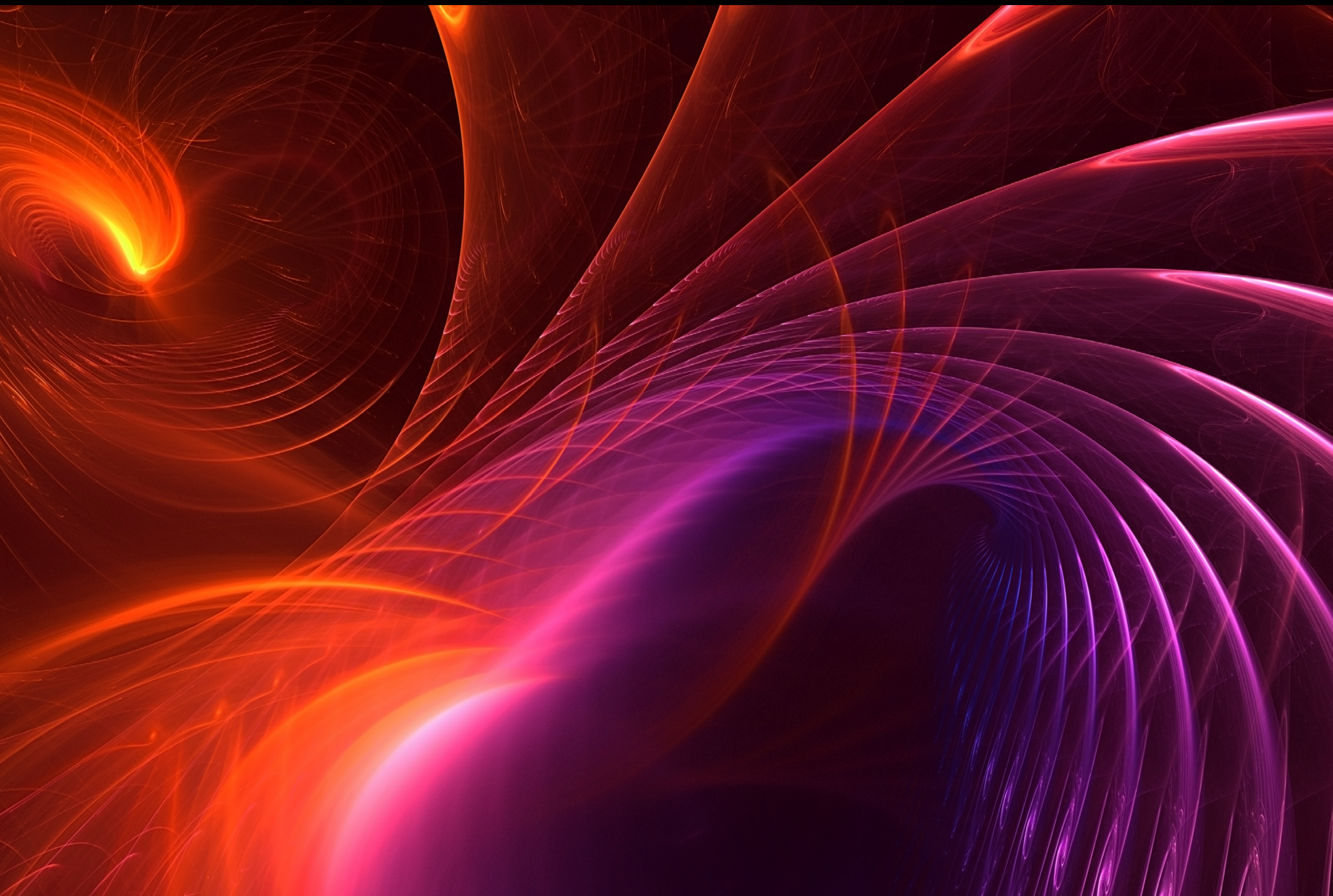
C# 13 Programming Essentials



.NET 9 Edition

Payload publishing

C# 13 Programming Essentials

.NET 9 Edition

C# 13 Programming Essentials - .NET 9 Edition

ISBN-13: 978-1-965764-01-5

© 2024 Neil Smyth / Payload Media, Inc. All Rights Reserved.

This book is provided for personal use only. Unauthorized use, reproduction and/or distribution strictly prohibited. All rights reserved.

The content of this book is provided for informational purposes only. Neither the publisher nor the author offers any warranties or representation, express or implied, with regard to the accuracy of information contained in this book, nor do they accept any liability for any loss or damage arising from any errors or omissions.

This book contains trademarked terms that are used solely for editorial purposes and to the benefit of the respective trademark owner. The terms used within this book are not intended as infringement of any trademarks.

Rev: 1.0

https://www.payloadbooks.com

Chapter 23

23. C# Exception Handling

In a perfect world, C# code would never encounter an error. Since we don't live in a perfect world, it's impossible to guarantee that an error (more commonly referred to as an *exception*) won't occur while your code is executing. Therefore, it's crucial to ensure the code gracefully handles any exceptions that may arise.

This chapter covers exception handling using C#, introducing topics like exception types, how exceptions are thrown, and how to handle them using *try-catch* statements.

23.1 Understanding exception handling

No matter how meticulously your C# code is designed and implemented, there will always be situations beyond the app's control. For instance, an app relying on an active network connection can't control the loss of that connection. What the app can do, however, is implement robust exception handling, like displaying a message to the user, or silently attempting to re-establish the lost network connection.

Exception handling in C# involves two main aspects: triggering (or throwing) an exception when desired results aren't achieved within a method and catching and handling the exception after it's thrown by another method.

When an exception is thrown, it has a specific exception type that can be used to identify the nature of the exception and decide on the most appropriate course of action. When throwing exceptions in your code, you can use the built-in exception types, or create and throw custom exception types derived from the System Exception class.

While learning about exception handling in C#, remember that in addition to implementing code in your app to throw exceptions when necessary, many API methods in the C# and .NET frameworks also throw exceptions that must be handled within your app's code. Failure to do so typically results in a runtime exception and your app crashing.

23.2 Creating the ExceptionDemo project

Launch VS Code, open your "C# Essentials" workspace folder, and create a new .NET Console App project named ExceptionDemo. Once the project is ready, modify the *Program.cs* file as follows:

```
namespace ExceptionDemo
{
    internal class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

The project is a hypothetical app that controls an electric motor. During the power-on diagnostics, the app reads the current voltage and prevents the motor from starting if it exceeds 500 volts. Remaining in the *Program.cs* file, add a method to perform the voltage check and call it from the Main() method:

```
namespace ExceptionDemo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            VoltageCheck(501);
        }

        private static void VoltageCheck(int voltage)
        {
            if (voltage > 500)
            {
                Console.WriteLine("Voltage overload.");
            }
            else
            {
                Console.WriteLine("Voltage is safe.");
            }
        }
    }
}
```

Run the app and check that the voltage overload message appears in the console:

```
Voltage overload.
```

A voltage overload is a serious issue, so we want to do more than display a message and hope the user doesn't just start the motor. Instead, we want to force the app to take preventative measures. To do this, we will throw an exception.

23.3 Throwing exceptions

The first step in throwing an exception is to choose an exception type. As we mentioned previously, exceptions can be thrown using one of the many built-in types or a custom type you have created. We will cover custom exception types later in the chapter, so for now, we will use the ArgumentOutOfRangeException type. This exception type is typically thrown when an argument passed to a method is outside an acceptable range (in our case a voltage reading exceeding 500 volts).

Next, an instance of the exception type is created and initialized. All exception types must be initialized with a message explaining the cause of the problem. Some types, including ArgumentOutOfRangeException, also require the name of the parameter that caused the exception (in our case, the voltage). With this information, we can create a new exception as follows:

```
ArgumentOutOfRangeException voltageException = new(nameof(voltage), "Voltage
exceeds safety parameters.");
```

Exceptions are thrown using the *throw* statement followed by the exception object, as follows:

```
throw voltageException;
```

In the above examples, we have created and thrown the exception as separate steps. This sequence is, however, typically performed in a single operation:

```
throw new ArgumentOutOfRangeException(nameof(voltage), "Voltage exceeds safety parameters.");
```

Modify the VoltageCheck() method to throw an exception as follows:

```
private static void VoltageCheck(int voltage)
{
    if (voltage > 500)
    {
        throw new ArgumentOutOfRangeException(nameof(voltage),
                                "Voltage exceeds safety parameters.");
    }
    else
    {
        Console.WriteLine("Safe Voltage");
    }
}
```

Refer to the console, where the following output will appear:

```
Unhandled exception. System.ArgumentOutOfRangeException: Voltage exceeds safety parameters. (Parameter 'voltage')
   at ExceptionDemo.Program.VoltageCheck(Int32 voltage) in Program.cs:line 16
   at ExceptionDemo.Program.Main(String[] args) in Program.cs:line 8
```

This message tells us the app was terminated due to an unhandled exception. The error includes the exception type, the message, and the name of the parameter that caused the exception.

We now need to add additional code to handle the exception when it is thrown.

23.4 Handling exceptions

Exceptions in C# are handled using *try-catch* statements. To prevent our app from crashing when the exception is thrown, we must make the following changes to the Main() method:

```
static void Main(string[] args)
{
    try
    {
        VoltageCheck(501);
    }
    catch
    {
        Console.WriteLine("Voltage too high - disconnecting power");
    }
}
```

When the app runs, the *try-catch* statement will handle the exception and allow the app to take corrective action.

It is also possible to access the exception object within the body of the catch block, providing access to the message:

```
try
{
```

```
    VoltageCheck(501);
}

catch (ArgumentOutOfRangeException ex)
{
    Console.WriteLine(ex.Message);
}
```

So far, we have implemented code to handle a single exception type. The code within the body of the try block may, however, throw a variety of exception types. By accessing the exception object, we can identify the exception type and take type-specific actions. For example:

```
catch (Exception ex)
{
    if (ex is ArgumentOutOfRangeException)
    {
        Console.WriteLine(ex.Message);
    }
    else if (ex is NullReferenceException)
    {
        Console.WriteLine(ex.Message);
    }
}
```

23.5 Creating exception filters with the “when” keyword

In the above example, we used an *if … else if* statement to take different actions depending on the type of the caught exception. The *when* keyword provides a higher-level filtering option to control whether a catch block meets specific rules. The syntax for using *when* in a catch block is as follows:

```
try
{
}
catch (<exception>) when (<boolean expression>)
{

}
```

The code within the catch block will only execute if the Boolean expression after the *when* keyword evaluates to true. This is particularly useful for catching more than one exception type within a single catch block. The following code, for example, will only catch the exception if it matches either of the given types:

```
catch (Exception ex) when (ex is ArgumentOutOfRangeException ||
                           ex is ArgumentException)
```

Exception filters can contain any Boolean logical expression, with or without exception type matching. The following catch block will only execute if the exception type is ArgumentException and the *motorTemp* variable exceeds 102:

```
try
{
    VoltageCheck(501);
}
```

```
catch (Exception ex) when (ex is ArgumentOutOfRangeException && motorTemp > 102)
{
}
```

23.6 Using finally blocks

The purpose of the *finally* block is to specify code to be executed just before a *try-catch* statement finishes and is typically used to perform clean-up tasks, for example:

```
try
{
    VoltageCheck(501);
}
catch (Exception ex)
{
    Console.WriteLine(exception.Message);
}
finally
{
    Console.WriteLine("Resetting system.");
}
```

23.7 Using the try-finally Statement

From previous experience, we know that an unhandled exception will cause a running app to crash, but have yet to explain where exceptions can be caught and handled. The examples covered in this chapter so far have implied that exceptions must be handled at the point that they occur using a *try-catch* statement. In practice, however, this is not the case. Before exploring this topic in more detail it helps to view the execution path of a running app as a hierarchy of method calls, starting with the Main() method representing the top level.

Consider a scenario where the Main() method calls Method A which, in turn, calls methods B and C. Method C then calls Method D, and so on. This hierarchy of method calls is referred to as a *call tree* or *call hierarchy*. In diagram form, our theoretical call hierarchy might appear as shown in Figure 23-1 below:

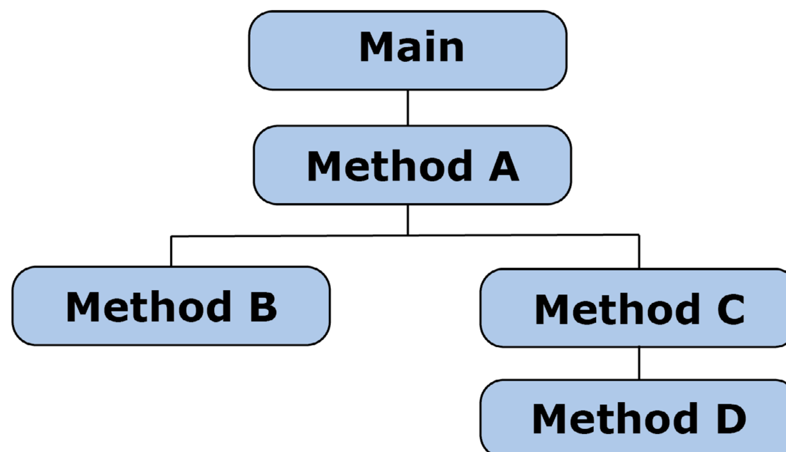


Figure 23-1

Suppose an exception is thrown in Method D. This exception will propagate up through the call hierarchy until the exception is handled. If the exception reaches the Main() method without being caught, the app will crash.

This means that even though the exception was thrown in Method D, we can handle it in any direct ancestor method up to and including Main() to prevent the app from crashing.

To experience this firsthand, add an interim method named StartMotor() between the Main() and VoltageCheck() methods and remove the *try-catch* statement:

```
namespace ExceptionDemo
{
    internal class Program
    {
        static void Main(string[] args)
        {
            StartMotor();
        }

        private static void StartMotor()
        {
            VoltageCheck(501);
            Console.WriteLine("Clean up after check.");
        }

        private static void VoltageCheck(int voltage)
        {
            if (voltage > 500)
            {
                throw new ArgumentOutOfRangeException(nameof(voltage),
                                        "Voltage exceeds safety parameters.");
            }
            else
            {
                Console.WriteLine("Safe Voltage");
            }
        }
    }
}
```

Since the exception is not caught in any of the three methods, the app will crash when it runs. We can, however, catch the error by adding a *try-catch* statement to VoltageCheck(), StartMotor() or Main(). For this example, use a *try-catch* statement to call StartMotor() from within the Main() method:

```
static void Main(string[] args)
{
    try
    {
        StartMotor();
    }
    catch (Exception ex) {
        Console.WriteLine(ex.Message);
```

```
    }
}
```

Rerun the app and confirm that the exception propagated up to the Main() method and was caught and handled:

```
Voltage exceeds safety parameters. (Parameter 'voltage')
```

Note that the above console output does not perform the cleanup operation after the voltage check. As currently implemented, StartMotor() will be unaware that the voltage check threw an exception and any code in StartMotor() after the VoltageCheck() will be skipped and control returned immediately to Main(). Suppose, however, that StartMotor needs to perform the cleanup after the voltage check, regardless of whether an exception was thrown. We can achieve this behavior using a *try-finally* statement:

```
private static void StartMotor()
{
    try
    {
        VoltageCheck(501);
    }
    finally
    {
        Console.WriteLine("Clean up whether or not voltage is safe.");
    }
}
```

Compile and run the app once more and verify that the cleanup occurs even when the exception is thrown:

```
Clean up whether or not voltage is safe.
Voltage exceeds safety parameters. (Parameter 'voltage')
```

23.8 Re-throwing exceptions

Sometimes an exception needs to be caught at multiple levels of the call hierarchy. This is achieved by *re-throwing* the exception after it has been caught so that it continues to pass upward in the hierarchy. Exceptions are re-thrown using the throw statement in the catch body as outlined below:

```
static void Main(string[] args)
{
    try
    {
        StartMotor();
    }
    catch (Exception ex) {
        Console.WriteLine("Caught in Main() " + ex.Message);
    }
}

private static void StartMotor()
{
    try
    {
        VoltageCheck(501);
    }
```