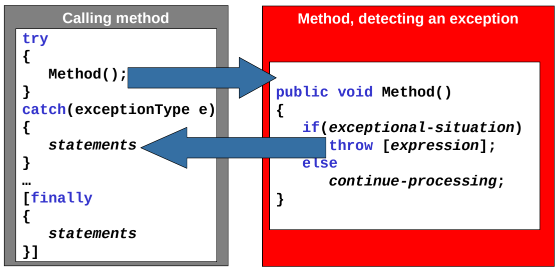
Error Handling with Exceptions

Exceptions – object-oriented solutions for error handling when the normal flow of code path is impractical or imprudent. There is a difference between an exception and an expected event, such as reaching the end of file.

Exception Handling – with key words try, catch, throw and finally.

When a method detects an exceptional situation, it throws an exception to the calling method by using the throw keyword. The calling method receives this exception via the catch keyword and decides what course of action to take.

try, catch, throw, finally



```
try
{
   Method();
}
catch(exceptionType e)
{
   statements
}
…
[finally
{
   statements
}]
```

```
public void Method()
{
   if(exceptional-situation)
      throw [expression];
   else
      continue-processing;
}
```

System.Exception Class

The base class – represents errors that occur during application execution.

Constructors:

- Default constructor – initializes a new instance of the Exception class.

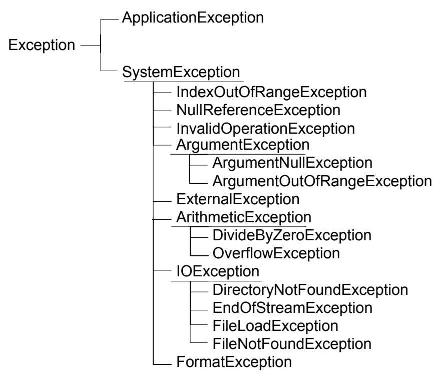
  public Exception();

- With a parameter string message – initializes a new instance of the Exception class with a specified error message. (The property Exception.Message gets a message that describes the current exception.)

  public Exception (string message);

- With two parameters: string message – specifies the error message and Exception innerException – a reference to the inner exception that is the cause of this exception

  public Exception (string message, Exception innerException);



1. Throwing an Exception

throw [*expression*];

expression – optional;
an object of type System.Exception
(or a derived class)

```
public void <method_name> ()
{
   if (<exceptional situation>)
      throw new Exception();
   else
      <continue processing>;
}
```

2. Catching an Exception

```
try
{
    <statements>
}
catch (<exception_type> <exception>)
{
    <statements>
}
...
[finally
{
    <statements>
}]
```

3. Throwing Exceptions from Constructors – signal to the constructor's calling method that an error occurred during object construction

Example: InRange class enters a number in a range – it throws an exception and the calling method Main catches the exception via the try-catch statement.

```
using System;
class InRange
{
   private int number;
   public int Number
   {
       get
       {   return number;   }
   }
```

```
   public InRange (int min, int max)
   {
      Console.Write ("Enter an integer in the range [{0},{1}]: ",
                         min, max);
      number = int.Parse (Console.ReadLine());
      if (number < min || number > max)
         throw new Exception ("The number is out of range");
   }
}
```

```
class Test
{
   static void Main (string[] args)
   {  int n;
      InRange r;
      try
      {  r = new InRange (-10, 10);
         n = r.Number;
      }
      catch (Exception e)
      {
         Console.WriteLine("Exception caught - '{0}'",e.Message);
      }
   }
}
```

Results:

```
Enter an integer in the range [-10,10]: 20
Exception caught - 'The number is out of range'
```

The Parse method throws the System.FormatException exception when the format of an argument does not meet the parameter specifications of Parse and the Main method catches this exception.

```
Enter an integer in the range [-10,10]: abc
Exception caught - 'Input string was not in the correct format'
```

If a method doesn't catch an exception the runtime aborts the application.

4. Rethrowing an Exception

After a method catches an exception, it can rethrow the exception back up the call stack using the throw keyword, so that further error processing can be done.

Example: The InRange class constructor calls the Parse method, that can throw any of three exceptions (ArgumentNullException, FormatException, OverflowException) and catches the base class exception. If the user enters incorrect string, the constructor passes the exception back up the calling stack so that the caller knows that the number is not valid.

```
using System;
class InRange
{
    private int number;
    public int Number
    {
        get   {   return number;   }
    }
```

```
    public InRange (int min, int max)
    {
        Console.Write ("Enter an integer in the range [{0},{1}]: ",
                                 min, max);
        try
        {
            number = int.Parse (Console.ReadLine());
        }
        catch (Exception e)
        {   Console.WriteLine
            ("Caught exception of type - '{0}'", e.GetType());
            throw;
        }
        if (number < min || number > max)
           throw new Exception ("The number is out of range");
    }
}
```

```
class Test
{   static void Main (string[] args)
    {   int n;
        InRange r;
        try
        {   r = new InRange(-10,10);
            n = r.Number;
        }
        catch (Exception e)
       {
           Console.WriteLine ("Caught exception - '{0}'",e.Message);
       }
    }
}
```

Results:

```
Enter an integer in the range [-10,10]: abc
Caught exception of type - 'System.FormatException'
Caught exception - 'Input string was not in the correct format'
```

5. Cleaning Up with finally

The code in the finally block is always run.

Example:

```
using System;
class InRange
{
    private int number;
    public int Number
    {
        get
        {
            return number;
        }
    }
```

```
    public InRange (int min, int max)
    {   Console.Write ("Enter an integer in the range [{0},{1}]: ",
                                 min, max);
        try
        {
            number = int.Parse (Console.ReadLine());
        }
        catch (Exception e)
        {
           Console.WriteLine
            ("Caught an exception of type - '{0}'",e.GetType());
            throw;
        }
        if (number < min || number > max)
           throw new Exception ("The number is out of range");
    }
}
```

```
class Test
{   static void Main (string[] args)
    {   int n;
        InRange r = null;
        try
        {
            r = new InRange(-10,10);
             n = r.Number;
        }
        catch (Exception e)
        {
           Console.WriteLine("Caught exception - '{0}'", e.Message);
        }
```

```
      finally
      {   if (r != null)
             Console.WriteLine
                           ("Successfully object construction!");
          else
             Console.WriteLine
                           ("Unsuccessfully object construction!");
      }
    }
}
```

Results:

```
Enter an integer in the range [-10,10]: 20
Caught exception - 'The number is out of range'
Unsuccessfuly object construction!
```

6. Catching Multiple Exception Types

Rule: The base class is handled last – if its catch block is first, the other catch blocks would be unreachable.

Example:

```
using System;
class MultipleExceptions
{
   static void Main (string[] args)
   {
      int [] A = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
      int i, k;
      i = 11;
      k = 3;
```

```
      try
      {
         int temp = A[i] / k;
         A[i+1] = int.Parse ("10" + temp);
      }
      catch (System.IndexOutOfRangeException e)
      {
         Console.WriteLine ("Error in indexing");
      }
      catch (System.DivideByZeroException e)
      {
        Console.WriteLine ("Error in divide by zero");
      }
      catch (System.FormatException e)
      {
         Console.WriteLine
            ("Error in converting a string into an integer");
      }
```

```
      catch (System.Exception e)
      {
         Console.WriteLine ("Error: {0}", e.ToString());
      }
   }
}
```

Results:

```
Error in indexing
```

7. Deriving an Own Exception Class

- Derive an own Exception class
- Put the needed code in the class constructor

Rules:

- It's good programming practice to use Exception suffix at the end of the exception class name
- Implement all three System.Exception constructors

Example:

```
using System;
public class MonthException : Exception
{
   public MonthException () : base ()
   {
      LogException();
   }
   public MonthException (String message) : base (message)
   {
      LogException();
   }
   public MonthException (String message,
    Exception innerException) : base (message, innerException)
   {
     LogException();
   }
```

```
  protected void LogException()
  {
     Console.WriteLine ("Error: {0}: {1}.",
                              this.GetType(), this.Message);
  }
}
```

```
class DerivedExceptionApp
{   public static string MonthName (int month)
    {   switch (month)
        { case 1:   return "January";
          case 2:   return "February";
          case 3:   return "March";
          case 4:   return "April";
          case 5:   return "May";
          case 6:   return "June";
          case 7:   return "July";
          case 8:   return "August";
          case 9:   return "September";
          case 10: return "October";
          case 11: return "November";
          case 12: return "December";
          default:   throw new MonthException ("Month Error");
        }
    }
```

```
  static void Main()
  {   string month = null;
      try
      {
         month = MonthName (13);
      }
      catch (Exception e)
      {
         Console.WriteLine ("Error! {0}", e.ToString());
      }
      finally
      {
         if (month != null)
            Console.WriteLine ("The month is {0}.", month);
      }
   }
}
```

Results:

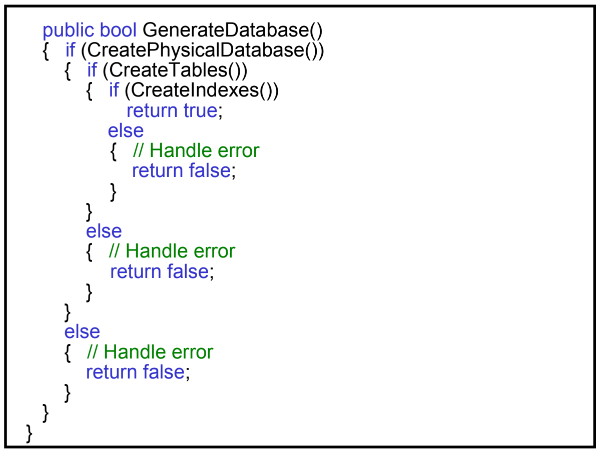
```
Error: MonthException: Month Error.
Error! MonthException: Month Error
   at DerivedExceptionApp.MonthName(Int32 month) in
     f:\...monthexception\class1cs: line 35
   at DerivedExceptionApp.Main() in
     f:\...monthexception\class1cs: line 43
```

8. Comparing Error-Handling Techniques

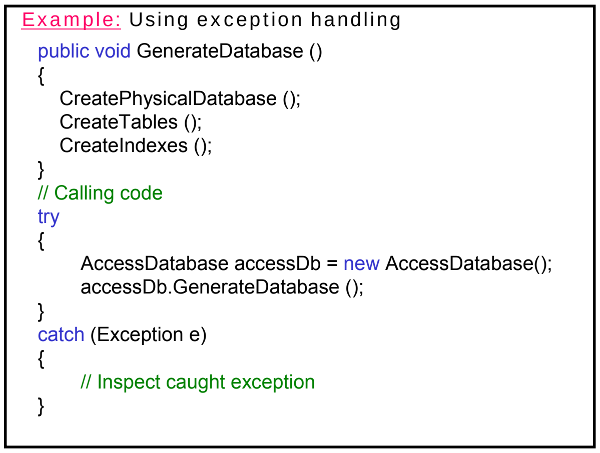
- Return an error code – standard approach to error handling;
  - disadvantage: there's no guarantee that the caller will check the returned error code.
- Benefits of exception handling over return codes:
  - There's guarantee that the caller will handle the exception – control is passed back up the call stack and the caller would be forced to deal with
  - Handling error in the correct context – the error information is contained within a class, new error conditions can be added and the calling method will remain unchanged: enables extensibility – the most significant advantage
  - Improving code readability

Example: Using return code

```
class AccessDatabase
{
   public bool CreatePhysicalDatabase ()
   {
   }
   public bool CreateTables ()
   {
   }
   public bool CreateIndexes ()
   {
   }
```



```
    public bool GenerateDatabase()
    {   if (CreatePhysicalDatabase())
        {   if (CreateTables())
            {   if (CreateIndexes())
                    return true;
                else
                {   // Handle error
                    return false;
                }
            }
            else
            {   // Handle error
                return false;
            }
        }
        else
        {   // Handle error
            return false;
        }
    }
}
```



Example: Using exception handling

```
public void GenerateDatabase ()
{
    CreatePhysicalDatabase ();
    CreateTables ();
    CreateIndexes ();
}
// Calling code
try
{
        AccessDatabase accessDb = new AccessDatabase();
        accessDb.GenerateDatabase ();
}
catch (Exception e)
{
        // Inspect caught exception
}
```