = 1.0;
    double ka2 = 1.0;
    res = tri.UdregnHypo(ka1,ka2);
    Console.WriteLine("res {0}",res);

    double h = 5.0;
    double a = 3.0;

    res = tri.UdregnArea(h,a);
    Console.WriteLine("res {0}",res);

    Console.ReadLine();
  }


}
```

Direct interface reference

- Can be used for test on runtime

Example:

```
...

    Triangle tri = new Triangle();

    IPyth ip = (IPyth) tri;

    double res = 0.0;
    double ka1 = 1.0;
    double ka2 = 1.0;
    res = ip.UdregnHypo(ka1,ka2);
    Console.WriteLine("res {0}",res);
...
```

An illegal cast can be caught on runtime:

```
...
public class CirkeLine //does not implement IArea

{
    public double UdregnArea(double r)
   {

        return (Math.PI*(r*r));

   }



}
...
// in Main

   CirkeLine cl = new CirkeLine();
   IArea ia;

    try{
```

```
      ia = (IArea) cl;


    } catch(InvalidCastException ice)
    {
         Console.WriteLine("Error {0}",ice.Message);
    }

...
```

Output:

Error Specified cast is not valid.

Test with as or is

Test with **as**:

```
...
   CirkeLine cl = new CirkeLine();
   IArea ia;

   ia = cl as IArea;

   if(ia == null)
               Console.WriteLine("Error!!!");
...
```

Or **is**:

```
...
         CirkeLine cl = new CirkeLine();
         if(cl is IArea)
                Console.WriteLine("hurra!!!");
         else
                Console.WriteLine("Error!!!");
...
```

C# 7: Updated is keyword

Safely call without try/catch
…

if(myShapes[i] is IPointy i**p**)
  Console.WriteLine(“ {0} ”, **ip.Points**);

Interfaces as method parameter

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceLeg1
{

    public interface ITegnBar
    {
        void Tegn();
    }

    public interface ISCalerBar
    {
        void Scaler();
    }

    public class Firkant : ITegnBar, ISCalerBar
    {
        public void Tegn()
        {
            Console.WriteLine("ITegnBar - Firkant");

        }

        public void Scaler()
        {
            Console.WriteLine("IScalerBar - Firkant");

        }

    }

    public class Streg : ITegnBar
    {
```

```
        public void Tegn()
        {
            Console.WriteLine("ITegnBar - Streg");

        }

    }


    class Program
    {
        static void Main(string[] args)
        {
            Firkant f = new Firkant();
            Streg s = new Streg();

            if (f is ISCalerBar)
                EnScalerBar(f);

            if (s is ISCalerBar)
                EnScalerBar(s);

        }


        public static void EnScalerBar(ISCalerBar icb)
        {
            Console.WriteLine("DENNE ER SCALERBAR");
            icb.Scaler();
        }
    }
}
```

Won't work Compiler-time Error

Error	1	The best overloaded method match for 'InterfaceLeg1.Program.EnScalerBar(InterfaceLeg1.ISCalerBar)' has some invalid arguments	d:\documents and settings\henrik\dokumenter\visual studio 2010\Projects\InterfaceLeg1\InterfaceLeg1\Program.cs	59	17	InterfaceLeg1

Interfaces as return values

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceLeg1
{

    public interface ITegnBar
    {
        void Tegn();
```

```
    }

    public interface ISCalerBar
    {
        void Scaler();
    }

    public class Firkant : Figur,ITegnBar, ISCalerBar
    {
        public void Tegn()
        {
            Console.WriteLine("ITegnBar - Firkant");

        }

        public void Scaler()
        {
            Console.WriteLine("IScalerBar - Firkant");

        }

    }

    public class Streg : Figur,ITegnBar
    {
        public void Tegn()
        {
            Console.WriteLine("ITegnBar - Streg");

        }

    }

    public class Figur
    {

    }

    class Program
    {
        static void Main(string[] args)
        {
            Figur[] figurer = new Figur[2];

            figurer[0] = new Firkant();
            figurer[1] = new Streg();

            ITegnBar itb = (ITegnBar) figurer[0];
            itb.Tegn();

            itb = (ITegnBar)figurer[1];
            itb.Tegn();


            // another way:

            ITegnBar itegn = HentInterfaceImpl(figurer[0]);
            itegn.Tegn();

            itegn = HentInterfaceImpl(figurer[1]);
```

```
            itegn.Tegn();




        }


        public static ITegnBar HentInterfaceImpl(Figur f)
        {
            Console.WriteLine("Udtrækker Interface del");
            if (f is ITegnBar)
                return f as ITegnBar;
            else
                return null;
        }



    }
}
```

Arrays of interfaces

- It is possible to extract the interface part direct
- See page 311

Name clash with interfaces

- Two interfaces can define the same method (name)
- Can be used directly by using direct reference

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace InterfaceLeg1
{

    public interface ITegnBar
    {
        void Tegn();

    }

    public interface ISCalerBar
    {
        void Scaler();
        void Tegn();
    }

    public class Firkant : ITegnBar, ISCalerBar
    {
        void ITegnBar.Tegn()
        {
            Console.WriteLine("Tegn: ITegnBar - Firkant");

        }

        void ISCalerBar.Tegn()
        {
            Console.WriteLine("Tegn: ISCalerBar - Firkant");

        }


        public void Scaler()
        {
            Console.WriteLine("IScalerBar - Firkant");

        }

    }

    public class Streg : ITegnBar
    {
        public void Tegn()
        {
            Console.WriteLine("ITegnBar - Streg");

        }

    }



    class Program
    {
        static void Main(string[] args)
        {


            Firkant f  = new Firkant();


            ITegnBar itb = (ITegnBar) f;
```

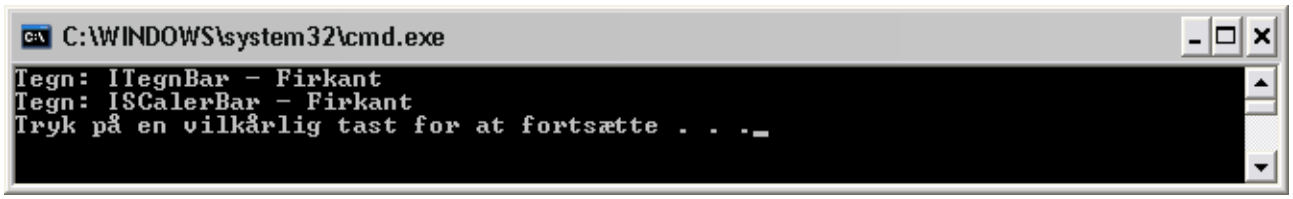
```
            itb.Tegn();

            ISCalerBar isc = (ISCalerBar)f;
            isc.Tegn();



        }


    }
}
```



”Inheritance”

- Interfaces can ”inherit” from several interfaces: ”**multiple inheritance”**

```
using System;


interface IA
{
   void UdskrivIA();
}

interface IB
{
   void UdskrivIB();
}

interface IC:IA,IB
{
   void UdskrivIC();
}
```

```
public class UseI:IC

{


 public void UdskrivIA()
 {
   Console.WriteLine("IA");

 }

 public void UdskrivIB()
 {
   Console.WriteLine("IB");
 }
 public void UdskrivIC()

 {

   Console.WriteLine("IC");

 }


}

public class UseMultipleInterfaces
{

  public static void Main()

  {
        UseI usei = new UseI();
        usei.UdskrivIA();
        usei.UdskrivIB();
        usei.UdskrivIC();
        Console.ReadLine();
  }


}
```

- Visual Studio can implement interfaces with Class View

“Multiple inheritance” and interfaces

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfacePlay
{
    public interface IA
    {
        void AM();
    }

    public interface IB:IA
    {
        void BM();
    }

    public interface IC : IA
    {
        void CM();
    }

    public interface ID : IB,IC
    {
        void DM();
    }

   public class Program:ID
    {
       public void AM()
        {
        }

        public void BM()
        {
        }

        public void CM()
        {
        }

        public void DM()
        {
        }


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

- Works perfect
    - No implemtation to worry about

IEnumerator

```
using System.Collections;
...
public class Garage : IEnumerable
{
// System.Array already implements IEnumerator!
private Car[] carArray = new Car[4];
public Garage()
{
carArray[0] = new Car("FeeFee", 200);
carArray[1] = new Car("Clunker", 90);
carArray[2] = new Car("Zippy", 30);
carArray[3] = new Car("Fred", 30);
}
public IEnumerator GetEnumerator()
{
// Return the array object's IEnumerator.
return carArray.GetEnumerator();
}
}
```

About the IEnumerable interface

- If we want to run through our own objectsin a foreach construction
  - The class must implement IEnumerable
  - Defining the method GetEnumerator
  - That returns an IEnumerator

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

Iterator methods using yield:

Building Iterator Methods with the yield Keyword

There's an alternative way to build types that work with the `foreach` loop via *iterators*. Simply put, an *iterator* is a member that specifies how a container's internal items should be returned when processed by `foreach`. To illustrate, create a new Console Application project named CustomEnumeratorWithYield and insert the `Car`, `Radio`, and `Garage` types from the previous example (again, renaming your namespace definitions to the current project if you like). Now, retrofit the current `Garage` type as follows:

```
public class Garage : IEnumerable
{
   private Car[] carArray = new Car[4];
   ...
   // Iterator method.

   public IEnumerator GetEnumerator()
   {
     foreach (Car c in carArray)
     {
       yield return c;
     }
   }
}
```

C# 7: local functions

Using a Local Function (New)

When the `GetEnumerator()` method is called, the code isn't executed until the value returned from the method is iterated. Update the method to the following code so that an exception is thrown on the first line:

```
public IEnumerator GetEnumerator()
{
  //This will not get thrown until MoveNext() is called
  throw new Exception("This won't get called");
  foreach (Car c in carArray)
  {
    yield return c;
  }
}
```

If you were to call the function like this and do *nothing else*, the exception will never be thrown:

```
IEnumerator carEnumerator = carLot.GetEnumerator();
```

It's not until `MoveNext()` is called that the code will execute and the exception is thrown. Depending on the needs of your program, that might be perfectly fine. But it might not. Recall from Chapter 4 the C# 7 local

The private function:

```
public IEnumerator GetEnumerator()
{
  //This will get thrown immediately
  throw new Exception("This will get called");

  return actualImplementation();

  //this is the private function
  IEnumerator actualImplementation()
  {
    foreach (Car c in carArray)
    {
      yield return c;
    }
  }
}
```

Cloning of objects

- Same reference to an object:

```
using System;

public class Point
{
  private int x,y;
  public Point()
  {}
  public Point( int x,int y)
  {
      this.x = x;
      this.y = y;

  }
  public void Change(int x, int y)
  {
       this.x = x;
       this.y = y;
  }
  public override string ToString()
  {
    return "x:"+x+"y:"+y;

  }

}
```

```
public class UsePoints
{


  public static void Main()

  {
        Point glP = new Point(2,3);
        Point nyP = glP;
        nyP.Change(8,3); // OH-NO!!
        Console.WriteLine("glP: {0}",glP.ToString());
        Console.WriteLine("nyP: {0}",nyP.ToString());
        Console.ReadLine();
  }




}
```

Output:

glP: x:8y:3
nyP: x:8y:3

Clone in the ICloneable interface

- Clone

```
using System;

public class Point:ICloneable
{
  private int x,y;
  public Point()
  {}
  public Point( int x,int y)
  {
      this.x = x;
      this.y = y;

  }
  public void Change(int x, int y)
  {
       this.x = x;
```

```
        this.y = y;
  }
  public override string ToString()
  {
    return "x:"+x+"y:"+y;

  }

  public object Clone()
  { // deep copy
    return new Point(this.x,this.y);
  }

}


public class UsePoints
{


  public static void Main()

  {
        Point glP = new Point(2,3);
        Point nyP = (Point)glP.Clone();
        nyP.Change(8,3);
        Console.WriteLine("glP: {0}",glP.ToString());
        Console.WriteLine("nyP: {0}",nyP.ToString());
        Console.ReadLine();
  }




}
```

Output:

glP: x:2y:3
nyP: x:8y:3

- Use MemberwiseClone if references are not used (shallow copy)
- Otherwise use deep copy

IComparable – To compare and sort objects

- Sorting own objects
  - IComparable Interface

Example:

```
using System;

public class Point:ICloneable,IComparable
{
  private int x,y;
  public Point()
  {}
  public Point( int x,int y)
  {
      this.x = x;
      this.y = y;

  }
  public int X
  {
     get{return x;}

  }
  public int Y
  {
     get{return y;}

  }


  public void Change(int x, int y)
  {
       this.x = x;
       this.y = y;
  }
  public override string ToString()
  {
    return "x:"+x+"y:"+y;

  }

  public object Clone()
```

```
  {
    return new Point(this.x,this.y);
  }
                  // deciedes < > ==
  int IComparable.CompareTo(object o)
  {
       Point tmp = (Point) o;
       double thisLen =
             Math.Sqrt((this.x*this.x)+(this.y*this.y));
       double tmpLen =
             Math.Sqrt((tmp.X*tmp.X)+(tmp.Y*tmp.Y));

       if(thisLen > tmpLen)
            return 1;

       if(thisLen < tmpLen)
            return -1;
       else
            return 0;
  }


}


public class UsePoints
{


  public static void Main()

  {
        Point glP = new Point(2,3);
        Point nyP = (Point)glP.Clone();
        nyP.Change(8,3);
        Console.WriteLine("glP: {0}",glP.ToString());
        Console.WriteLine("nyP: {0}",nyP.ToString());
        Point[] sortPoints = new Point[3];
        sortPoints[0] = new Point(6,8);
        sortPoints[1] = new Point(4,4);
        sortPoints[2] = new Point(12,3);

        Array.Sort(sortPoints);

        foreach( Point p in sortPoints)
              Console.WriteLine("P:  {0}",p.ToString());
        Console.ReadLine();
  }
```

```
}
```

Multiple sorting parameters

```
using System;
using System.Collections;

public class Point:ICloneable,IComparable
{
  private int x,y;
  private byte r,g,b;
  public Point()
  {}
  public Point( int x,int y,byte r,byte g, byte b)
  {
      this.x = x;
      this.y = y;
      this.r = r;
      this.g = g;
      this.b = b;

  }
  public int X
  {
     get{return x;}

  }
  public int Y
  {
     get{return y;}

  }

  public int R
  {
     get{return r;}

  }


  public void Change(int x, int y)
  {
       this.x = x;
       this.y = y;
  }
  public override string ToString()
  {
```

```
    return "x:"+x+"y:"+y+"R:"+r+"G"+g+"B"+b;

  }

  public object Clone()
  {
    return new Point(this.x,this.y,
this.r,this.g,this.b);
  }

  int IComparable.CompareTo(object o)
  {
       Point tmp = (Point) o;
       double thisLen =
          Math.Sqrt( (this.x*this.x)+(this.y*this.y));
       double tmpLen =
          Math.Sqrt( (tmp.x*tmp.x)+(tmp.y*tmp.y));

       if(thisLen > tmpLen)
            return 1;

       if(thisLen < tmpLen)
            return -1;
       else
            return 0;
  }


}

public class ColorTest :IComparer
{

    int IComparer.Compare(object a, object b)
    {
          Point tmpa = (Point) a;  // CHECK RED
          Point tmpb = (Point) b;

          if( tmpa.R>tmpb.R)
             return 1;

          if( tmpa.R<tmpb.R)
             return -1;

          else
             return 0;
     }
}
```

```
public class UsePoints
{


  public static void Main()

  {

        Point[] colorPoints = new Point[3];
        colorPoints[0] = new Point(6,8, 100,100,200);
        colorPoints[1] = new Point(4,4,40,56,255);
        colorPoints[2] = new Point(12,3,89,66,50);

        Array.Sort(colorPoints, new ColorTest());

        foreach( Point p in colorPoints)
              Console.WriteLine("P:  {0}",p.ToString());
        Console.ReadLine();
  }


}
```

- Array.Sort uses the IComparer.Compare

ArrayList class (since .NET 0.0) ☺

Example: ArrayList (parts of the class):

```
...
     Point[] colorPoints = new Point[3];
     colorPoints[0] = new Point(6,8, 100,100,200);
     colorPoints[1] = new Point(4,4,40,56,255);
     colorPoints[2] = new Point(12,3,89,66,50);
     ArrayList pointBeholder = new ArrayList();
     for(int i = 0; i <3;i++)
        pointBeholder.Insert(i,colorPoints[i]);
     pointBeholder.Add(new Point(1,1,0,0,0));
     foreach( Point p in pointBeholder)
        Console.WriteLine("indhold: {0}",
                                  p.ToString());
...
```

- Problems?
    - boxing – unboxing

Collections & Generics

System.Collections namespace

- Interfaces

System.Collections Interface	Meaning in Life
ICollection	Defines general characteristics (e.g., size, enumeration, and thread safety) for all nongeneric collection types.
ICloneable	Allows the implementing object to return a copy of itself to the caller.
IDictionary	Allows a nongeneric collection object to represent its contents using key/value pairs.
IEnumerable	Returns an object implementing the IEnumerator interface (see next table entry).
IEnumerator	Enables foreach style iteration of collection items.
IList	Provides behavior to add, remove, and index items in a sequential list of objects.

- classes in System.Collections

Table 9-1. *Useful Types of System.Collections*

System.Collections Class	Meaning in Life	Key Implemented Interfaces
ArrayList	Represents a dynamically sized collection of objects listed in sequential order.	IList, ICollection, IEnumerable, and ICloneable
BitArray	Manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0).	ICollection, IEnumerable, and ICloneable
Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.	IDictionary, ICollection, IEnumerable, and ICloneable
Queue	Represents a standard first-in, first-out (FIFO) collection of objects.	ICollection, IEnumerable, and ICloneable
SortedList	Represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.	IDictionary, ICollection, IEnumerable, and ICloneable
Stack	A last-in, first-out (LIFO) stack providing push and pop (and peek) functionality.	ICollection, IEnumerable, and ICloneable

Generics

- Generic
  - classes
  - interfaces
  - structs
  - and delegates
  - All with type parameters

Generic members (Array.Sort<>)

Example:

int[] myInts = {10,12,3,4,38};

Array.Sort<int>(myInts);

The System.Collections.Generic namespace

- Interfaces

Table 9-4. Key Interfaces Supported by Classes of System.Collections.Generic

System.Collections.Generic Interface	Meaning in Life
ICollection<T>	Defines general characteristics (e.g., size, enumeration, and thread safety) for all generic collection types.
IComparer<T>	Defines a way to compare to objects.
IDictionary<TKey, TValue>	Allows a generic collection object to represent its contents using key/value pairs.
IEnumerable<T>	Returns the IEnumerator<T> interface for a given object.
IEnumerator<T>	Enables foreach-style iteration over a generic collection.
IList<T>	Provides behavior to add, remove, and index items in a sequential list of objects.
ISet<T>	Provides the base interface for the abstraction of sets.

- Classes:

Generic Class	Supported Key Interfaces	Meaning in Life
Dictionary<TKey, TValue>	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	This represents a generic collection of keys and values.
LinkedList<T>	ICollection<T>, IEnumerable<T>	This represents a doubly linked list.
List<T>	ICollection<T>, IEnumerable<T>, IList<T>	This is a dynamically resizable sequential list of items.
Queue<T>	ICollection (Not a typo! This is the nongeneric collection interface) , IEnumerable<T>	This is a generic implementation of a first-in, first-out (FIFO) list.
SortedDictionary<TKey, TValue>	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	This is a generic implementation of a sorted set of key/value pairs.
SortedSet<T>	ICollection<T>, IEnumerable<T>, ISet<T>	This represents a collection of objects that is maintained in sorted order with no duplication.
Stack<T>	ICollection (Not a typo! This is the nongeneric collection interface) , IEnumerable<T>	This is a generic implementation of a last-in, first-out (LIFO) list.

Example: List<T> class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericPlay
{
    public class Fugl
    {
        private string navn;

        public string Navn
        {
            get { return navn; }
        }
        public Fugl(string navn)
        {
            this.navn = navn;
        }

    }
    public class Program
```

```
    {
        static void Main(string[] args)
        {
            List<Fugl> mineFugle = new List<Fugl>()
                { new Fugl("undulat"),
                  new Fugl("papegøje")
                };

            foreach (Fugl f in mineFugle)
                Console.WriteLine(f.Navn);


            Console.ReadLine();

        }
    }
}
```

- Contents (partial listning)

```
// A partial listing of the List<T> class.
namespace System.Collections.Generic
{
public class List<T> :
IList<T>, ICollection<T>, IEnumerable<T>, IReadOnlyList<T>
IList, ICollection, IEnumerable
{
...
public void Add(T item);
public ReadOnlyCollection<T> AsReadOnly();
public int BinarySearch(T item);
public bool Contains(T item);
public void CopyTo(T[] array);
public int FindIndex(System.Predicate<T> match);
public T FindLast(System.Predicate<T> match);
public bool Remove(T item);
public int RemoveAll(System.Predicate<T> match);
public T[] ToArray();
public bool TrueForAll(System.Predicate<T> match);
public T this[int index] { get; set; }
}
}
```

Example: Stack<T> class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericPlay
{
    public class Fugl
    {
        private string navn;

        public string Navn
        {
            get { return navn; }
        }
        public Fugl(string navn)
        {
            this.navn = navn;
        }

    }
    public class Program
    {
        static void Main(string[] args)
        {
            Stack<Fugl> mineFugle = new Stack<Fugl>();
            mineFugle.Push(new Fugl("undulat"));
            mineFugle.Push(new Fugl("papegøje"));


            Console.WriteLine(mineFugle.Pop().Navn);
            Console.WriteLine(mineFugle.Pop().Navn);
            Console.ReadLine();




        }
    }
}
```

- Uses
  - Push()
  - Pop()

Example: Queue<T> class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericPlay
{
    public class Fugl
    {
        private string navn;

        public string Navn
        {
            get { return navn; }
        }
        public Fugl(string navn)
        {
            this.navn = navn;
        }

    }
    public class Program
    {
        static void Main(string[] args)
        {

            Queue<Fugl> minepippipFugle= new Queue<Fugl>();

            minepippipFugle.Enqueue(new Fugl("undulat"));
            minepippipFugle.Enqueue(new Fugl("papegøje"));
            minepippipFugle.Enqueue(new Fugl("spurv"));

             Console.WriteLine("Første i kø: {0}",minepippipFugle.Peek().Navn);

             Console.WriteLine("Udskriv køen");


             try
             {
                 for(int i = 0; i<3;i++)
                   Console.WriteLine("Fugl nr.{0}: {1}", i,minepippipFugle.Dequeue().Navn);
             }
             catch (InvalidOperationException ioe)
             {
                 Console.WriteLine("Exception: " + ioe.Message);
             }



            Console.ReadLine();


        }
```

```
    }
}
```

Example: SortedSet<T> class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace GenericPlay
{
    public class Fugl
    {
        private string navn;

        public string Navn
        {
            get { return navn; }
        }
        public Fugl(string navn)
        {
            this.navn = navn;
        }

    }

    public class SorteredeFugle:IComparer<Fugl>
    {
        public int Compare(Fugl første, Fugl næste)
        {
            if(første.Navn.Length> næste.Navn.Length)
                return 1;
            if (første.Navn.Length < næste.Navn.Length)
                return -1;
            else
                return 0;
        }


    }
    public class Program
    {
        static void Main(string[] args)
        {


            SortedSet<Fugl> sortFugle = new SortedSet<Fugl>(new SorteredeFugle())
```

```
            {
                new Fugl("undulat"),
                new Fugl("spurv"),
                new Fugl("dværgpapegøje"),
                new Fugl("due")


            };


             foreach (Fugl f in sortFugle)
                 Console.WriteLine(f.Navn);




            Console.ReadLine();









        }
    }
}
```

- Sorts objects!

ObservableCollection<T>

- Uses a delegate to detect changes
- Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Collections.ObjectModel;
using System.ComponentModel;

namespace ObservableColExample
{

    public class Car
    {
        private int ID;
        private String RegNo;
        private String name;
```

```
        public Car(int ID, string name, string RegNo)
        {
            this.ID = ID;
            this.RegNo = RegNo;
            this.name = name;
        }

        public void NewRegNo(string RegNo)
        {
            this.RegNo = RegNo;
        }

        public string ReturnInfo()
        {
            return "ID: " + ID.ToString()+ " name: "+name+" RegNo: " + RegNo ;
        }


    }
    public class Program
    {

        static void Main(string[] args)
        {
            ObservableCollection<Car> cars =
            new ObservableCollection<Car>()
            {
              new Car(678,"Lada","AA10100"),
              new Car(128,"Skoda","AA10101"),
              new Car(765,"Peugeot","AA10102"),

            };

            cars.CollectionChanged += cars_CollectionChanged;


            cars.RemoveAt(0);

            foreach (Car c in cars)
            {

                Console.WriteLine(c.ReturnInfo());
            }

            Console.WriteLine("Changing Reg. number");

            Car carToChange= cars.ElementAt(1);
            carToChange.NewRegNo("AA64101");



            foreach (Car c in cars)
            {

                Console.WriteLine(c.ReturnInfo());
            }
```

```
        }

        static void cars_CollectionChanged(object sender,
                        System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
        {

            Console.WriteLine(e.Action.ToString());

        }


    }


}
```

Own generic classes and structs

```
...
public struct Point<T>
    {
        // Generic state date.
        private T xPos;
        private T yPos;

        public Point(T xVal, T yVal)
        {
            xPos = xVal;
            yPos = yVal;
        }

        // Generic properties.
        public T X
       {
            get { return xPos; }
            set { xPos = value; }
       }

       public T Y
       {
```

```
            get { return yPos; }
            set { yPos = value; }
        }

        public override string ToString()
        {
         return string.Format("[{0}, {1}]", xPos, yPos);
        }

 // The 'default' keyword is overloaded in C# 2.0.
 // when used with generics, it represents the default
 // value of a generic parameter.
         public void ResetPoint()
         {
            xPos = default(T);
            yPos = default(T);
                     }
}
...
```

- Notice the **default** keyword
- The Type is decided on run-time

About <T>

- Can (of course) be anything else than <T>

```
...
   public struct Point<P>
    {
        private P xP;
        private P yP;

        public Point(P x, P y)
         {

                 xP =x;
                 yP =y;
         }

        public override string ToString()
        {
            return string.Format("[{0}, {1}]", xP, yP);
        }
```

```
...
```

Use:

```
...

            Point<float> pointF = new Point<float>(12.4f, 3.78f);
            Point<int> pointD = new Point<int>(12, 4);
            Console.WriteLine(pointF.ToString());
            Console.WriteLine(pointD.ToString());


...
```

More about types

- Several ”args”

```
  public struct Pixel<P,T>
   {
       private P xP;
       private P yP;
       private T red;
       private T green;
       private T blue;

       public Pixel(P x, P y,T red, T green, T blue)
       {

           xP = x;
           yP = y;

           this.red    = red;
           this.green  = green;
           this.blue   = blue;
       }

       public override string ToString()
       {
           return string.Format("[{0}, {1},{2},{3},{4} ]", xP, yP,red,green,blue);
       }

   }
```

```
...
```

```
      Pixel<float, float> pixelA = new Pixel<float, float>(12.4f, 6.4f, 0f, 210.20f, 16.0f);
      Pixel<float, int> pixelB = new Pixel<float, int>(12.4f, 6.4f, 48,16, 155);
      Console.WriteLine(pixelA.ToString());
      Console.WriteLine(pixelB.ToString());


...
```

Generic methods

- Also known from C++

```
...
// Swaps any type
public static void Swap<T>(ref T a, ref T b)
{
   Console.WriteLine("You sent the Swap() method a {0}",
                                        typeof(T));
    T temp;
    temp = a;
    a = b;
    b = temp;
}
...
```

- Notice ref (the reference)

Generic interfaces

Example: generic interface:

```
public interface IBinaryOperations<T>
 {
    T Add(T arg1, T arg2);
    T Subtract(T arg1, T arg2);
    T Multiply(T arg1, T arg2);
    T Divide(T arg1, T arg2);
 }
```

```
public class BasicMath : IBinaryOperations<int>
{
  // Implementations

  public int Add(int arg1, int arg2)
  { return arg1 + arg2; }

  public int Subtract(int arg1, int arg2)
  { return arg1 - arg2; }

  public int Multiply(int arg1, int arg2)
  { return arg1 * arg2; }

  public int Divide(int arg1, int arg2)
  { return arg1 / arg2; }

}
```

Where (limitations)

Table 9-8. *Possible Constraints for Generic Type Parameters*

Generic Constraint	Meaning in Life
where T : struct	The type parameter <T> must have System.ValueType in its chain of inheritance (i.e., <T> must be a structure).
where T : class	The type parameter <T> must not have System.ValueType in its chain of inheritance (i.e., <T> must be a reference type).
where T : new()	The type parameter <T> must have a default constructor. This is helpful if your generic type must create an instance of the type parameter because you cannot assume you know the format of custom constructors. Note that this constraint must be listed last on a multiconstrained type.
where T : NameOfBaseClass	The type parameter <T> must be derived from the class specified by NameOfBaseClass
Generic Constraint	**Meaning in Life**
where T : NameOfInterface	The type parameter <T> must implement the interface specified by NameOfInterface. You can separate multiple interfaces as a comma-delimited list.