EXCEPTIONS & EXCEPTION HANDLING

HOW CAN ERRORS BE HANDLED?

Errors in Your Code

- Three major types of errors
- Syntax
  - *A problem with the code itself*
  - *Prevents your program from compiling/running*
- Run-Time
  - *A problem that occurs as the program is running*
  - *Usually "crashes" your program*
- Logic
  - *Program runs but produces unexpected results*

Handling Syntax Errors

Syntax errors:

- *A problem with the code itself*
- *Prevents your program from compiling/running*

What does ***Visual Studio*** do?

- *Visual Studio helps with syntax errors*
- *Gives us useful error messages*
- *Uses Intellisense (red squiggly lines)*

What do ***WE*** need to do?

- *Correct spelling*
- *Check for re-declaration of variables*
- *Call methods that exist*
- *etc.*

Handling Run-Time Errors

Run-Time errors:

- *A problem that occurs as the program is running*
- *Usually "crashes" your program*

What does ***Visual Studio*** do?

- *Debugger system (breakpoints, stepping through, F5)*
- *Breakpoints and stepping through code help us to "see" the path our code takes*
- *Can help us to understand the point which a program crashes*

Handling Run-Time Errors

What do ***WE*** need to do?

Attempt to prevent the errors

1. Check variable values before use
   - *Use TryParse()*
   - *Use conditionals to allow certain code to occur if a value is ok*

2. Force the user to re-enter invalid input
   - *If the error is related to user input*
   - *"Enter a number between 0 – 10"*

3. Let the program crash
   - *Probably not a great idea*

Unavoidable Errors

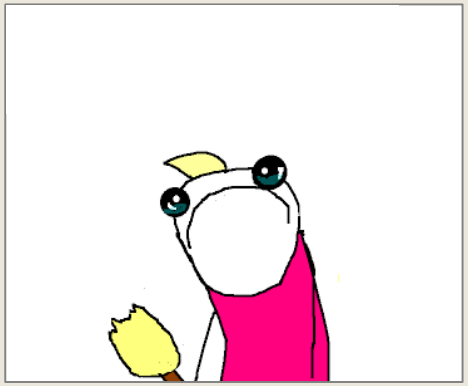
You should **prevent** errors as much as possible!

- ***Do all the things!!***
- *Use TryParse().*
- *Check variable values.*
- *Use conditionals to ensure valid data*
- *Require user to re-enter data.*
- *Use Contains() to determine if data is inside list.*
- *Check for null values.*

But…

Sometimes you just can't prevent an error.

- *Maybe you don't know if the error will occur*
- *Maybe the error is out of your control*



EXCEPTIONS

Exceptions

All C# run-time errors are **exceptions**

- *You've seen this before in your coding…*
- *NullPointer Exception*
- *KeyNotFound Exception*
- *IndexOutOfRange Exception*
- *InvalidOperation Exception*

What does it actually *mean*?

Exceptions

Exceptions are technically *objects.*

- *Created automatically by C# when things go wrong*
- *You can also create them if necessary*

Exceptions represent errors that occur *while your program is running.*

Your code can interact with these exceptions.

Helpful for developers, not for users

How Exceptions Are Created

1. C# detects something went wrong – cannot perform requested action

2. It creates an exception object
   - *Instantiated like any other object*
   - *Its properties are set*
   - *(This happens behind the scenes)*

3. The exception is then *thrown*
   - *How the rest of the program is notified*
   - *"This exception object exists! Something happened!"*

4. If not handled properly, the program **will** crash
   - *"An exception! Abort! Ship's going down!"*

Exception Objects

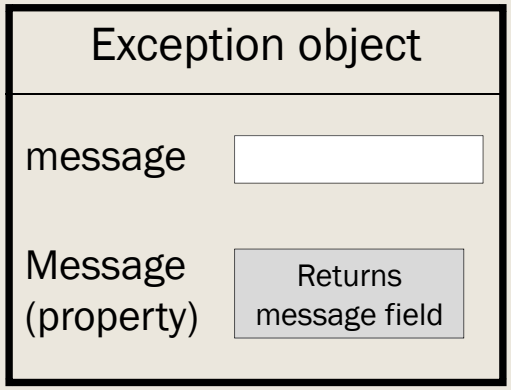
Similar to other objects you've used

Contain fields, methods and properties

- *Like .Count in the List class*

Example of a useful property is `.Message`

- *Contains the actual error message*
- *The same one you see when the program crashes*
- *Display this in a C.WL() to get detailed information about the Exception object*



Exception objects are created by Visual Studio when something goes wrong.

Why?

- The Exception is a way that one part of your code can communicate to another.

Hey, line of code that's calling a statement that caused an exception...

What you're asking for cannot be accomplished!

Do something else instead!

What kind of communication?

```
List<int> myNumbers = new List<int>();
myNumbers.Add(5);
myNumbers.Add(10);
myNumbers.Add(15);

int valueInElement24 = myNumbers[24];
```

The List class is communicating with Main that what we're asking for (get index 24 in a list of only 3 elements) cannot be done!

It does this by causing an exception.

It's OUR job to decide what to do with it.

Detecting exception communication

- What if I told you we can stop a crash from happening?!?

- Can write code to detect that an exception was made, and then
- React to them however we want!
  - *Run alternate code instead*

- If we properly handle the exception…
- Our program never actually crashes!

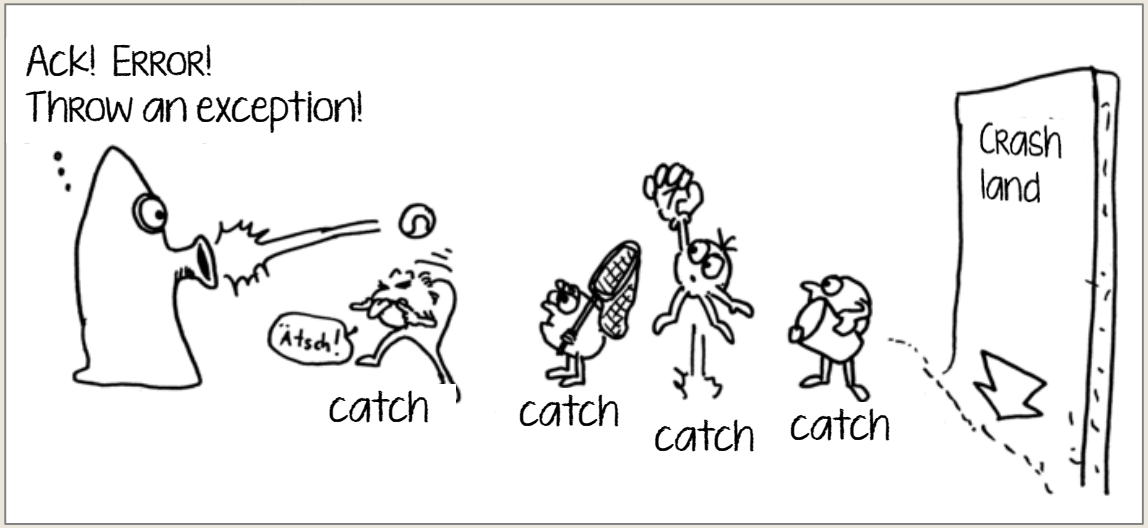
TRY AND CATCH

Catching Exceptions

Exceptions that are *thrown* must be *caught*

- *If not, they will cause your program to crash*

A special code block will catch exceptions: The **try/catch** block



Try/Catch Example

```
try
{
    CodeThatMightThrowExceptions();
    //If the code in here causes an exception,
      it will NOT crash the program
}
```

```
catch (Exception myExceptionVariable)
{
    //Instead, the code in here will run
    DoSomethingUsefulInstead();
}
```

Try/Catch – Deeper Look

No parentheses

```
try
{
    CodeThatMightThrowExceptions();
    //If the code in here causes an exception,
      it will NOT crash the program
}
```

Declare a new Exception object, kind of like a parameter

```
catch (Exception myExceptionVariable)
{
    //Instead, the code in here will run
    DoSomethingUsefulInstead();
}
```

Try/Catch “Rules”

To catch an exception:

- *It **must** be thrown inside a try block*
- *Exceptions occurring outside a try cannot be caught*

Once the exception occurs:

- *The rest of the code in the try is **skipped***
- *The program immediately jumps to the catch block*

Another Example

```
try
{
      int x = 10;
      int y = 0;
      int result = x / y;  //Dividing by zero causes an error!

      Console.WriteLine("This will NOT show up!");
}
catch (Exception ex)
{
      //This will write the exception's actual message
         to the user
      Console.WriteLine(ex.Message);
}
```

Scope Issues

Remember that { }'s act like boundaries

Variables declared in the try block cease to exist outside

They will NOT be available in the catch

Scope Issue - Example

```
try
{
    int result = x / y;
}
catch(Exception e)
{
    result = -1;
    // Problem: Variable ‘result’ does NOT exist here!
}
```

May give a divide-by-zero error

Set a default result value if things go wrong

Fixed Scope Issue - Example

```
int result = 0;

try
{
    result = x / y;
}
catch(Exception e)
{
   result = -1;
}
```

Define variable outside of the blocks

Don't re-define variable

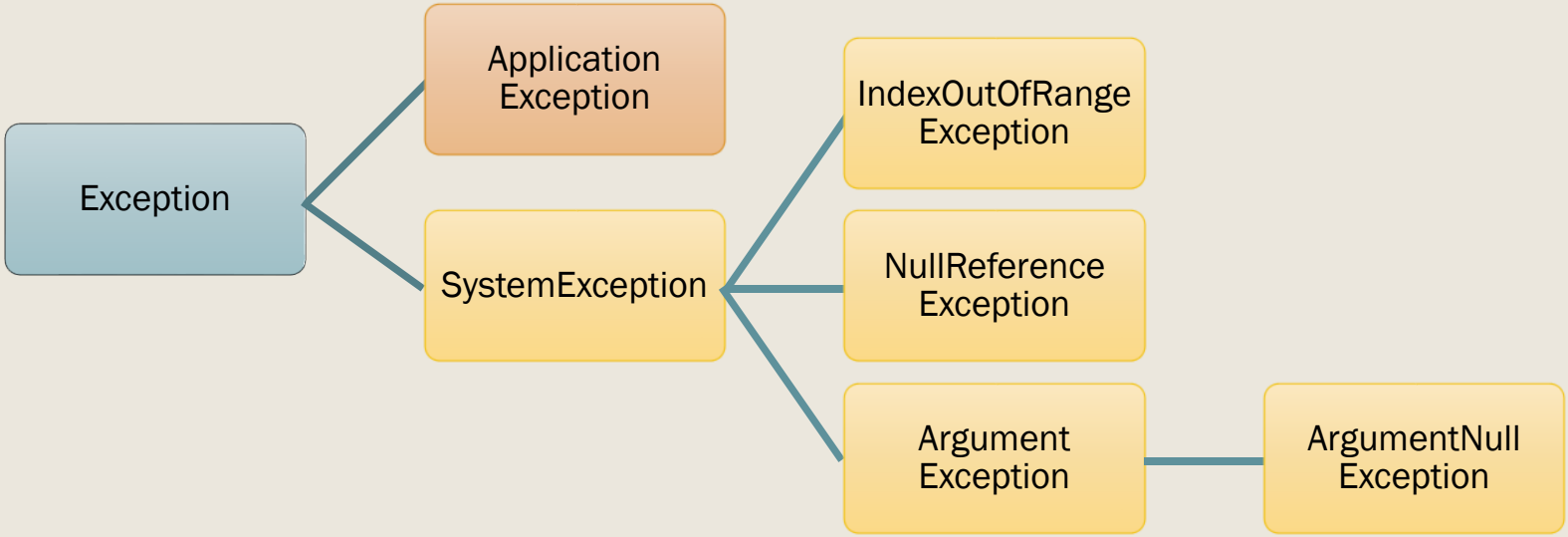
Set default value if there's an error

CATCHING MULTIPLE EXCEPTIONS

Catching different exceptions

- There are many different exception classes
  - *Ranging from general to specific*
  - *Example: DivideByZeroException*
- All exception classes inherit from “Exception”
  - *Base class for all exceptions*
- We can catch any type of exception

Exception hierarchy



The exceptions we're working with will come from the `SystemException` "branch" (yellow),
not the `ApplicationException` branch (orange).

Multiple exceptions

- Q: What if a method can throw multiple exceptions?
- A: They will all be caught by the single catch, regardless of exception type.

- Q: What if we want to handle them differently?
- A: Use multiple catch blocks and specify a different exception type in each block

Multiple exceptions - Example

```
try
{
   // Code that could throw multiple exceptions
}
catch(IndexOutOfRangeException e)
{
   // Handle only index out of range exceptions here
}
catch(SomeOtherException e)
{
   // Handle a different kind of exception here
   // Obviously SomeOtherException is not a valid C#
   //   exception that exists. ☺
}
```

Are these equivalent? (Exception order)

```
try
{
}
catch(Exception e)
{
}
catch(ArgumentException e)
{
}
```

```
try
{
}
catch(ArgumentException e)
{
}
catch(Exception e)
{
}
```

No. In fact, the left one won't even compile!
Exceptions must be caught in a particular order: Most-specific to most-general

If the most general exception comes first:

- The first catch block will capture all exceptions
- C# considers that a syntax error

Exception order

- Exceptions must be caught in a particular order
- Most-specific to most-general
- If the most general exception comes first
  - *The first catch block will capture all exceptions*
  - *C# considers that a syntax error*

GUIDELINES TO EXCEPTIONS

One Last Thing

Remember: **Preventing** exceptions is FAR better than catching exceptions

A simple if statement is **much** faster than:

- *C# detecting an error*
- *And generating an object*
- *And throwing the exception*
- *And catching it*

Use try/catch where necessary

- *It's not a replacement for basic error checking!*

So… When should you catch exceptions?

When you cannot prevent them from happening!

You CAN prevent:

- Index out of range
- Bad user data
- Calling the wrong method on the wrong object
- Incompatible type casting
- Calling a method or accessing a property on a null reference

You CANNOT prevent:

- Invalid file access

THROWING YOUR OWN EXCEPTIONS

Can I throw an exception too?

- We have the power to make Exceptions happen!
- Imagine you are writing a List class…
  And someone wants to retrieve index 24 in a list with a count of 2…
- What would your code do?
  - *Allow the crash to happen? No – that's a bad interface.*
  - *Return a default value (0 for ints, false for bools, etc.) so that the user thinks that 0 is at index 24? No – that's misleading.*
- Just like the built-in C# list class does, we can force the program to throw an exception!

Throwing exceptions?

- You should throw your own exceptions when…
- **When your code can't actually complete the specified task!**

- For instance:
  - *Your method accepts the name of a player to return, but that player doesn't exist. You can't actually return a player – so throw an Exception!*

- Why not just return null?
  - *Null means there is no pointer, but*
  - *The exception explicitly means "I cannot do what you want me to"*

Using exceptions

- What does an exception really represent?
- Something going wrong
- *So wrong* that the current method can't operate

- **Exceptions are a way to notify a *different* part of your code (another method or class) when errors occur**

Throwing your own exceptions

- C# has a *throw* keyword
- Your job is to: Create an exception object, then throw it yourself
- The current method will immediately end

```
Exception ex = new Exception("Error message");
throw ex;
```

- One line version:

```
throw new Exception("Error message");
```

Throwing - Example

```
// Asks for the name of a Player, then returns the Player
object with a matching Name property
public Player GetPlayerByName(string name)
{
    foreach(Player currentPlayer in playerList)
    {
        if(currentPlayer.Name == name )
            return currentPlayer;
    }

    // If we got here, the player doesn't exist!
    // We can't actually carry out the task.
    // (Notice there is no return statement here)
    throw new Exception("Player not found!");
}
```

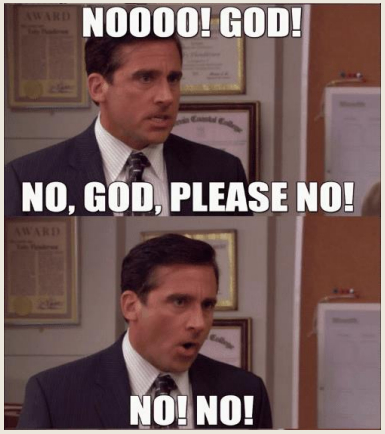
Throwing vs. return

- In the previous example, there's no return at the very end of the method
- Throws can be used IN PLACE of a return statement! They BOTH immediately end the method.
- Throwing is a valid way to end a method, even if the method requires a return value
  - *It immediately ends the method*
  - *The rest of the code will not run*

Where do I catch my throws?

- NOT INSIDE THE SAME METHOD WHERE YOU THROW THEM!

```
public static void MethodThatCouldThrowAnException(bool
problem)
{
    try
    {
          if( problem == true )
          {
              throw new Exception();
          }
    }
    catch(Exception e)
    {
          Console.WriteLine("There was a problem!");
    }
}
```



Where do I catch my throws?

- Do this instead:

```
public static void MethodThatCouldThrowAnException(bool
problem)
{
          if( problem == true )
          {
              throw new Exception();
          }
}
```

Catch this OUTSIDE of this method!
Catch it wherever you are calling
MethodThatCouldThrowAnException()!

Where do I catch my throws?

- Catch the exception wherever you are CALLING the method that throws the exception!

```
public static void Main(string[] args)
{
    try                                          YES YES YES
    {                                            YES YES YES
       MethodThatCouldThrowAnException();        YES YES YES
    }                                            YES YES YES
    catch(Exception e)
    {
       Console.WriteLine("There was a problem!");
    }
}
```

So where should I catch?

- If you're throwing an exception…
- Generally means the method CAN'T go on
  - *Things have gone wrong! Abort mission!*
- If the method CAN go on…
  - *It's not an exceptional circumstance*
  - *Don't throw an exception!*
  - *Use other error handling techniques*

More specific exceptions

- C# comes with many exception classes
  - *IndexOutOfRangeException*
  - *DivideByZeroException*
  - *NullReferenceException*
  - *FormatException*
  - *And more*
- Most are used by C# for specific errors
  - *They inherit from SystemException, which inherits from Exception*
- They might not make sense for all errors
  - *Especially for errors in your own methods*

One last time…
Where Should I Catch? Example

Try/Catch here

```
static void Main(...)
{
  try
  {
    DoStuff(10);
    DoStuff(-2);
  }
  catch(Exception e)
  {
    // We catch here!
  }
}
```

Throw here, don't catch

```
void DoStuff(int value)
{
  // Check for invalid
  // parameter data
  if( value < 0 )
    throw new Exception();

  // Data is valid - do
  // something useful
  // here.
  ...
}
```