# Exception Handling

1

## OUTLINE

- Exception handling
- Example: Try, catch, finally
- Understanding execution flow
- Throwing exceptions
- Custom exceptions
- Filtered exceptions
- Inner exceptions
- Best practices

2

## EXCEPTION HANDLING

The **exception handling** features of the C# language let you deal with any unexpected or exceptional situations that occur whilst your code is running

There are four keywords used:

- **Try**: Try actions that may not succeed
- **Catch**: Handle failures
- **Finally**: Clean up resources
- **Throw**: Generate an exception

3

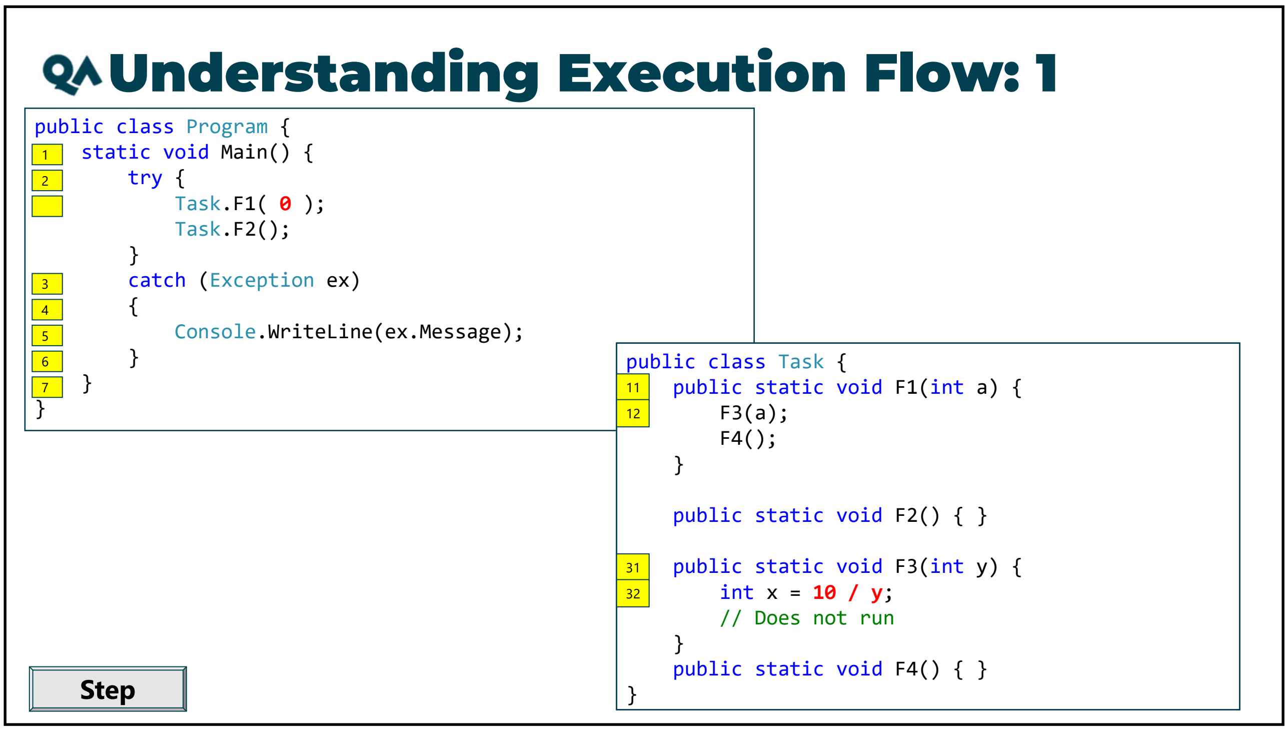
## EXAMPLE: TRY CATCH FINALLY

```
SqlConnection conn = new SqlConnection();
try
{
    conn.Open();
    //do things with connection
}
catch (SqlException ex)
{
    Console.WriteLine("Data access error: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("General error: " + ex.Message);
}
finally
{
    if (conn != null && conn.State == ConnectionState.Open)
    {
        conn.Close();
    }
}

//remainder of method
```

4

4



## Understanding Execution Flow: 1

```
   public class Program {
1      static void Main() {
2          try {
               Task.F1( 0 );
               Task.F2();
           }
3          catch (Exception ex)
4          {
5              Console.WriteLine(ex.Message);
6          }
7      }
   }
```

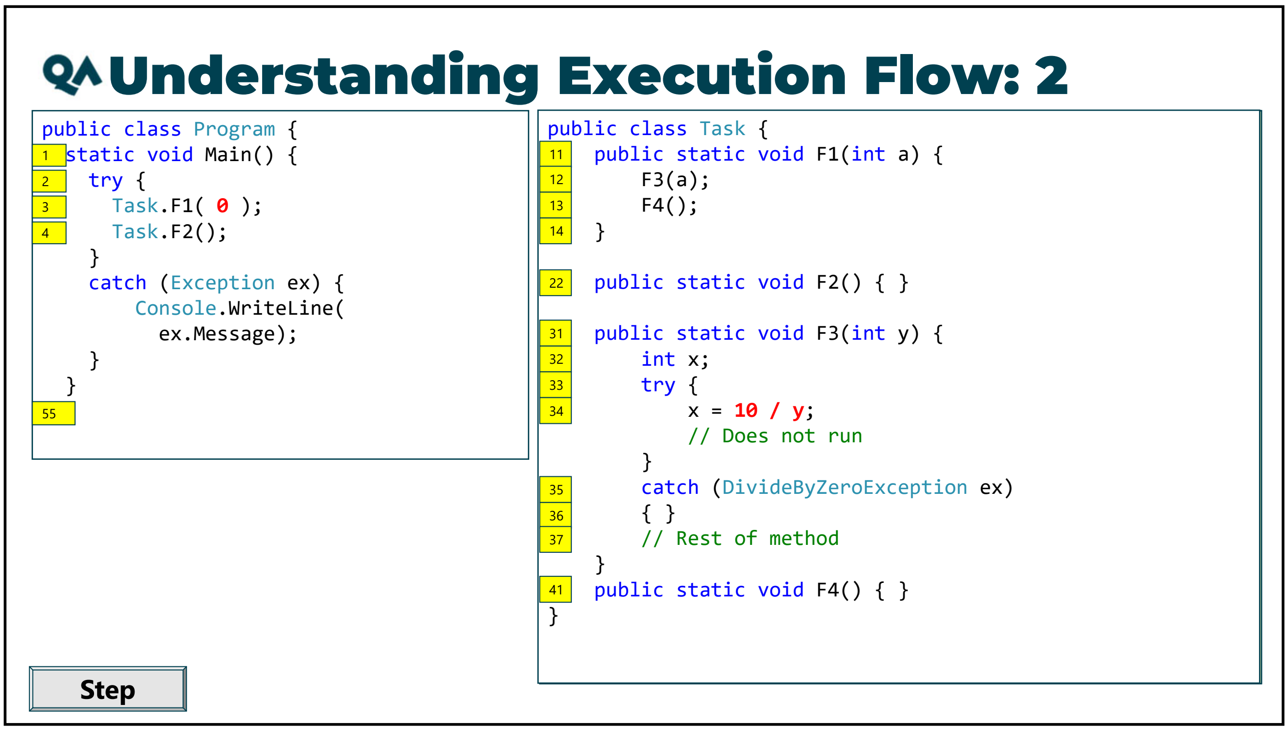
```
    public class Task {
11      public static void F1(int a) {
12          F3(a);
            F4();
        }

        public static void F2() { }

31      public static void F3(int y) {
32          int x = 10 / y;
            // Does not run
        }
        public static void F4() { }
    }
```

Step

5



## Understanding Execution Flow: 2

```
    public class Program {
1    static void Main() {
2      try {
3         Task.F1( 0 );
4         Task.F2();
       }
       catch (Exception ex) {
           Console.WriteLine(
             ex.Message);
       }
     }
55
```

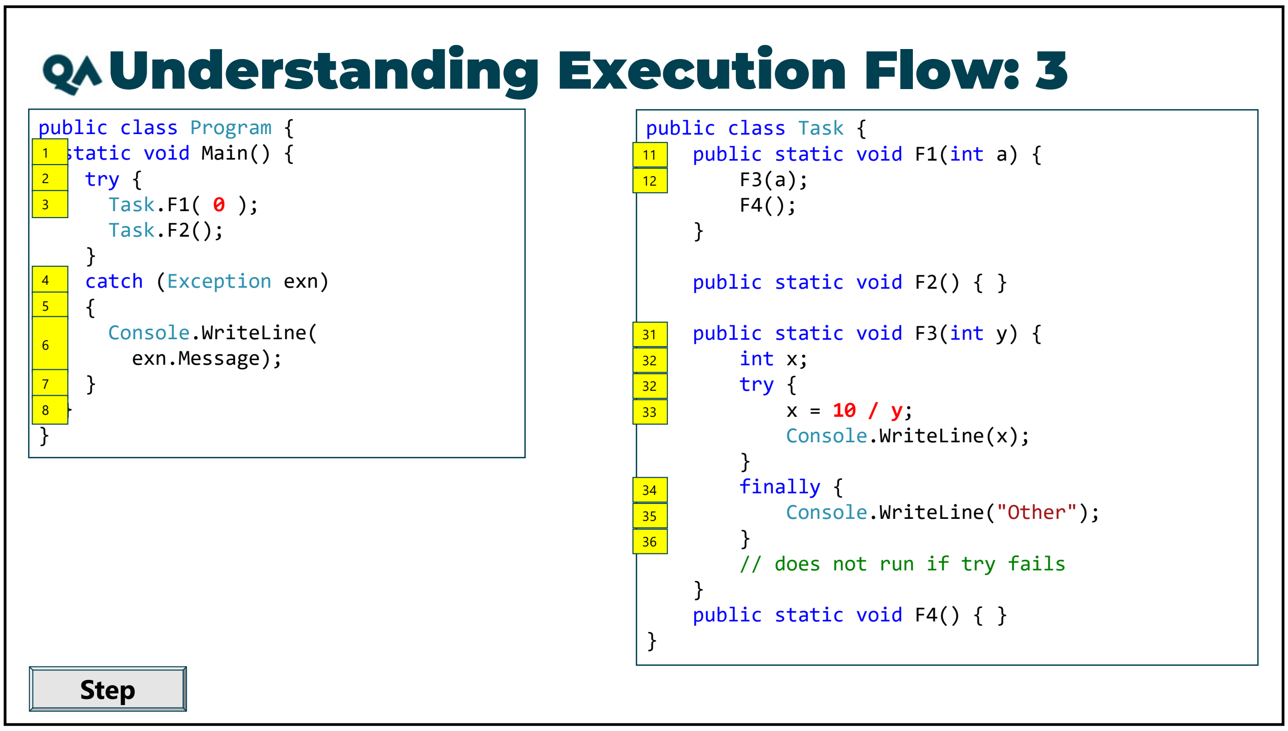
```
    public class Task {
11      public static void F1(int a) {
12          F3(a);
13          F4();
14      }

22      public static void F2() { }

31      public static void F3(int y) {
32          int x;
33          try {
34              x = 10 / y;
                // Does not run
            }
35          catch (DivideByZeroException ex)
36          { }
37          // Rest of method
        }
41      public static void F4() { }
    }
```

Step

6



## Understanding Execution Flow: 3

```
public class Program {
1  static void Main() {
2    try {
3      Task.F1( 0 );
       Task.F2();
     }
4    catch (Exception exn)
5    {
6      Console.WriteLine(
         exn.Message);
7    }
8  }
}
```

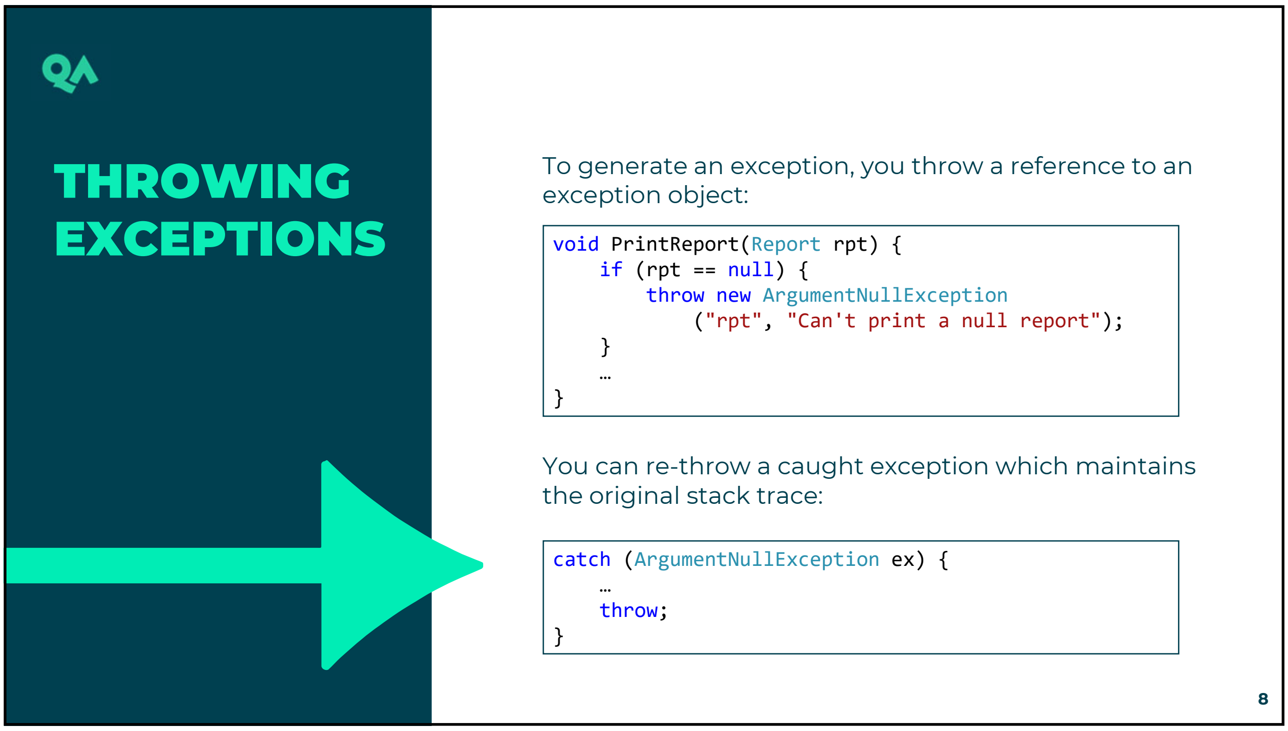
```
public class Task {
11    public static void F1(int a) {
12        F3(a);
          F4();
      }

      public static void F2() { }

31    public static void F3(int y) {
32        int x;
32        try {
33            x = 10 / y;
              Console.WriteLine(x);
          }
34        finally {
35            Console.WriteLine("Other");
36        }
          // does not run if try fails
      }
      public static void F4() { }
}
```

Step

7



## THROWING EXCEPTIONS

To generate an exception, you throw a reference to an exception object:

```
void PrintReport(Report rpt) {
    if (rpt == null) {
        throw new ArgumentNullException
            ("rpt", "Can't print a null report");
    }
    …
}
```

You can re-throw a caught exception which maintains the original stack trace:

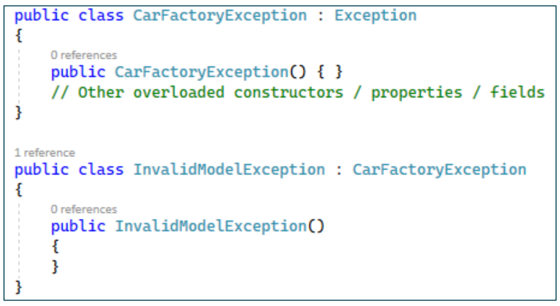
```
catch (ArgumentNullException ex) {
    …
    throw;
}
```

8

8

## CUSTOM EXCEPTIONS

Derive the class from **System.Exception**



```
public class CarFactoryException : Exception
{
    0 references
    public CarFactoryException() { }
    // Other overloaded constructors / properties / fields
}

1 reference
public class InvalidModelException : CarFactoryException
{
    0 references
    public InvalidModelException()
    {
    }
}
```

To provide rich exception details:

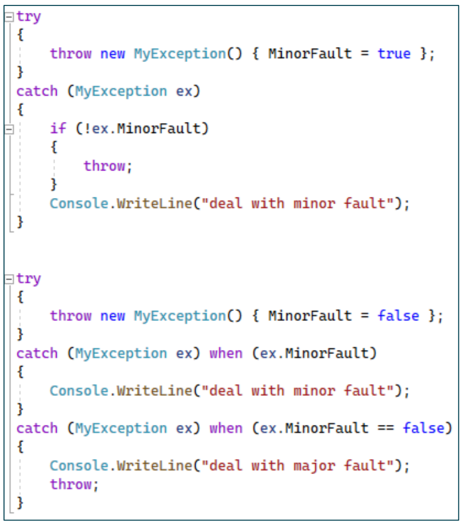
- Overload the constructor to pass in information
- Provide public properties to allow retrieval

9

9

## FILTERED EXCEPTIONS

- **User-filtered exception handlers** catch and handle exceptions based on requirements you define
- Use the **when** keyword with the **catch** statement:

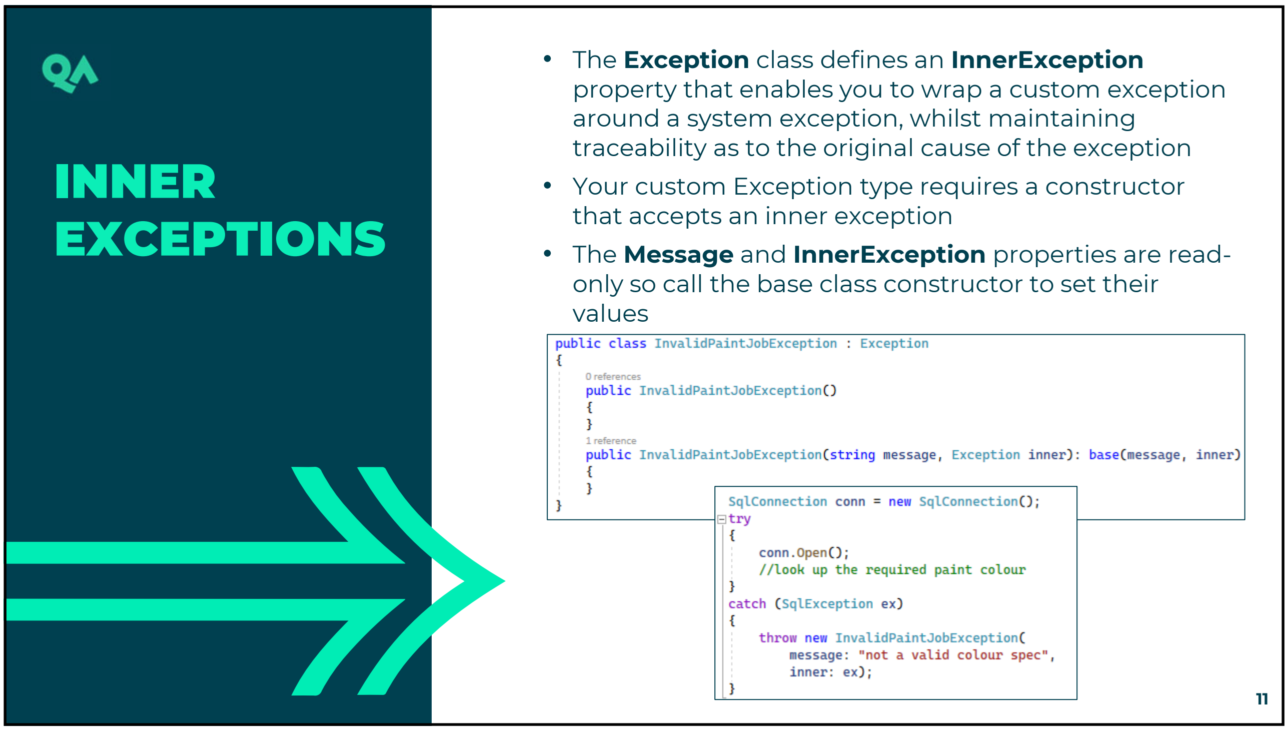


```
try
{
    throw new MyException() { MinorFault = true };
}
catch (MyException ex)
{
    if (!ex.MinorFault)
    {
        throw;
    }
    Console.WriteLine("deal with minor fault");
}


try
{
    throw new MyException() { MinorFault = false };
}
catch (MyException ex) when (ex.MinorFault)
{
    Console.WriteLine("deal with minor fault");
}
catch (MyException ex) when (ex.MinorFault == false)
{
    Console.WriteLine("deal with major fault");
    throw;
}
```

10

10



## INNER EXCEPTIONS

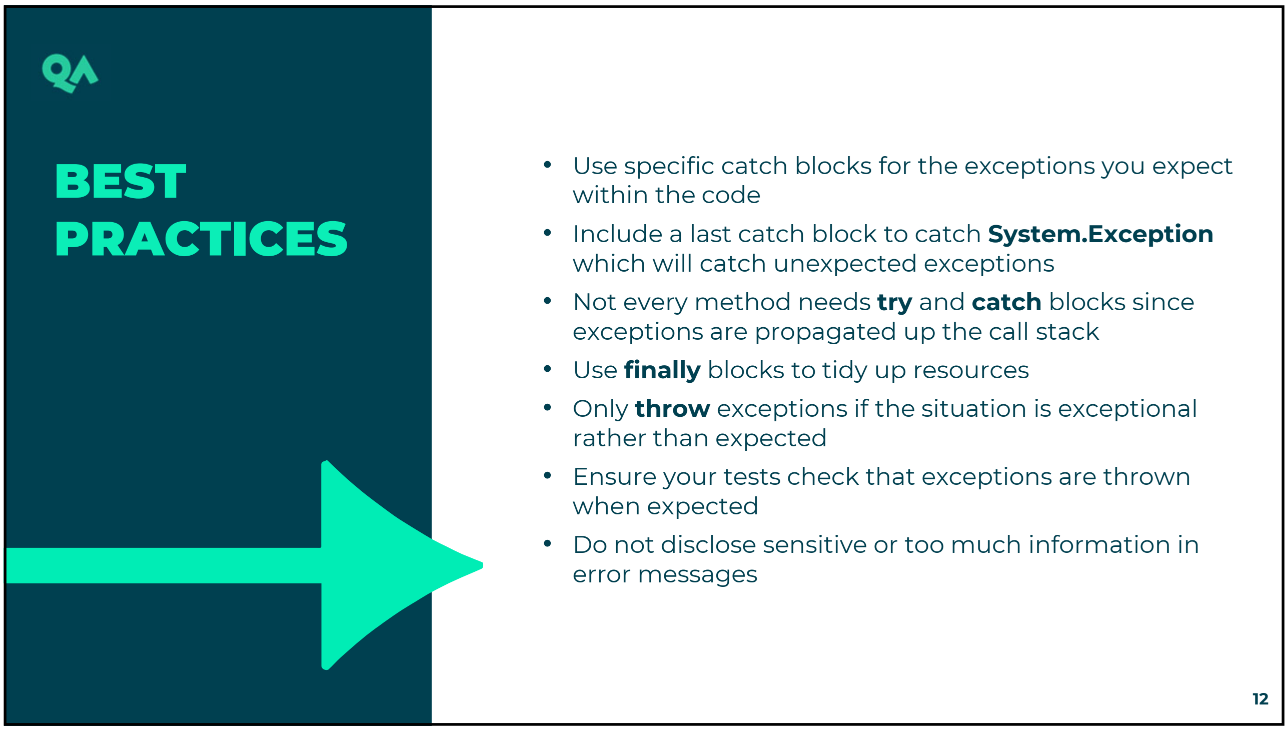
- The **Exception** class defines an **InnerException** property that enables you to wrap a custom exception around a system exception, whilst maintaining traceability as to the original cause of the exception
- Your custom Exception type requires a constructor that accepts an inner exception
- The **Message** and **InnerException** properties are read-only so call the base class constructor to set their values

```
public class InvalidPaintJobException : Exception
{
    0 references
    public InvalidPaintJobException()
    {
    }
    1 reference
    public InvalidPaintJobException(string message, Exception inner): base(message, inner)
    {
    }
}
```

```
SqlConnection conn = new SqlConnection();
try
{
    conn.Open();
    //look up the required paint colour
}
catch (SqlException ex)
{
    throw new InvalidPaintJobException(
        message: "not a valid colour spec",
        inner: ex);
}
```

11

11

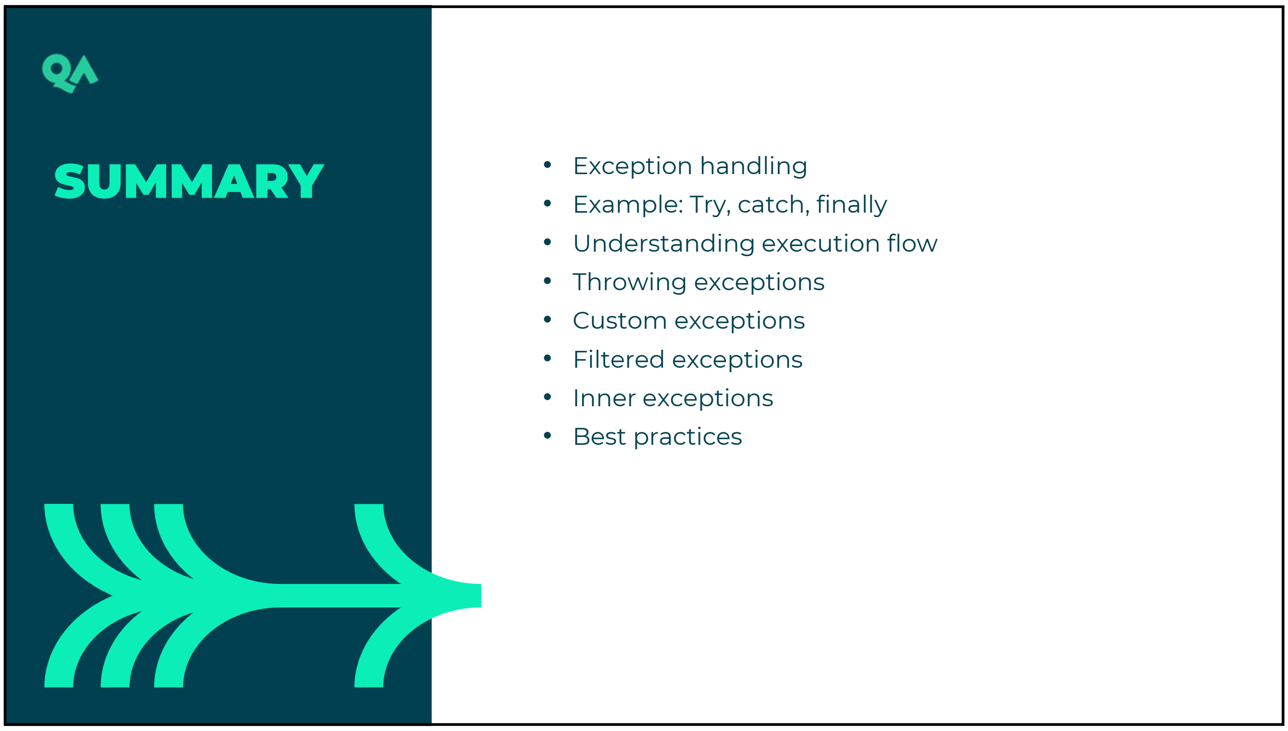


## BEST PRACTICES

- Use specific catch blocks for the exceptions you expect within the code
- Include a last catch block to catch **System.Exception** which will catch unexpected exceptions
- Not every method needs **try** and **catch** blocks since exceptions are propagated up the call stack
- Use **finally** blocks to tidy up resources
- Only **throw** exceptions if the situation is exceptional rather than expected
- Ensure your tests check that exceptions are thrown when expected
- Do not disclose sensitive or too much information in error messages

12

12



## SUMMARY

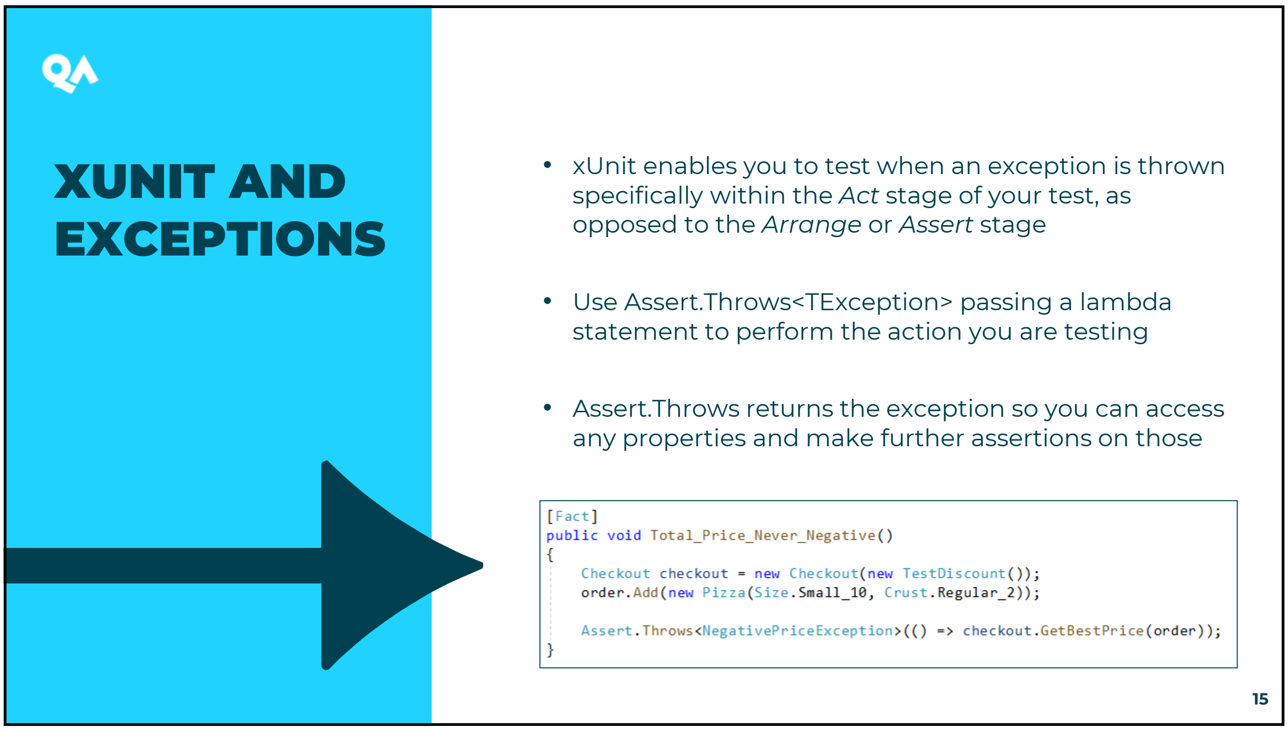
- Exception handling
- Example: Try, catch, finally
- Understanding execution flow
- Throwing exceptions
- Custom exceptions
- Filtered exceptions
- Inner exceptions
- Best practices

13

## Activity: Exercise 13

14

14



## XUNIT AND EXCEPTIONS

- xUnit enables you to test when an exception is thrown specifically within the *Act* stage of your test, as opposed to the *Arrange* or *Assert* stage
- Use Assert.Throws<TException> passing a lambda statement to perform the action you are testing
- Assert.Throws returns the exception so you can access any properties and make further assertions on those

```
[Fact]
public void Total_Price_Never_Negative()
{
    Checkout checkout = new Checkout(new TestDiscount());
    order.Add(new Pizza(Size.Small_10, Crust.Regular_2));

    Assert.Throws<NegativePriceException>(() => checkout.GetBestPrice(order));
}
```

15

15