Section 4 Class Notes

**Errors**

Two different ways to handle errors:

|  |  |
| --- | --- |
| **Return Types** | **Errors** |
| Typically legacy type | First Class Citizen of the Language |
| -Check Return Value | .Net handles errors (not return types)\_ |
| -Response Accordingly | Must handle errors |
| -Query Info |  |

OpenFile(string) -> Pointer

If something goes wrong, the return type will tell you that. It will return null, for instance. ‘I cannot do this operation, but I’m not telling you why’. In a return type environment, you are expected to check the return type value and respond accordingly. Query Information.

So you would first catch the Return Value.

You then have to write code to react to an error return value.

If no error code is generated, then query the information to figure out what happened.

Var error = GetLastError()

{

Switch (Error)

{

Access.denied: do something; break;

FileNotFound: do something; break;

}

}

Errors:

First Class citizen of the language. An error type is returned by the language or platform. You don’t have to query in order to know what the error is.

Var file = OpenFile(“”);

If this fails, it will crash your program. We need to be able to handle errors.

Build out errors based on what occurred. For instance, if a database call fails… have your code try it again before sending in an error.

To generate errors in .Net, use ‘throw’.

Throw Expression of type T, where T is an exception, or derives from it. Exception is the base type for all errors.

throw Et

throw new Exception(); //notifies caller an error has occurred

As soon as a throw method executes, it always terminates the method.

Message – gives general message of what happened. Specifies why it went wrong.

Stack Trace – a call stack trace of where you were when error happened

InnerException – you can chain errors together

Type – what type of error occurred. Basically specifies what went wrong.

**Exception Types:**

**Exception** is the base type.

**Argument Exception** is the base exception for when the arguments (parameters to the functions) are wrong. The code is bad, not the user interaction. Allows you to return which argument (parameter) was wrong.

* ArgumentNullException – defacto exception you throw when what is received is null.
* ArgumentOutOfRangeException – exception you throw where data received is outside the bounds allowed.

**Invalid Operation Exception** – you are trying to do something that is not valid at the time. Ex. submitting order to Amazon without having paid for it. So what you are trying to do does not make sense.

Another example would be trying to send something to the database before connecting to the database.

**Not Implemented Exception** – typically when we are working on code and haven’t finished it.

**Format Exception**  - format of data is not valid.

Exception Types are done to tell the coder what went wrong.

If iValidator fails, return validationexception

Try/Catch – guard code. Inside the Try is the ‘unsafe code’. If any exception is raised by any code inside of the try block, the catch block executes.

Rethrow??

Quiz notes?

void Foo( MyType value)

1. If argument ‘value’ is null, throw an exception

If (value == null)

Throw new argumentNullException(nameof(value));

1. Handle argument exception when calling foo

Try

{

Foo(value);

}catch(exception e)

{

}

Void Log(exception)

1. Rethrow exception after logging

Log(e);

Throw;

**Files**

System IO

File

* Has methods for working with files

Directory

* Has methods for working with directories

(cannot create instances of these two types)

File Info – contains information about a file

DirectoryInfo – contains information about a directory

Streaming – read and/or write data as needed. Generally requires more code.

Type Stream – nothing more than a series of bytes. Or a stream of bytes.

First thing you have to do is get access to the stream. In the case of a file, there are a couple of Open methods.

Streams have to be cleaned up. You don’t want to wait for garbage collection to take care of it. You clean up a stream by using the Close method. Stream.Close()

This was all to show how streams work. We don’t work with streams. We work with stream partners. StreamReader and StreamWriter for text. BinaryReader and BinaryWriter are for binary.

IDisposable interface identifies a resource you need to clean up. It has no parameters. Returns void. When you see something that uses IDisposable, you should realize you don’t want to leave it without cleanup for any length of time.

A **Using** statement is a Try/finally. Needs the body of the code you need to execute. And it needs to know what disposable object that you need cleaned up. 90% of the time it will be a variable you declared.

Static classes – don’t have any properties. That could cause issues.

Static classes consist of static members. All members must be static. **No Constructors** with a static class. No instances! You can’t use ‘new’ with a static class.

Static classes are generally used for helper methods.

**Extension method**

Allow you to take any type (interface, enum, primitive, strings, etc.) and insert a new method into it. You are extending an existing type with new methods. Hence, extension method.

Extension Methods allow you to do this because you define what that method does. The difference between them and a normal method

1. Extension Methods only work with static classes.
2. The type you are extending must be the first parameter to that method.
3. Add ‘this’ keyword in front of first parameter. That one word changes it from a static method to an extension method.

Guidelines

Class name is class name + ‘Extensions’. Typically don’t put I in front.

This first parameter is always named ‘source’.

You do not worry about null with extension methods because you are not emulating what an instance method would actually do.

**Use Cases for Extension Methods**

Do NOT make all your static methods extension methods

No Type Access

Expose instance functionality

Generally useful or isolated

Do not extend primitives.

Extending Interfaces (number one reason for using extension methods)

**LINQ  
USED FOR QUERYING DATA! It is NOT used for modifying data!!**

Biggest reason for LINQs is to replace the foreach statement. It is more readable. Gives you a database type syntax without have to worry about how it works under the hood.

In order for LINQ to work, it must have a datasource, and it must be implement IEnumerable<T>.

LINQ is query based. So read-only data. You cannot modify data!

Start with what data are you selecting from.

So:

From [variable name for current row] from item in \_products

Select item select item;

The results of a LINQ query is ALWAYS IEnumerable<T>

LINQ uses deferred execution. It is efficient because of this. The entire linq is not run until results are asked for.

Where needs a Func<T, bool>

First – must have at least one value. Don’t ever use First.

FirstorDefault - First element in the result of IEnumberable<T>. If there are no elements, it returns the default for <T>.

SingleorDefault – Can ONLY be one element. If more than one, an exception is thrown. Never use.

LastorDefault – same as First, but gets the last in the list.

Lambda – anonymous method. They are extension methods.

Syntax – (args) => { }

Arguments are variable names. Types are implied. The curly braces define the body of the method.

You don’t need parentheses unless y ou have multiple or no parameters.

You only need curly braces if you have multiple statements.

Lambdas work anywhere you need a method and it only does one or two things.

Quiz Questions? Using LINQ

1. Given Students, final all students with fristname of Bob?

from student in Students

where student.FirstName == “Bob”

select Student

Note: this returns IEnumerable<Student>

1. Determine if there is a student named “Sue” “Miller”

(from student in Students

where student.FirstName == “Sue” && Student.LastName = “Miller”

select Student).FirstOrDefault() != null

Note: You could also use .Any() instead of .FirstOrDefault() != null

or lambda Students.Where( s => s.FirstName == “Sue” && s.LastName == “Miller”).Any()

1. Define extension method to return student’s full name

Public static class StudentExtensions{

Public static string FullName( this Student source)

{ Return source.FirstName + source.LastName; }

}

Anonymous types – allows you to create many types without having to typing them out. You always have to use object initializer syntax. They are always reference types. They can never use inheritance or be inherited from. Their products are always public. They cannot have methods or events. Biggest limitation…. They are only valid in the context where the new statement was given. You can’t pass them by parameters or returning them to functions.

Tuple<T1, T2>

Item1

Item2

(nothing more than a grouping concept)

Benefit – the type is predefined. But you cannot change the names Item1 and Item2

**Expression Body Member** (look this up)

Get => \_description = value?.Trim();

Getter only property with expression body

Public decimal CalculatedProperty => 0M;

//Field with expression body

Public decimal CalculatedProerty2 = 0M

Quiz Questions?

**Given Students, select first and Last Name**

From student in Students

Where !student.IsPassing

Select new { FirstName = student.FirstName, (note: this is anonymous type)

LastName = student.LastName}

Doesn’t matter what you are using, everything going down to ADO.NET when you are working with data access. The bulk of ADO.NET is in System.Data. Inside System.Data is SqlClient, OracleClient, ODBCClient, etc.

Have to have a Connection to data source. Connections implement IDisposable. Use a ‘using’ statement to clean up connections.

Command – an instruction to the database for data.

Connection Strings are generally:

Server = [name of server];

Database = [name of database];

Query returns recordsets

ExecuteReader – should be your default choice

FillDataSet

DML – insert, Update, Delete. Does not return recordsets… but returns a status or id, etc.

ExecuteNonQuery() –

ExecuteScalar()-

ExecuteReader – have to use Read in order to read rows. What you need to walk through rows, including the first row.

Lookup Heuristically

In order to use column names with reader:

reader["Id"]

problem is, reader does not know underlying type, so you have to do the conversion

So, Convert.ToInt32(reader[“Id”]};

If reader brings back a null column, you have to write code to handle it, otherwise it will throw an error.

\*Don’t use an iterator inside a using statement. It will hold up disposal.