[0730] Let's Play Catch!

Whenever you think an exception might occur in your code, you need put that code inside of a "try/catch block" like this:

```
try {
   File file = new File("textfile.txt");
} catch (FileIOException e) {
  Console.WriteLine("Unable to create new file. Is your disk full?");
}
```

The trick is to make sure that you surround just the appropriate code that might cause an exception and not just e-v-e-r-y-t-h-i-n-g.

Lazy programmers will put a "try" statement at the top of their program, and a "catch (Exception e)" statement at the bottom. That ensures that they catch every exception that might even possibly happen inside their code. Great! Right? (right?) Nope... Not even close.

Catching exceptions with a huge "catch-all" block like this is worse than letting the exceptions fall thru. The problem is your code doesn't know what exceptions happened when and why. Without that context, all your code can reasonably do is log the error somewhere and hope a human can figure it out later. 99 times out of 100, the lazy programmers that do this simply ignore the exception entirely - which is a terrible thing to do.

When Should You Worry About Exceptions?

By their very nature, Exceptions should be exceptional - as in "rare" or "happening in unusual circumstances." Indeed, one of the great things about Exceptions is that they don't usually junk up your code. They "hide in the shadows" until you need them, letting your normal "mainline" code remain clear and readable. #yay

There are two broad categories of exceptions - System exceptions and Application exceptions.

*Note: These two categories of exceptions are **conceptual** in nature. Microsoft screwed things up when defining C# exceptions and now exceptions like "ArgumentOutOfRangeException" are technically "System.Exception" exceptions in the C# inheritance hierarchy instead of "ApplicationExceptions" . #dumb*

While, in theory, System exceptions can happen at any moment in a program's lifespan, most of those exceptions - like OutOfMemoryException for example - are connected to problems that you - the application - can do nothing about. There is almost no way to recover gracefully if the system is truly out of memory and so most programs do not even try.

What's much more common are system exceptions related to I/O - input/output operations - places where your program uses a device inside your computer. Disk files and Network systems are the main

culprits here. Unfortunately, those things are as reliable as we'd like and there are hundreds of reasons why disk I/O and network I/O can fail. Programmers almost always need to surround code that accesses either of those two things with exception handlers.

Whenever your program is interacting with a different program, module or subsystem that you don't fully control, be alert for System Exception situations.

Application exceptions are generally less dire than System exceptions although, if they are not handled properly, your program will still crash. Application exceptions (also called "runtime" exceptions) occur when your program gets into a situation it should never get into. For instance, applications should never be asked to divide by zero. They should never be asked to de-reference a null pointer. They should never be asked to store data past the end of an array. All of those situations - and many more - result in a run-time exception being thrown.

Unlike most System exceptions, some Application exceptions can be recovered from without alerting the user. Most, however, are just good old-fashioned bugs caused by programmers that didn't test their code thoroughly.

OK, I Caught One. Now What?

After catching an exception, you have a couple of choices:

- You can try to recover from it
- You can re-throw it
- You can ignore it

Which one you pick depends heavily on your program, its run-time environment and the severity of the exception involved.

If you think you can recover from the exception, you can try to do that inside the catch block. Sometimes things can be fixed without the user knowing. Other times, you may need to get the user involved by putting up an error message or asking for more input. Regardless, it is important to make sure that any code you put in a catch block works reliably no matter what. If the code in a catch block causes an exception, things get really weird really quickly!

Sometimes, mainly in reusable libraries, your code will catch an exception that needs to be converted into a different exception due to context. For example, let's say that you are creating a reusable Stack class using an array. There might be times were your code catches an ArrayIndexOutOfBoundsException - maybe because the user added too many things to the stack. The problem is that the user thinks they are working with a stack. They know nothing about the array you are using to implement that stack. If they get an "ArrayOutOfBoundsException", they will be very confused. In cases like this, instead of letting the confusing exception through, you can "re-throw" it as a "StackOverflowException" which makes much more sense in this context.

Finally, you can choose to ignore any exception you catch by just letting the code fall through the catch block. This is not recommended because it means that no one will know that the exception happened.

In theory, exceptions are *e-x-c-e-p-t-i-o-n-a-l* and therefore they should not be ignored.