C# Advanced Cheat Sheet (1 of 4))

Generics

C# has two separate mechanisms for writing code that is reusable across different types: inheritance and generics. Whereas inheritance expresses re-usability with a base type, generics express reusability with a "template" that contains "placeholder" types.

```
using System;
using System.Collections.Generic;
namespace Generics
{
    public class Customer
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            List<Customer> myCustomers = new List<Customer>(); //empty
            myCustomers.Add(new Customer() { Id = 1, Name = "Jack" });
            myCustomers.Add(new Customer() { Id = 2, Name = "Jill" });
            foreach (Customer cust in myCustomers) {
                                Console.WriteLine(cust.Name); }
        }
    }
}
```

Lists of Objects

Lists were covered briefly in the Basics section of this cheat sheet series, but the example was only a list of integers. Here was have a list of our own objects based on our own class: Customer.

```
class Customer
{
    public int Id = 0;
    public string Name = "";
    public string Status { get; set; }
}
class Repository
{
    private static List<Customer> privatecust = new List<Customer>();
    public static IEnumerable<Customer> Customers {
        get { return privatecust; }
    }
    public static void AddCustomers(Customer customer) {
        privatecust.Add(customer);
    }
    public static int NumberOfCustomers {
        get { return privatecust.Count; }
    }
}
class Program
{
    static void Main(string[] args)
    {
        var cust1 = new Customer { Id = 1, Name = "Joe",
            Status = "Active" };
        var cust2 = new Customer { Id = 1, Name = "Sally",
            Status = "Active" };
        Repository.AddCustomers(cust1);
        Repository.AddCustomers(cust2);
        foreach (Customer cust in Repository.Customers)
        { Console.WriteLine($"Name: {cust.Name} Id: {cust.Id} " +
            $"Status: {cust.Status}"); }
        Console.WriteLine($"Number of customers: " +
            $"{Repository.NumberOfCustomers}");
    }
}
```

Here is another example of a list of objects, without the Repository.

```
class Program
{
    public class Customer
    {   // mix fields with a property just for demonstration
        public int Id = 0;
        public string Name = "";
        public string Status { get; set; }
    }
    static void Main(string[] args)
    {
        var customers = new List<Customer>
        {   // using object initialization syntax here
            new Customer { Id = 4, Name = "Jack", Status = "Active"},
            new Customer { Name = "Sally", Status = "Active"}
        };
        customers.Add(new Customer { Name = "Sam" });
        foreach (Customer cust in customers)
                  { Console.WriteLine(cust.Id + " " + cust.Name +
                                                  " " + cust.Status); }
    }
}
```

Delegates

A delegate is an object that “holds” one or more methods. A delegate is a reference to a function or ordered list of functions with a specific signature. You can “execute” a delegate and it will execute the method or methods that it “contains” (points to). A delegate is a user-defined reference type, like a class. You can create your own delegate or use the generic ones: `Func<>` and `Action<>`. First, we’ll create our own.

```
class Program
{
    delegate int Multiplier(int x);  // type declaration
    static void Main()
    {
        Multiplier t = Cube; // Create delegate instance
        // by assigning a method to a delegate variable.
        int result = t(2); // Invoke delegate: t(3)
        Console.WriteLine(result); // 8
    }
    static int Cube(int x) => x * x * x;
}
```

Here is second example.

```
using System;
namespace ReturnValues
{
    // Illustrated C# 7 Fifth Edition page 361
    delegate int MyDel(); // Declare delegate with return value.
    class MyClass
    {
        private int IntValue = 5;
        public int Add2() { IntValue += 2; return IntValue; }
        public int Add3() { IntValue += 3; return IntValue; }
    }
    class Program
    {
        static void Main()
        {
            MyClass mc = new MyClass();
            MyDel mDel = mc.Add2; // Create initialize delegate.
            mDel += mc.Add3; // Add a method.
            mDel += mc.Add2; // Add a method.
            Console.WriteLine($"Value: { mDel() }");  // output 12
        }
    }
}
```

Here is a more realistic example of delegates. Here we create a multicast delegate. The consumer of our code is the method `Main()`. We have an object that we need to “process” with several methods in order, and we also want the code to be **extensible** so the consumer can add their own methods in `Main()` to the list of our methods.

```
class MyClass
{
    public string MyString { get; set; }
    public int MyInt { get; set; }

    public static MyClass MyClassDoMethod()
    {
        return new MyClass();  // we don't use these
    }
}
```

Our code has 3 methods that act upon the above class. They are: `AddOne()`, `DoubleIt()` and `AppendString()`.

```
class MyClassMethods
{
    public void AddOne(MyClass mc)
    {  // here we do something with the object mc
        mc.MyInt = mc.MyInt + 1;
        Console.WriteLine("AddOne: " + mc.MyString + " " + mc.MyInt);
    }
    public void DoubleIt(MyClass mc)
    {
        mc.MyInt = mc.MyInt * 2;
        Console.WriteLine("DoubleIt:" + mc.MyString + " " + mc.MyInt);
    }
    public void AppendString(MyClass mc)
    {
        mc.MyString = mc.MyString + " appending string now ";
        Console.WriteLine("AppendString: " + mc.MyString + " "
                                                + mc.MyInt);
    }
}
class MyClassProcessor
{
    public int MyAmount { get; set; }
    public delegate void MyClassMethodHandler(MyClass myclass);

    public void Process(MyClassMethodHandler methodHandler)
    {   // methodHandler is a delegate
        // instantiate with object initialization syntax
        var myclass = new MyClass { MyString = "In Process method ",
                                    MyInt = 1 };
        methodHandler(myclass);
        // we do not define the methods we want to run here because
        // we are going to let the consumer define that.
    }
}
class Program
{
    static void Main(string[] args)
    {
        var myclassprocessor = new MyClassProcessor();
        var myclassmethods = new MyClassMethods();
        MyClassProcessor.MyClassMethodHandler
                        methodHandler = myclassmethods.AddOne;
        // MyClassMethodHandler is a delegate (multicast)
        // methodHandler is pointer to a group of functions (delegate)
        methodHandler += myclassmethods.DoubleIt;
        methodHandler += FromConsumerMinusThree;
        methodHandler += myclassmethods.AppendString;

        // Process() takes a delegate
        myclassprocessor.Process(methodHandler);
    }
    static void FromConsumerMinusThree(MyClass myC)
    {
        myC.MyInt = myC.MyInt - 3;
        Console.WriteLine("FromConsumerMinusThree: " + myC.MyString +
                                    myC.MyInt);
    }
}
```

Output:

```
AddOne: inside Process method  2
DoubleIt: inside Process method  4
FromConsumerMinusThree: inside Process method 1
AppendString: inside Process method  appending string now  1
```

Func<> and Action<>

In .NET we have 2 delegates that are generic: `Action<>` and `Func<>`. Each also come in a non-generic form. Modifying the above program requires us to use `Action<>` and introducing a new processor (we’ll call it `MyClassGenericProcessor`) and removing our custom delegate in there and adding `Action<>`. Also in the `Main()` program we need to change the first line and the third line of code.

C# Advanced Cheat Sheet (2 of 4)

Func<> and Action<> continued…

```
class MyClassGenericProcessor
{
    public int MyAmount { get; set; }
    // public delegate void MyClassMethodHandler(MyClass myclass);

    public void Process(Action<MyClass> methodHandler)
    {   // methodHandler is a delegate
        // instantiate with object initialization syntax
        var myclass = new MyClass { MyString = "in Process method ",
                                                    MyInt = 1 };
        methodHandler(myclass);
        // we do not define the methods we want to run here because
        // we are going to let the consumer define that.
    }
}
```

Below is a partial listing of our `Main()` program showing the changes.

```
var myclassprocessor = new MyClassGenericProcessor(); // generics
var myclassmethods = new MyClassMethods();
Action<MyClass> methodHandler = myclassmethods.AddOne;
```

Anonymous Types

An anonymous type is a simple class created on the fly to store a set of values. To create an anonymous type, you use the **new** keyword followed by an object initializer **{ }**, specifying the properties and values the type will contain. Anonymous types are used in LINQ queries.

```
static void Main(string[] args)
{
    var person = new { Name = "Bob", Number = 32 };
    Console.WriteLine($"Name: {person.Name} " +
        $"Number: {person.Number}");
    // output: Name: Bob Number: 32
}
```

Here is another example.

```
class Program
{
    static void Main(string[] args)
    {
        var person = new
        {
            Name = "John",
            Age = 29,
            Major = "Computers"
        };
        Console.WriteLine($"{ person.Name }, Age { person.Age }, "
                          + $"Major: {person.Major}");
        // the code below produces the same results
        string Major = "Computers";
        var guy = new { Age = 29, Other.Name, Major };
        Console.WriteLine($"{guy.Name }, Age {guy.Age }, "
                          + $"Major: {guy.Major}");
        // John, Age 29, Major: Computers
    }
}
class Other
{
    // Name is a static field of class Other
    static public string Name = "John";
}
```

Lambda

A lambda expression is an unnamed method written in place of a delegate instance. A lambda expression is an anonymous method that has no access modifier, no name and no return statement. We have code below that we can re-factor using a lambda expression. The => is read as “goes to”.

```
class Program
{
    delegate int MyDel(int InParameter); // custom delegate

    static void Main(string[] args)
    {
        MyDel AddTwo = x => x + 2;
        Func<int, int> AddThree = number => number + 3;
        Console.WriteLine(AddOne(0));
        Console.WriteLine(AddTwo(0));
        Console.WriteLine(AddThree(0));
    }
    static int AddOne(int number)
    {
        return number + 1;
    }
}
```

Here is another example.

```
static void Main(string[] args)
{
    Console.WriteLine(Square(3));  // 9
    Func<int, int> squareDel = Square;
    Console.WriteLine(squareDel(3));  // 9
    Func<int, int> squareLambda = m => m * m;
    Console.WriteLine(squareLambda(3));  // 9
    Func<int, int, long> multiplyTwoInts = (m, n) => m * n;
    Console.WriteLine(multiplyTwoInts(3,4));  // 12
}
static int Square(int number)
{
    return number * number;
}
```

Here is another example that is more realistic. Here we have a list of Products. We also have a repository of products. We use object initialization syntax to initialize the list with a series of products. `FindAll()` takes a predicate. A predicate is something that evaluates to true or false.

```
class Product
{
    public string Title { get; set; }
    public int Price { get; set; }
}

class ProductRepository
{
    public List<Product> GetProducts()
    {
        return new List<Book>
            {
                new Product () { Title ="product 1", Price = 5},
                new Product () { Title = "product 2", Price = 6 },
                new Product () { Title = "product 3", Price = 17 }
            };
    }
}
class Program
{
    static void Main(string[] args)
    {
        var products = new ProductRepository().GetProducts();
        List<Product> cheapProducts = products.FindAll(b =>
                                                b.Price < 10);
        foreach (var product in cheapProducts)
        {
            Console.WriteLine(product.Title + " $" + product.Price);
        }
    }
}
```

You can use a lambda expression when argument requires a delegate.

Events

1. define a delegate (define signature) or use EventHandler<>
2. define an event based on that delegate (ItemProcessed in this case)
3. raise the event

Here is an example program that uses events.

```
public class Item
{
    public string Name { get; set; }  // a property
}
public class ItemEventArgs : EventArgs
{
    public Item Item { get; set; }
}
public class ItemProcessor
{
    // public delegate void ItemProcessedEventHandler(object source,
    //                                                 ItemEventArgs args);
    public event EventHandler<ItemEventArgs> ItemProcessed;

    public void ProcessItem(Item item)
    {
        Console.WriteLine("Processing Item...");
        Thread.Sleep(1500); // delay 1.5 seconds
        OnItemProcessed(item);
    }
    protected virtual void OnItemProcessed(Item item)
    {
        ItemProcessed?.Invoke(this, new ItemEventArgs() { Item = item
});
        //  if (ItemProcessed != null)
        //  ItemProcessed(this, new ItemEventArgs() { Item = item });
    }
}
public class SubscriberOne
{
    public void OnItemProcessed(object source, ItemEventArgs args)
    {  // maybe send an email
        Console.WriteLine("SubscriberOne: " + args.Item.Name);
    }
}
class SubscriberTwo
{
    public void OnItemProcessed(object source, ItemEventArgs args)
    {  // maybe send SMS (text message)
        Console.WriteLine("SubscriberTwo: " + args.Item.Name);
    }
}
```

Here is the main program.

```
class Program
{
    static void Main(string[] args)
    {
        var item = new Item() { Name = "Item 1 name" };
        var itemProcessor = new ItemProcessor();  // publisher
        var subscriberOne = new SubscriberOne();  // subscriber
        var subscriberTwo = new SubscriberTwo();  // subscriber

        Console.WriteLine("Beginning program EventsExample...");

        // itemProcessed is a list of pointers to methods
        itemProcessor.ItemProcessed += subscriberOne.OnItemProcessed;
        itemProcessor.ItemProcessed += subscriberTwo.OnItemProcessed;

        itemProcessor.ProcessItem(item);
    }
}
```

Attributes

Attributes allow you to add metadata to a program’s assembly. Attribute names use Pascal casing and end with the suffix Attribute. An attribute section consists of square brackets enclosing an attribute name and sometimes a parameter list. A construct with an attribute applied to it is said to be decorated, or adorned, with the attribute. Use the `[Obsolete]` attribute to mark the old method as obsolete and to display a helpful warning message when the code is compiled.

Preprocessor Directives

C# includes a set of preprocessor directives that are mainly used for conditional compilation. The directives `#region` and `#endregion` delimit a section of code that can be expanded or collapsed using the outlining feature of Visual Studio and can be nested within each other.

C# Advanced Cheat Sheet (3 of 4)

Extension Methods

Extension methods allow an existing type to be extended with new methods, without altering the definition of the original type. An extension method is a static method of a static class, where the `this` modifier is applied to the first parameter. The type of the first parameter will be the type that is extended. Extension methods, like instance methods, provide a way to chain functions.

```
public static class MyStringExtensions
{
    public static string Shorten(this String str, int numberOfWords)
    {
        if (numberOfWords < 0) throw new
                ArgumentOutOfRangeException("must contain words");
        if (numberOfWords == 0) return "";
        string[] words = str.Split(' ');
        if (words.Length <= numberOfWords) return str;
        return string.Join(" ", words.Take(numberOfWords)) + "...";
    }
}
class Program
{
    static void Main(string[] args)
    {
        string senten = "A very very long sentence...";
        Console.WriteLine("Number of chars: " + senten.Length);
        var shortededSentence = senten.Shorten(10);
        var s2 = shortededSentence.ToUpper();
        var s3 = s2.PadRight(60);
        Console.WriteLine("[" + s3 + "]");
    }
}
```

LINQ

LINQ stands for Language Integrated Query and is pronounced “link.” LINQ is an extension of the .NET Framework and allows you to query collections of data in a manner like using SQL to query databases. With LINQ you can query data from databases (LINQ to Entities), collections of objects in memory (LINQ to Objects), XML documents (LINQ to XML), and ADO.NET data sets (LINQ to Data Sets).

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace LINQint
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = { 6, 47, 15, 68, 23 }; // Data source
            IEnumerable<int> bigNums = // Define & store the query.
                        from n in numbers
                        where n > 30
                        orderby n descending
                        select n;
            foreach (var x in bigNums) // Execute the query.
                Console.Write($"{ x }, ");  // output: 68, 47
        }
    }
}
```

Now let’s use a more realistic example. First we’ll show the code **without** LINQ, then with LINQ. We have a class of our objects called `Product` and we have a `ProductRepository`.

```
class Product
{
    public string Name { get; set; }
    public float Price { get; set; }
}
class ProductRepository
{
    public IEnumerable<Product> GetProducts()  // method
    {
        return new List<Product>
        {
            new Product() {Name = "P one", Price = 5},
            new Product() {Name = "P two", Price = 9.99f},
            new Product() {Name = "P three", Price = 12},
        };
    }
}
class Program
{
    static void Main(string[] args)
    {
        var products = new ProductRepository().GetProducts();
        var pricyProducts = new List<Product>();
        // ------without LINQ---------------------------
        foreach (var product in products)
        {
            if (product.Price > 10)
                pricyProducts.Add(product);
        }
        // ------without LINQ---------------------------
        foreach (var product in pricyProducts)
           Console.WriteLine("{0} {1:C}",product.Name, product.Price);
    }
}
```

When you type product followed by the dot, Intelisense gives you a few methods and a long list of extension methods. One extension method is Where<>. Where is asking for a delegate. `Func<Product,bool> predicate.` It points to a method that gets a Product and returns a bool based on the predicate. Whenever we see Func<> as a delegate we can use a Lambda expression such as p => p.Price > 10. Here is the code **with** LINQ.

```
// -----with LINQ-----------------------------------------
var pricyProducts2 = products.Where(p => p.Price > 10);
// -----with LINQ-----------------------------------------
```

The LINQ extension methods can be chained. When we use `Select` in this case we get back a list of strings, not products.

```
// -----LINQ-----------------------------------------
var pricyProducts2 = products.Where(p => p.Price > 8)
                             .OrderBy(p => p.Name)
                             .Select(p => p.Name);  // string
// -----LINQ-----------------------------------------
foreach (var product in pricyProducts2)
    Console.WriteLine(product);
```

There are several LINQ extension methods beyond `Where()`. A few are listed in the C# comments below. If you only want one Product you can use `Single()` or `SingleOrDefault()`. `Single()` will throw an error InvalidOperationException if it can’t find a match. The OrDefault will return null if it can’t find a match, which is probably better.

```
var product = products.Single(p => p.Name == "P two");
var product2 = products.SingleOrDefault(p => p.Name == "P unknown");
Console.WriteLine(product.Name);  // P two
Console.WriteLine(product2 == null);  // output: True
var product3 = products.First();
Console.WriteLine(product3.Name); // P one
// FirstOrDefault() Last() LastOrDefult()
// Skip(2).Take(3) will skip the first 2 and take the next 3
// Count() Max() Min() Sum() Average()
// Average(p => p.Price)
```

Nullable Types

Reference types can represent a nonexistent value with a null reference. Normally value types cannot be null, however to represent null in a value type, you must use a special construct called a nullable type which is denoted with a value type immediately followed by the **?** symbol. An important use case for nullable types is when you have a database with a column like MiddleName or BirthDate which may have a null value.

```
static void Main(string[] args)
{
    // DateTime is a value type - cannot be null, but...
    System.Nullable<DateTime> d = null;
    DateTime? dt = null;
    Console.WriteLine("GetValueOrDefault: " + dt.GetValueOrDefault());
    Console.WriteLine("HasValue: " + dt.HasValue);  // property
    // below line causes InvalidOperationException when null
    // Console.WriteLine("Value: " + dt.Value);  // property
    Console.WriteLine(dt);

    // output: 0001-01-01 12:00:00 AM
    // output: False
    // output:
}
```

What about conversions and the null-coalescing operator?

```
// Conversions
DateTime? date = new DateTime(2019, 1, 1);
// DateTime date2 = date;  compiler says cannot convert
DateTime date2 = date.GetValueOrDefault();
Console.WriteLine("date2: " + date2);
DateTime? date3 = date2;
Console.WriteLine(date3.GetValueOrDefault());

// Null Coales Operator: ??
DateTime? date4 = null;
DateTime date5;
// if date has a value use that, otherwise use today
if (date4 != null)
    date5 = date4.GetValueOrDefault();
else
    date5 = DateTime.Today;
// null
date5 = date4 ?? DateTime.Today; // same as if block above
```

When working with nullable types, `GetValueOrDefault()` is the preferred way of doing things.

Dynamics

Programming languages are either static or dynamic. C# and Java are static, but Ruby, JavaScript and Python are dynamic. With static languages the types are resolved at compile time, not at run time. The CLR (.NET’s virtual machine) takes compiled code (verified by the compiler) which is in Intermediate language (IL) and converts that to machine code at runtime. Runtime checking is performed by the CLR. Runtime type checking is possible because each object on the heap internally stores a little type token. You can retrieve this token by calling the `GetType` method of object (reflection). With C# dynamics and the keyword dynamic, we don’t need to use reflection. Much cleaner code results. When converting from dynamic to static types, if the runtime type of the dynamic object can be implicitly converted to the target type we don’t need to cast it.

```
dynamic name = "Bob";
name = 19;  // this works because name is dynamic!
name++;
Console.WriteLine(name);  // 20
dynamic a = 4, b = 5;
var c = a + b; // c becomes dynamic
Console.WriteLine(c);  // 9
int i = 7;
dynamic d = i;
long l = d;
Console.WriteLine(l);  //
```

C# Advanced Cheat Sheet (4 of 4)

Asynchronous

In the synchronous model the program executes line by line, but in the asynchronous model (e.g. media players, web browsers), responsiveness is improved. In .NET 4.5 (in 2012) Microsoft introduced a new a new asynchronous model, instead of multi-threading and call-backs. It uses `async` and `await` keywords. In our example we have a WPF program that has 2 blocking operations (downloading and writing). You can only use the `await` operator inside an `async` method. Async affects only what happens inside the method and has no effect on a method's signature or public metadata.

```
using System.IO;
using System.Net;
using System.Threading.Tasks;
using System.Windows;
namespace AysnchronousProgramming
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private async void Button_Click(object sender,
                                        RoutedEventArgs e)
        {
            await DownloadHtmlAsync("http://begincodingnow.com");
        }
        public async Task DownloadHtmlAsync(string url)
        {   // decorate method async, use Task, and only by convention
            // put "Async" at end of the method name.
            var webClient = new WebClient();
            // use TaskAsync not Async, and await is a compiler marker
            var html = await webClient.DownloadStringTaskAsync(url);
            using (var streamWriter = new
                                StreamWriter(@"c:\temp\result.html"))
            {   // use the Async one: WriteAsync and add await
                await streamWriter.WriteAsync(html);
            }
            MessageBox.Show("Finished downloading","Asynch Example");
        }
        public void DownloadHtml(string url)
        {   // NOT asynchronous! - just shown here for comparison
            var webClient = new WebClient();
            var html = webClient.DownloadString(url);

            using (var streamWriter = new
                                StreamWriter(@"c:\temp\result.html"))
            {
                streamWriter.Write(html);
            }
        }
    }
}
```

Now we will modify our program. The message box "Waiting…" executes **immediately**. We can execute other code here. Another message box executes **after** the blocking operation completes.

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    //await DownloadHtmlAsync("http://begincodingnow.com");
    // Note: if we use await we must use async in method definition.
    // var html = await GetHtmlAsync("http://begincodingnow.com");
    var getHtmlTask = GetHtmlAsync("http://begincodingnow.com");
    // executes immediately
    MessageBox.Show("Waiting for task to complete...");
    var html = await getHtmlTask;
    // executes after html is downloaded
    MessageBox.Show(html.Substring(0, 500));
}
public async Task<string> GetHtmlAsync(string url)
{
    var webClient = new WebClient();
    return await webClient.DownloadStringTaskAsync(url);
}
```

Exception Handling

We write exception handling code to avoid those Unhandled Exception messages when the program crashes. We can use a Try Catch block. The four keywords of exception handling are: try, catch, finally and throw. The first code example crashes with an unhandled exception. In the second example we handle the exception.

```
public class Calculator
{
    public int Divide(int numerator, int denominator)
    {
        return numerator / denominator;
    }
}
class Program
{
    static void Main(string[] args)
    {
        var calc = new Calculator();
        // Unhandled Exception: System.DivideByZeroException:
        // Attempted to divide by zero. CRASHES !!
        var result = calc.Divide(89, 0);
    }
}
```

Let's refactor our Main() method to use a try catch block.

```
static void Main(string[] args)
{
    try
    {
        var calc = new Calculator();
        var result = calc.Divide(89, 0);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Unexpected error!"); // F9, F5
    }
}
```

To implement multiple catch blocks set a break point (with F9) and run it in debug mode (F5). Place your cursor on "ex" and click the small right-arrow icon in the pop-up to bring up more details. Properties have the wrench icon. Look at Message, Source (the DLL or assembly), StackTrace (sequence of method calls in the reverse order – click the magnifier icon), TarketSite (method where exception happened) and the others.

```
static void Main(string[] args)
{
    try
    {
        var calc = new Calculator();
        var result = calc.Divide(89, 0);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Unexpected error! " +
            ex.Message);  // F9, F5
        // Unexpected error! Attempted to divide by zero.
    }
}
```

Multiple catch blocks example below.

```
static void Main(string[] args)
{
    try
    {
        var calc = new Calculator();
        var result = calc.Divide(89, 0);
        // type DivideByZeroException and F12
        // for Object Browser to see inheritance
        // hierarchy & click parent (bottom right)
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine("Cannot divide by zero. " + ex.Message);
    }
    catch (ArithmeticException ex)
    {
        Console.WriteLine("Arithmetic exception. " + ex.Message);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Unexpected error! " +
            ex.Message);  // F9, F5
        // Unexpected error! Attempted to divide by zero.
    }
    finally  // unmanaged resources are not handled by CLR
    {  } // we need to .Dispose() of those here, unless we employ
// the using statement.
}
class Program
{   // we need using System.IO;
    static void Main(string[] args)
    {
        try
        {   // using creates finally block in background
            using (var strmRdr = new StreamReader(@"c:\not.txt")) ;
        }
        catch (Exception ex)
        {
            Console.WriteLine("Unexpected error!");
        }
    }
}
```

One of the new features in C# 6 was exception filters, which are not covered here. They give you more control over your catch blocks and further tailor how you handle specific exceptions.

Recursion

Recursion happens when a method or function calls itself. We must write a condition that checks that the termination condition is satisfied. Below is a program that tells you how many times a number is evenly divisible by a divisor.

```
public static int CountDivisions(double number, double divisor)
{
    int count = 0;
    if (number > 0 && number % divisor == 0)
    {
        count++;
        number /= divisor;
        return count += CountDivisions(number, divisor);
    }
    return count;
}
static void Main(string[] args)
{
    Console.WriteLine("Enter your number: ");
    double number = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Enter your divisor: ");
    double divisor = Convert.ToDouble(Console.ReadLine());
    int count = CountDivisions(number, divisor);
    Console.WriteLine($"Total number of divisions: {count}");
    Console.ReadKey();
}
```