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.