


Awarding Great British Qualifications



Designing and Developing Object-Oriented Computer Programmes

Topic 10:

Testing and Error Handling

1

---

---

---

---

---

---


---

---

Scope and Coverage

*This topic will cover:*

- *Test Cases*
- *Boundary Cases*
- *Unusual Data*
- *Exception Handling*



2

---

---

---

---

---

---


---

---

Learning Outcomes

*By the end of this topic students will be able to:*

- *Employ black box testing;*
- *Understand the importance of regression testing;*
- *Employ effective test case design to detect errors;*
- *Handle exceptions through the try-catch architecture.*



3

---

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.4

Introduction - 1

- Writing a program is only one step of the process. We also need to write programs that work **correctly**.
- When we write a piece of code, we usually run the program and informally test to see if it works.
- However, this often leaves bugs and incorrect assumptions that will need to be corrected – something that looks like it works correctly might not actually be functioning as intended.



4

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.5

Introduction - 2

- For example, the answer to  $2 + 2$  and  $2 * 2$  is 4 in both cases.
- If we are expecting numbers to be multiplied but they're actually being added, getting a correct answer doesn't necessarily mean it was for the right reasons.
- In this topic we're going to discuss some strategies for testing, and how we can make use of the C# exception handling system to deal with errors we can't otherwise prevent.



5

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.6

Test Cases - 1

- The first step in fixing a program is to ensure you can replicate the circumstances under which it goes wrong.
- The worst kind of bugs to fix are those that are **intermittent** – their causes can be many and varied and staring at the code hoping it reveals its secret is rarely very productive.
- In software development, we encourage a methodical approach to testing that focuses on developing good **test cases** and then using those to assess program correctness.



6

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.7

Test Cases - 2

- Consider the following simple function:

```
public Double findHypotenuse(int side1, int side2)
{
    int hyp = side1 * side1 + side2 * side2;
    return Math.Sqrt(hyp);
}
```



7

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.8

Test Cases - 3

- This is a simple implementation of Pythagoras theorem, which states that to find the square of the length of a hypotenuse of a right angled triangle you take the square of two known sides and sum them.
- This is a well understood system that we know how to evaluate – we'll know if we get the right answers out of this function.



8

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.9

Test Cases - 4

- However, we need to make sure that we can be confident the answers check all the possible ways this function might go wrong.
- For that, we need to develop test cases.



9

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.10

Test Cases - 5

- Test cases are defined by their input values (what we'll send into the function), an **expected** return value (what we know the answer to be) and an **actual** return value (what the function gives us).
- We can't exhaustively test all possible combinations of numbers, so we use good test cases to build up a reasonable amount of confidence.



10

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.11

Test Cases - 6

- The first test cases we develop should focus on creating **equivalence cases** where we pick a few representative samples in a set of related inputs.
- For example, 'all positive numbers', 'first is positive, second is negative', and 'first is negative, second is positive', 'all negative numbers' and so on.



11

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.12

Test Cases - 7

- What the correct equivalence categories will be will vary from function to function.
- We might want to look at 'well-formed strings' or 'fully populated objects'.
- We get to decide what these equivalences might be for the needs we have.



12

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.13

Test Cases - 8

- We then build these up into a testing strategy, comprising what we intend to be the set of data we want to test. This is a common format:

Test Case	Input 1	Input 2	Expected	Actual
Positive 1	10	5	11.18	
Positive 2	20	100	101.98	
Positive 3	100	100	141.42	
Positive 4	9999	8	9999.00	
Positive 5	1	2	2.23	
Positive 6	5	10	11.18	
Positive 7	100	20	101.98	
Positive 8	8	9999	9999.00	



13

---

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.14

Test Cases - 9

- Note here that we have a range of values – large numbers with small numbers, numbers that are the same, and ‘mirrors’ of earlier test cases.
- We can’t put every combination of numbers in here, so we want a range of them.
- Remember, we need to calculate by hand the **expected** value, or use a properly calibrated software tool in which we have confidence to do it for us.



14

---

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.15

Test Cases - 10

- To determine if our program works, we then input each of these test cases into a program and then see what it tells us.
- We can do that by hand, or we can write a (properly tested) **test harness** to input the values and then output the results for us.



15

---

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.16

Test Cases - 11

- In either case, we can then fill out our **actual** results to see whether the program is working correctly:

Test Case	Input 1	Input 2	Expected	Actual
Positive 1	10	5	11.18	11.18
Positive 2	20	100	101.98	101.98
Positive 3	100	100	141.42	141.42
Positive 4	9999	8	9999.00	9999.00
Positive 5	1	2	2.23	2.23
Positive 6	5	10	11.18	11.18
Positive 7	100	20	101.98	101.98
Positive 8	8	9999	9999.00	9999.00



16

---

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.17

Test Cases - 12

- You can see here than in every case the **expected** answer matches the **actual** answer.
- This function seems to be working correctly, for this set of data.
- Eight test cases isn't enough to say it works perfectly for all positive numbers – just that it works correctly for these.
- The more test cases we have, the more confident we can be that the results will correctly represent the untested possibilities.



17

---

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.18

Test Cases - 13

- We can never be sure however. Programming is inherently a discipline in which we must be prepared to accept uncertainty.
- Buoyed by these results, we might declare the function correct. But that's only with this single equivalence case.
- Let's try it with positive and negative numbers.
- We should expect it to work, but we need to be sure.



18

---

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.19

Test Cases - 14

- So:

Test Case	Input 1	Input 2	Expected	Actual
Pos & Neg 1	10	-5	11.18	11.18
Pos & Neg 2	-20	100	101.98	101.98
Pos & Neg 3	100	-100	141.42	141.42
Pos & Neg 4	-9999	8	9999.00	9999.00



19

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.20

Test Cases - 15

- That seems to work, and so too when we add in test cases for using negative numbers only.
- Within those equivalence classes, we get the correct results.
- But there are other test cases we need to consider.



20

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.21

Test Cases - 16

- Core to the testing process is **retesting** once you change anything in the code.
- It's a common industry benchmark that for every two bugs you fix, you'll often inadvertently introduce another.
- As such, **regression testing** is the process of making sure that every single fix is accompanied by a rigorous repetition of the test data that you've already entered.



21

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.22

Test Cases - 17

- This is where the idea of writing some code to automate this becomes especially valuable.
- That's something that needs to be done on an individual basis, but usually involves the writing of a class which is populated with test data, and then a for loop which executes the tested function for each test data object.



22

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.23

Boundary Cases - 1

- Boundary cases are those that exist at the point one part of a piece of code will change based on a value.
- For example, if we have a **for** loop that should iterate ten times, we would write test cases that would ensure it triggers correctly by clustering test cases around that boundary.
- We'd check that it works correctly for 8, 9, 10, 11 and 12.



23

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.24

Boundary Cases - 2

- If there were other loops inside the system, we'd check those too.
- If there are **if** statements, we check to see what happens if they are true or false, and cluster around those conditions.



24

---

---

---

---

---

---

---



Title of Topic: Topic 1 - 1.25

Boundary Cases - 3

- Let's consider a second example function to test:

```
public int calculatePower(int baseNum, int powerNum)
{
    int ans;
    int numRepeats;

    ans = baseNum;

    numRepeats = powerNum;

    for (int counter = 0; counter < numRepeats; counter++)
    {
        ans = ans * baseNum;
    }

    return ans;
}
```

NCC

25

---

---

---

---

---

---

---

Title of Topic: Topic 1 - 1.26

Boundary Cases - 4

- We've encountered this earlier in unit – here, we're separating it out into a function of its own for ease of testing.
- We'd cluster test cases around the **numRepeats** boundary, like so:

Test Case	Input 1	Input 2	Expected	Actual
Boundary 1	10	4	10000	
Boundary 2	11	4	14641	
Boundary 3	9	4	6501	
Boundary 4	10	3	1000	
Boundary 5	10	5	100000	
Boundary 6	10	2	100	
Boundary 7	10	6	1000000	

NCC

26

---

---

---

---

---

---

---

Title of Topic: Topic 1 - 1.27

Boundary Cases - 5

- Note here we're not wildly varying these cases – we're targeting them around a potential point of weakness in the code.
- Wherever a program branches, there's a chance that our code will function incorrectly.

NCC

27

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.28

Boundary Cases - 6

- When we run our test cases, we'll find out if that boundary check is likely to be safe (at least, for this set of data):

Test Case	Input 1	Input 2	Expected	Actual
Boundary 1	10	4	10000	100000
Boundary 2	11	4	14641	161051
Boundary 3	9	4	6501	59049
Boundary 4	10	3	1000	10000
Boundary 5	10	5	100000	1000000
Boundary 6	10	2	100	1000
Boundary 7	10	6	1000000	10000000



28

---

---

---

---

---

---

---

---


Title of Topic: Topic 1 - 1.29

Boundary Cases - 7

- None of those values are correct, and it's because in the process of writing this function we broke the code a little.
- We need to fix this line:

```
numRepeats = powerNum;
```
- So that it says:

```
numRepeats = powerNum - 1;
```



29

---

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.30

Boundary Cases - 8

- Make that fix, and then run the test cases again:

Test Case	Input 1	Input 2	Expected	Actual
Boundary 1	10	4	10000	10000
Boundary 2	11	4	14641	14641
Boundary 3	9	4	6501	6501
Boundary 4	10	3	1000	1000
Boundary 5	10	5	100000	100000
Boundary 6	10	2	100	100
Boundary 7	10	6	1000000	1000000



30

---

---

---

---

---

---


---

---

Title of Topic Topic 1 - 1.31

Boundary Cases - 9

- That's much better.
- Now we can see that not only does the code work correctly, it works correctly around the boundary.
- It's not the case we'd use equivalence cases **or** boundary cases.
- We'd use both together in a single testing regime.



31

---

---

---

---

---


---

---

Title of Topic Topic 1 - 1.32

Unusual Data - 1

- Next, we want to throw a whole set of 'unusual' data at a function to see what happens.
- For example, we might want to throw in the maximum number that can be represented, or floating point numbers when we expect ints. or booleans when we expect strings, or strings when we expect booleans.
- Perhaps binary data when we should be providing numbers.



32

---

---

---

---

---


---

---

Title of Topic Topic 1 - 1.33

Unusual Data - 2

- Remember, for most programs the values that we provide here will come from users, and often they'll be extracted from user elements such as text fields.
- We can't guarantee a user will enter anything sensible in there, so we need to know what's going to happen if they don't.



33

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.34

Unusual Data - 3

- Let's go back to our calculatePower function and go out of our way to break it:

Test Case	Input 1	Input 2	Expected	Actual
Unusual 1	MaxInt	MaxInt	MaxInt	
Unusual 2	MaxInt	0	0	
Unusual 3	0	MaxInt	0	



34

---

---

---

---

---

---

---

---


Title of Topic: Topic 1 - 1.35

Unusual Data - 4

- We can get the MaxInt value using Int32.MaxValue, so that's what we'll do. Run these test cases and we get...

Test Case	Input 1	Input 2	Expected	Actual
Unusual 1	MaxInt	MaxInt	???	ERROR
Unusual 2	MaxInt	0	1	2147483647
Unusual 3	0	MaxInt	0	0

- The problem here is that the function we have is not robust. We can't deal with MaxInt cleanly because we can't actually raise MaxInt to a power.



35

---

---

---

---

---

---

---

---


Title of Topic: Topic 1 - 1.36

Unusual Data - 5

- So now we know we need to fix our code so that first situation can't occur:

```
public int calculatePower(int baseNum, int powerNum)
{
    int ans;
    int numRepeats;

    if (baseNum == Int32.MaxValue && powerNum > 1) {
        return Int32.MaxValue;
    }
    ans = baseNum;
    numRepeats = powerNum - 1;
    for (int counter = 0; counter < numRepeats; counter++)
    {
        ans = ans * baseNum;
    }
    return ans;
}
```



36

---

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.37

Unusual Data - 6

- That fixes our first case, which now becomes what we set:

Test Case	Input 1	Input 2	Expected	Actual
Unusual 1	MaxInt	MaxInt	2147483647	2147483647
Unusual 2	MaxInt	0	1	2147483647
Unusual 3	0	MaxInt	0	0



37

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.38

Unusual Data - 7

- The second use case problem is that any number, when raised to a power of zero, should become one.
- We don't do that – we set the starting number to be the base number.



38

---

---

---

---

---

---

---

Title of Topic: Topic 1 - 1.39

Unusual Data - 8

- So we need to have another special case to handle a power of zero:

```
public int calculatePower(int baseNum, int powerNum)
{
    int ans;
    int numRepeats;


    if (baseNum == Int32.MaxValue && powerNum > 1) {
        return Int32.MaxValue;
    }

    if (powerNum == 0)
    {
        return 1;
    }

    ans = baseNum;
    numRepeats = powerNum - 1;

    for (int counter = 0; counter < numRepeats; counter++)
    {
        ans = ans * baseNum;
    }

    return ans;
}
```



39

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.40

Unusual Data - 9

- And then that will give us the following:

Test Case	Input 1	Input 2	Expected	Actual
Unusual 1	MaxInt	MaxInt	2147483647	2147483647
Unusual 2	MaxInt	0	1	1
Unusual 3	0	MaxInt	0	0



40

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.41

Unusual Data - 10

- As we add new and interesting unusual data into our test cases, we'll often find strange things happen.
- Good test cases will reveal flaws in our program – we want to really treat our code as viciously as we can to reveal the things we need to fix and improve.



41

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.42

Exception Handling - 1

- Sometimes we can't actually test for a scenario because it is user dependant.
- If we ask a user to enter a number into a textbox, we can't be sure that they won't write "eight" when we expect 8.
- If we ask a user to pick an element out of an array, we won't necessarily be able to know they'll enter a valid index.



42

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.43

Exception Handling - 2

- We can write code to check such input, but there are other scenarios where we won't have control.
- We won't know, for example, if a hard-drive fails as we're writing to a file, or if a database becomes corrupted as we read from it.
- If we're trying to connect to a remote website, we won't necessarily be able to be sure the internet connection is sustained through the transaction.



43

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.44

Exception Handling - 3

- Those are **exceptional** circumstances, and C# gives us a framework for dealing with them.
- They're called **exceptions**.



44

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.45

Exception Handling - 4

- With exception handling, we place code that might potentially go wrong inside a **try** block, which says to the virtual machine 'This might go horribly wrong so treat it with a bit of care'.
- If the code encounters a problem, the virtual machine will generate an **exception**, which is something that needs to be dealt with in some way.
- It **throws** that exception to us, and we handle it in a **catch** block polymorphically typed to the class of exception.



45

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.46

Exception Handling - 5

- Some of these are known as **unchecked** exceptions, which means that they'll happen at runtime and we don't necessarily need to deal with them – usually we can write code that ensures it doesn't happen.
- Other exceptions are **checked** exceptions and must be dealt with before the program will compile.
- We'll see some of those in the next two topics.



46

---

---

---

---

---

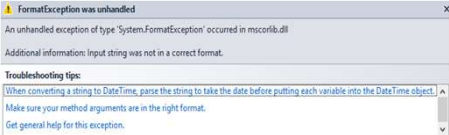
---


---

Title of Topic: Topic 1 - 1.47

Exception Handling - 6

- Let's consider a common example we've used throughout the module – letting someone enter a number into a text box.
- We parse that using `Int32.Parse`, but that doesn't work especially well if we try to parse a non-numerical string:





47

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.48

Exception Handling - 7

- That's what an **unchecked exception** looks like when it's thrown.
- If we want to avoid that happening when a program runs (and we do), we need to **try** the parsing and then provide an appropriate response when it occurs.
- Like so:

```
try
{
    num = Int32.Parse(txtNum.Text);
}
catch (FormatException ex)
{
    MessageBox.Show("Please enter a real number");
}
```



48

---

---

---

---

---

---


---



Title of Topic: Topic 1 - 1.49

Exception Handling - 8

- Note here that we catch a `FormatException` – that’s what the dialog above told us was being thrown.
- We can also catch the general case `Exception`, because all exceptions have `Exception` in their inheritance tree and the catch statement works polymorphically.
- That’s handy, but not exactly user friendly – we want to be able to give meaningful guidance, and that only happens when we know **what** exception they just triggered.



49

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.50

Exception Handling - 9

- We can however ‘tier’ these responses to ensure robust programs that are also helpful for the user.
- As with an if/else-if we can add Catch statements indefinitely – the only requirement is that the most specialised exception has to be at the top of the structure, or more general cases will catch it first:



50

---

---

---

---

---


---

---

Title of Topic: Topic 1 - 1.51

Exception Handling - 10

```
try
{
    num = Int32.Parse(txtNum.Text);
}
catch (FormatException ex)
{
    MessageBox.Show("Please enter a real number");
}
catch (Exception ex)
{
    MessageBox.Show("Something went wrong. I don't know what.");
}
```



51

---

---

---

---

---

---


---

Title of Topic: Topic 1 - 1.52

Exception Handling - 11

- Note too that we're giving the exception we catch a name – ex.
- The exception is an object like much of what we work with within C#.
- It also contains full details about the actual cause of the exception, which we can access:

```
catch (FormatException ex)
{
    MessageBox.Show("Please enter a real number: " + ex.Message);
}
```



52

---

---

---

---

---

---

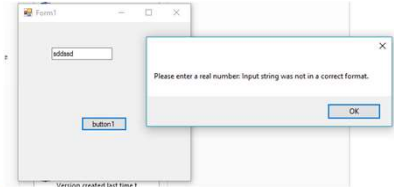
---


---

Title of Topic: Topic 1 - 1.53

Exception Handling - 12

- Usually these messages are not ready for users, but they can be useful to us when we're catching exceptions to make sure that we, while developing, are aware of what's going on:





53

---

---

---


---

---

---

---

---



Awarding  
Great British  
Qualifications

Topic 10 – Testing and Error Handling

Any Questions?

54

---

---

---

---

---

---

---

---