When and How to use Exceptions

Basics of Exceptions

In programming, *exceptions* are errors your programs *throw* in response to circumstances that are not intended and require special processing. You use the special `throw` command to interrupt the normal execution of the program when error ocurrs. This is donewhen an error has ocurred and needs to be handled by your program. In previous lessons, you've seen places where code you are calling throws an exception. One exception that you may have tried already happens when you typed in anything that wasn't a number during the guessing game. Doing this would result in `Unhandled Exception: System.FormatException: Input string was not in a correct format.` being displayed and your program would crash.

```
public static void Main()
{
    try
    {
        int sum = SumNumberStrings(new List<string> {"5", "4"});
        Console.WriteLine(sum);
    }
    catch (System.FormatException)
    {
        Console.WriteLine("List of numbers contained an invalid entry.");
    }
}

public static int SumNumberStrings(List<string> numbers)
{
    int total = 0;
    foreach (string numberString in numbers)
    {
        total += int.Parse(numberString);
    }
    return total;
}
```

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main()
    {
        try
        {
            // To see the error, try changing to or adding invalid numbers
            int sum = SumNumberStrings(new List<string> {"5", "4"});
            Console.WriteLine(sum);
        }
        catch (System.FormatException)
        {
            Console.WriteLine("List of numbers contained an invalid entry.");
        }
    }

    public static int SumNumberStrings(List<string> numbers)
    {
        int total = 0;
        foreach (string numberString in numbers)
        {
            total += int.Parse(numberString);
        }
        return total;
    }
}
```

Catching Exceptions

When code you write has the potential to throw an exception, it's important that the exception be caught and handled by something. You do this, by using a *try-catch* block in your code. This is an empty example of one:

```
try
{
    // code that may throw an exception here
}
catch (System.Exception ex)
{
    // handle the exception here
}
```

The first block of code, called a *try* block, is where you put all of your code that may throw an exception. If an exception is thrown in the *try* block, your program will immediately jump into your *catch* block. It's at this point that you are able to handle the exception. You should also notice the `(System.Exception ex)` variable declaration, which gives you access to the details of the exception during the catch block by using the `ex` variable.

When an exception ocurrs, your code will need to decide if it should continue running or whether it has no solution to the issue. If you have a solution to the issue, your code can correct the state of things to allow your program to continue. If your code cannot resolve the error, you should re-throw the exception. To do this, you use the `throw` statement on its own line without giving it an exception, like in this example:

```
try
{
    int x = int.Parse(Console.ReadLine());
}
catch (System.FormatException ex)
{
    // do something here before re-throwing
    throw;
}
```

> **Tip** {.tip .newLanguage}
> If your code isn't going to respond to or log the exception in any way, don't catch the exception in the first place.

Throwing a new exception will replace the previous exception, so only do this if you're using an exception specific to your application (and make sure to set the `InnerException` property of your new exception to the previous exception.

> **Tip** {.tip .newLanguage}
> It is common to log exceptions that occur, so that you can review them later and improve the program to avoid them, if possible.

Throwing Exceptions

When you're in one of those unexpected situations and your code cannot (or should not) handle this situation itself, it's time to throw an exception.

```
public List<People> GetAllCustomersByAge(int age)
{
    if (age < 18 || age > 150)
    {
        throw new ArgumentOutOfRangeException("Age must be between 18 and 150.", nameof(age)
    }
}
```

When throwing exceptions, you use the `throw` keyword to indicate that you're throwing an exception followed by the `Exception` object you're throwing. In the example above, the exception object is being created on the same line on which it's thrown. Also notice that a specific type was used here, the `ArgumentOutOfRangeException`. This means that anyone catching the exception knows that an argument received was not within the expected range. Some exception types, like the `ArgumentOutOfRangeException`, take additional arguments in their constructors. For the various exceptions that derive from the `System.ArgumentException`, like this example, the constructor takes an error message and the name of the argument containing the invalid value.

> **Note** {.note}
> Never throw `System.Exception`, `System.SystemException`, or `ApplicationException` directly; use more specific standard exceptions or custom exceptions. This allows callers to choose which exceptions they can handle instead of needing to respond to all exceptions.

The Finally Block

In your code, you may need to ensure you've cleaned up and released any resources you had allocated, whether you threw an exception or not. In these circumstances, the *finally* block is what you're looking for.

```
try
{
}
catch (System.Exception ex)
{
}
finally
{
    // This code will always run
    // even if your catch block re-throws
}
// Code here will run only if catch doesn't re-throw
```

```
using System;

public class Program
{
    public static void Main()
    {
        try
        {
            throw new Exception("Let's play catch!");
        }
        catch (System.Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // This code will always run
            // even if your catch block re-throws
            Console.WriteLine("FINALLY!");
        }
        // Code here will run only if catch doesn't re-throw
        Console.WriteLine("Still ran.");
    }
}
```

Most people ask why they need to use a finally block, since the code after the try-catch runs anyway. If the catch block handles the exception and allows your code to keep running, it will be functionally equivalent. If the catch block re-throws or throws a new exception (intentionally or not), your finally will still run. This means you can depend on the finally block executing, but be careful to not throw an exception in the finally block.

Additionally, once your program reaches the *try* block, the *finally* block will execute before returning, even if your method's *return* statement is inside the *try* block or the *catch* block.

It is also possible to use a *try-finally* block if you want the guarantee, but aren't going to be handling the exception.

```
try
{
}
finally
{
    // This code will always run - even after exceptions
}
// Code here will run only if no exceptions are thrown
```

Note {.note}
In most cases, resources that need to be cleaned up will use .NET's `IDisposable` interface, and can be wrapped in a using block.

Throwing From Catch Blocks

Your catch blocks can include any code statements, including code that directly throws an exception or calls to other methods that could throw exceptions. If an exception is thrown from inside of your catch block, the details of the original exception will be lost, so it's important to use caution. If you want to catch a low-level exception and throw one more relevant to your program, make sure to assign the low-level one to the `InnerException` property of the new exception. This will keep the information available should an investigation be necessary.

```
try
{
}
catch (ArgumentNullException ex)
{
    throw new UserRequiredException(ex); // InnerException property set by constructor
}
```

Most exception classes will take another exception as a constructor parameter allowing you to pass in the InnerException. In order to avoid losing the information, the InnerException property of exceptions is read-only, once set by the constructor.

If you're not handling the exceptions at the time that you've caught it, you generally want to just rethrow it like this:

```
try
{
}
catch (ArgumentNullException ex)
{
    Logger.LogError(ex); // log the error before rethrowing
    throw;
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        try
        {
            DoWork();
        }
        catch (System.Exception ex)
        {
            Console.WriteLine("We encountered an error. Please try again.");
            Console.WriteLine(ex.Message);
        }
    }

    public static void DoWork()
    {
        try
        {
            FindStudentId(null);
        }
        catch (ArgumentNullException ex)
        {
            // You would do some safe clean up work or logging here
            // Logger.LogError(ex);
            throw;
        }
    }

    public static int FindStudentId(string studentName)
    {
        if (string.IsNullOrEmpty(studentName))
        {
            throw new ArgumentNullException(nameof(studentName));
        }

        return 0; // we didn't really implement this
    }
}
```

Taking this approach will preserve the previous exception, so very little is lost by logging what happened.

Catching Specific Exceptions

In the previous examples, you usually saw `catch (System.Exception)`, which uses polymorphism, the programming concept you learned about from the lesson on Objects and Classes. Knowing that, you also know that you can pass any exception that inherits from `System.Exception`, and it can be used there as well. If you want to be more specific you can be.
Our `System.FormatException` did exactly that.

When you catch a specific exception, it will only catch the exceptions of that specific type (or ones that inherit from it). That means that catching `System.FormatException` will not catch any `System.Exception` exceptions that are thrown. If you want to handle both, differently, you would do this:

```
try
{
}
catch (System.FormatException)
{
    // Handle only FormatExceptions here
}
catch (System.Exception)
{
    // Handle all other exceptions here
}
```

```
using System;

public class Program
{
    public static void Main()
    {
        try
        {
            // uncomment either line to see an error
            //int.Parse("A");
            //throw new Exception();
        }
        catch (System.FormatException)
        {
            Console.WriteLine("This is a System.FormatException");
        }
        catch (System.Exception)
        {
            Console.WriteLine("This is a different System.Exception");
        }
    }
}
```

Note {.note}
The order of the catch blocks is important; only one catch block can catch an exception. The first catch block that can handle the exception will execute; any others are ignored.

In some instances you may want to handle more than one type of specific exception the same way and their common exception ancestor class is only `System.Exception`. In order to that, you use the `when` clause followed by a condition explaining when you want to use that catch block. It looks like this:

```
try
{
}
catch (Exception ex) when (ex is MemberNameNotFoundException || ex is FormatException)
{
    // Handle only MemberNameNotFoundException and FormatException here
}
```

By restricting in this way, you're able to reuse more code, while ignoring exception types you're not prepared to handle.

When to Avoid Throwing and Catching

It is important that you only throw exceptions when it's appropriate, when your code is in an unintended situation. If the situation seems likely, but requires taking a certain action, just handle it using normal application flow control elements like `if` statements.

A rule of thumb for determining if you need to throw an exception is if you encounter a situation in a method that doesn't allow you to return a valid result. In that case, an exception may be the best solution, or you may need to reconsider what your method is returning. The following two examples demonstrate two ways to deal with a method failing to perform its intended task:

```
public void SetMemberBirthday(int memberId, DateTime birthday)
{
    Member member = _memberList.SingleOrDefault(m => m.Id == memberId);
    if (member == null)
    {
        throw new MemberNotFoundException(id);
    }
    member.Birthday = birthday;
}
```

```
public bool SetMemberBirthday(int memberId, DateTime birthday)
{
    Member member = _memberList.SingleOrDefault(m => m.Id == memberId);
    if (member == null)
    {
        Logger.LogWarning($"SetMemberBirthday Error: Member {memberId} not found. Birthday n
        return false; // false tells the caller that the operation failed.
    }
    member.Birthday = birthday;
    return true;
}
```

Tip {.tip .newLanguage}
When you throw an exception, there is always the chance that the exception will go unhandled, so be careful to throw exceptions only when it feels appropriate.

Creating Your Own Exceptions

When you are going to throw an exception in your code, it's important that you use the correct exception. This makes it easier for you, and others who call your code, to catch the specific exception that you're throwing.

In order to create your own exceptions, you simply inherit from the `Exception` class or from a more-specific exception class like this:

```
public class MemberNameNotFoundException : Exception
{
    public MemberNameNotFoundException(string memberName)
        : base($"Could not find member: {memberName}.")
    {}
}
```

Once you have that exception, you're able to throw it uisng the same `throw` command you use for any other exceptions like this:

```
public int GetMemberIdByName(string memberName)
{
    Member member = _memberList.SingleOrDefault(m => m.Name == memberName);
    if (member != null)
    {
        return member.Id;
    }
    throw new MemberNameNotFoundException(memberName);
}
```

You throw your exception here, because the code is in an exceptional case. You have no `Id` to return from your method, so you throw an exception that the caller can catch and handle. The caller of this method, might handle the exception like this:

```
try
{
    int memberId = membersOnlyList.GetMemberIdByName("George");
    Console.WriteLine($"Come on in member, {memberId}");
}
catch (MemberNameNotFoundException)
{
    Console.WriteLine($"You're not on the list.");
}
```

In some cases, you may want to wrap low level exceptions with your own exceptions that are more specific to your application's model. In that case, you probably will want to retain the information from the original exception. In that case, be sure to include a constructor in your custom exception type that accepts an exception argument and sets the InnerException property. It would look like this:

```
public class OrderCheckoutException : System.Exception
{
    public OrderCheckoutException(System.Exception innerException)
        : base("There was an error while checking out.", innerException)
    {
    }
}
```

When catching the exception, you would wrap the exception like this:

```
try
{
    throw new Exception("This is the inner exception");
}
catch (System.Exception ex)
{
    Logger.LogError(ex);
    throw new OrderCheckoutException(ex);
}
```