Lecture 5:
Exceptions, Specifications, and Testing

Learning Goals

- Understand what an **exception** is, and what they are useful for
- Use exceptions in combination with a **try—catch** block
- Create and use a custom exception class
- Define and give examples of **preconditions** and **postconditions**
- Write complete and correct method **specifications**
- Partition inputs to design tests for a method that cover all expected behaviours
- Write unit tests against a method

Exceptions

Exceptions are special objects used to signal:

- bugs in code, allowing us to track down coding errors more effectively
- special conditions which should be handled differently

```
try
{
    string s = SomeFunction(); //some process assigns null to this variable
    var l = s.Length;
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}
```

System.NullReferenceException: Object reference not set to an instance of an object.

Exceptions (Throw Custom Message)

```
try
{
    string s = null;
    if (s == null)
        throw new Exception("Throw your own exception with custom message!")

}
catch (Exception ex)
{
    Console.WriteLine(ex);
}
```

Exceptions

Exceptions can by **thrown** in code using a `throw` statement (actually, *anything* can be thrown, but good programming practice dictates that they should be some derived class of `Exception`).

Exceptions can be **caught** in a `try - catch` block. If an exception is not caught, it propagates up the call stack, exiting each function call at the current location, destroying all local variables in the process. If left uncaught, it will eventually trigger the program to terminate.

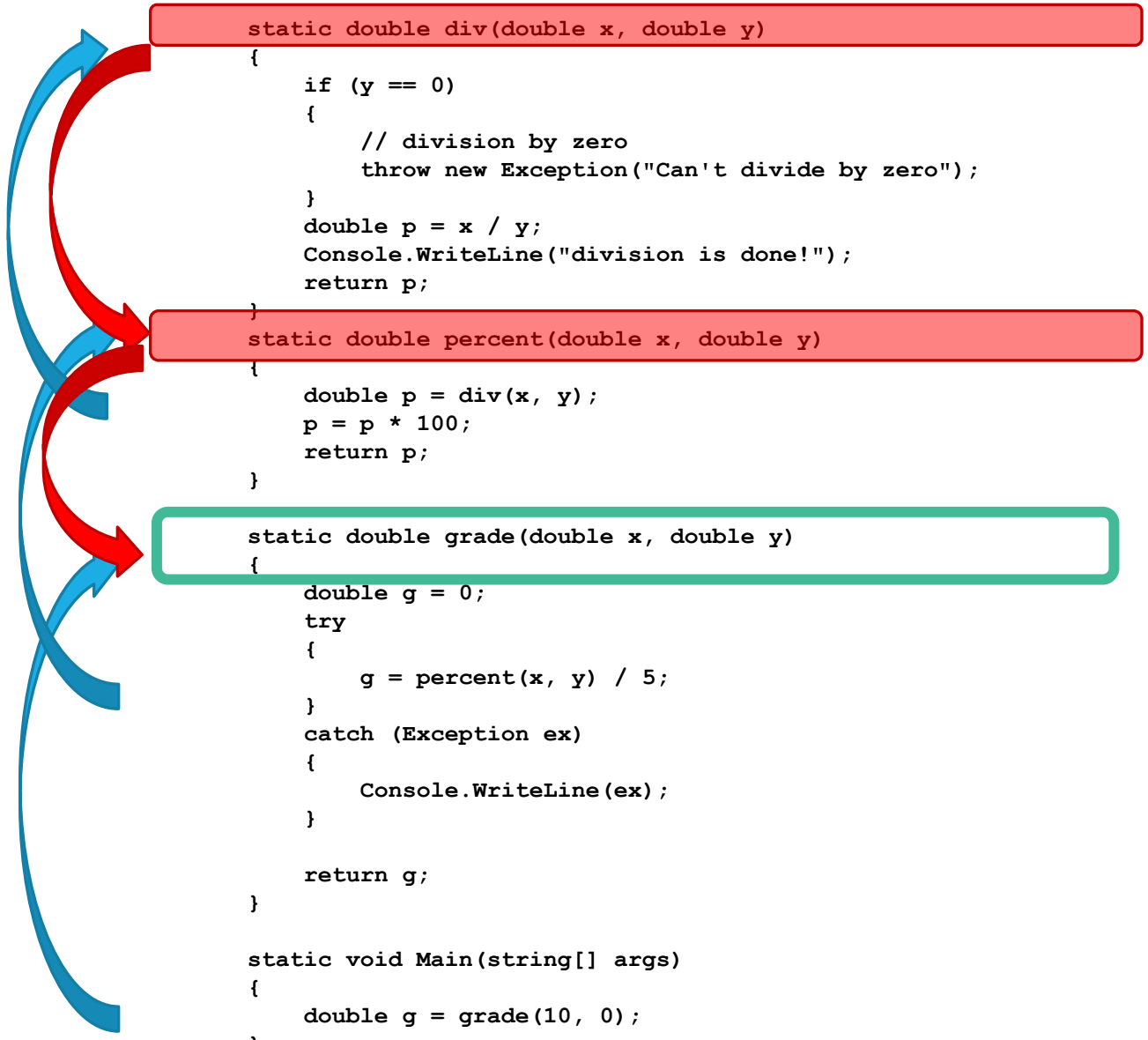
Exceptions

Exceptions:

- Can be **thrown** and **caught** in code
- Stop execution of current function at throw statement
- Propagate upwards through call stack until caught
- All local variables are destroyed as exception propagates

Useful for:

- Special handling of unexpected or detrimental events
- Separation of normal and error-handling code



```
static double div(double x, double y)
{
    if (y == 0)
    {
        // division by zero
        throw new Exception("Can't divide by zero");
    }
    double p = x / y;
    Console.WriteLine("division is done!");
    return p;
}
static double percent(double x, double y)
{
    double p = div(x, y);
    p = p * 100;
    return p;
}

static double grade(double x, double y)
{
    double g = 0;
    try
    {
        g = percent(x, y) / 5;
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
    }

    return g;
}

static void Main(string[] args)
{
    double g = grade(10, 0);
}
```

Custom Exception in C#

Create a class that inherits from `Exception`. Feel free to add any other useful members/methods for extracting useful information about the exception.

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            string s = null;
            if (s == null)
                throw new NullValueCustom();
        }
        catch
        {
            Console.WriteLine("My custom Exception says no Null Value");
        }

    }
}
```

```
public class NullValueCustom : Exception
{
    public NullValueCustom()
    {
    }
}
```

Multiple Catch Statements

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            string s = null;
            double numerator = 10; double denominator = 0;
            if (s == null)
                throw new NullValueCustom();
             if (denominator == 0)
                throw new NotDividbyZero();

        }
        catch(NullValueCustome)
        {
            Console.WriteLine("My custom Exception says no Null Value");
        }
         catch(NotDividbyZero)
        {
            Console.WriteLine("Denominator Cannot be zero");
        }

    }
}
```

Exceptions: Uses and Abuses

Exceptions are designed for **exceptional circumstances**. They should not be used for common occurrences. For conditions that are likely to occur but might trigger an exception, consider handling them in a way that will avoid the exception.

There is a performance **penalty** when code takes the exception path.

Exceptions allow us to **separate** error-handling code from the normal execution path. Important uses are in File IO, network communication, and when users enter invalid inputs in GUIs (think: error message boxes).

https://docs.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions

Specifications

Many software bugs arise when interfacing between two pieces of code because of a misunderstanding of what a particular function does, or how it behaves for a set of given set of inputs.

What does this function do?

```
int find(int[] data, int val)

int a0 = find({0, 1, 2, 3, 4, 5}, 2);
int a1 = find({0, 1, 2, 3, 4, 5}, 6);
int a2 = find({0, 1, 1, 1, 4, 5}, 1);
int a3 = find({}, 1);
```

Specifications

A **specification** is a ***contract*** detailing **conditions** on the inputs, description of the **outputs**, and any **guarantees** that the method or class makes. It should provide all information a programmer needs to be able to use your code effectively, ***without*** needing to know the implementation details.

Preconditions: set of conditions that must be satisfied by the input arguments. The onus is on the caller to guarantee these, otherwise the behaviour of the code is left unspecified.

Postconditions: if the preconditions hold, precisely specifies the outputs: which variables are modified, if exceptions are thrown, and any other *effects*. The implementer has the obligation to guarantee these.

Specifications

Preconditions

```
// REQUIRES:  data is a non-empty array
// EFFECTS:   returns the index of the last occurrence
//            of val in data, or -1 if not found
int find(int[] data, int val);
```

Postconditions

```
int a0 = find({0, 1, 2, 3, 4, 5}, 2);
int a1 = find({0, 1, 2, 3, 4, 5}, 6);
int a2 = find({0, 1, 1, 1, 4, 5}, 1);
int a3 = find({}, 1);  // fails precondition, output indeterminate
```

Specifications

```
// REQUIRES:  any array data, integer val
// EFFECTS:   returns the index of the first occurrence
//            of val in data, throws an ItemNotFound
//            exception if not found
int find(int[] data, int val);
```

```
int a0 = find({0, 1, 2, 3, 4, 5}, 2);
int a1 = find({0, 1, 2, 3, 4, 5}, 6);
int a2 = find({0, 1, 1, 1, 4, 5}, 1);
int a3 = find({}, 1);
```

Specifications

```
// REQUIRES:  data to be sorted in ascending order
// EFFECTS:   returns the index of an occurrence
//            of val in data, -1 if not found.
// GUARANTEES: search time is O(log(n))
int find(int[] data, int val);
```

```
int a0 = find({0, 1, 2, 3, 4, 5}, 2);
int a1 = find({0, 1, 2, 3, 4, 5}, 6);
int a2 = find({0, 1, 1, 1, 4, 5}, 1);
int a3 = find({}, 1);
```

Specifications as Documentation

```
/**
 * Searches the vector data to find the element val
 * in O(log(n)) time
 *
 * @param data array to search, sorted in ascending order
 * @param val  value to find
 * @return first occurence of val in data
 * @throws ItemNotFound if val is not found
 */
int find(int[] data, int val);
```

Note: These are Doxygen-style comments. Doxygen is a tool that can be used to automatically compile online documentation. https://www.doxygen.nl/index.html

Specifications

If there are no specified **preconditions** on an input, then it is assumed to accept any value according to its type.

If the **postconditions** don't mention modifying an input, then it is assumed they remain untouched.

***All* effects** must be **explicitly** described, including if internal member variables are modified in a way to have observable changes in the class's behaviour.

Specifications should *never* mention local variables, private fields, or implementation details. That way, the programmer should be free to change the implementation without needing to modify the specification.

Specifications

```
// if c1 will be modified directly, use
void use_cylinder( ref int c1);

// if a copy of c1 is needed, use
void use_cylinder(int c1);
```

Forces c1 to be non-null, but can be modified

Forces c1 to be non-null, guaranteed not to modify original (makes a copy)

Testing

Testing is a component of a more general process called **validation**.

Validation includes:

- **Formal reasoning**: also called **verification**, constructs a formal proof that the program is correct.
- **Code review**: code "proof-reading" by a colleague or peer to look for bugs/inefficiencies.
- **Testing**: running the program on *carefully* **selected inputs** and checking the results against **known answers**.

Testing

Proper testing can be hard. It requires us to switch from programmer to tester, ***lose the ego***, and purposely **try to break** our own code.

Exhaustive Testing: going through every possible input, checking all outputs against known solution. This is often infeasible.

Haphazard Testing: choosing a few inputs/outputs to verify method functions as expected. This is often a first step, will likely catch basic bugs, shows that code is at least partially functional. Does nothing to instill confidence in program is correct.

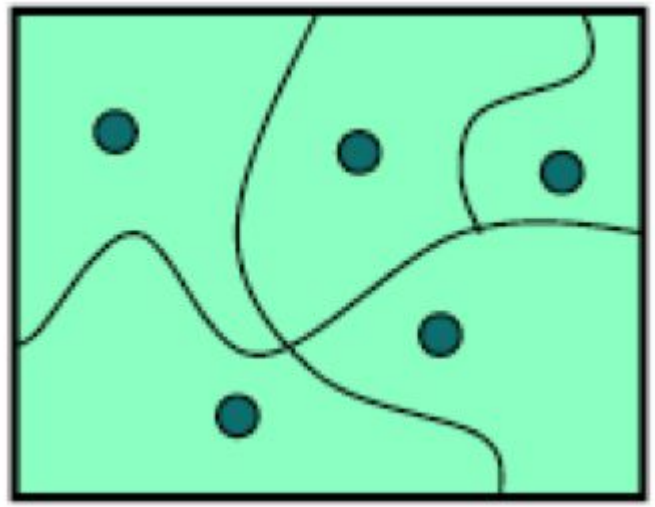
Random/Statistical Testing: randomly generate inputs, potentially comparing outputs to a different implementation already proven to be correct. Unfortunately, software behaves discontinuously and discretely across the space of possible inputs, often at an input boundary. e.g. the Pentium division bug.

http://www.willamette.edu/~mjaneba/pentprob.html

Testing

The most effective approach is to smartly and **systematically** choose test cases, based on the specifications, that cover all sets of expected distinct behaviours.

- **Partition** the input domain into a set of sub-domains, each consisting of inputs with similar expected program behaviours
- Choose one or two test cases from each sub-domain
- Choose test cases to trigger every possible *effect* (i.e. postcondition)
- Choose one or two test cases from each **boundary** (or **edge**, or **corner**) between sub-domains



https://ocw.mit.edu/ans7870/6/6.005/s16/classes/03-testing/

Testing

```
/**
 * Searches the array data to find the element val
 * in O(log(n)) time
 *
 * @param data array to search, sorted in ascending order
 * @param val  value to find
 * @return first occurence of val in data
 * @throws ItemNotFound if val is not found
 */
int find(int[] data, int val);
```

Input sub-domains:

- data contains one instance of val
- data contains multiple instances of val
- data contains no instance of val

Boundaries:

- array of size zero
- array of size one
- val is first item
- val is last item

Testing

Unit Testing: testing each method/class (i.e. a **unit**) in **isolation**. Each partition of inputs is usually separated into a separate test so that a failed test will indicate exactly what type of bug to search for.

Integration Testing: testing the **system as a whole** to ensure the various components of the software function properly in combination. If all unit tests pass, then you know the issue is somewhere in the interfacing or timings.

Regression Testing: testing the system to ensure it performs exactly as it did in a **previous functional** version. Often important when adding new features or optimizing parts of code, making sure you don't break any part that works.

Unit Testing Frameworks

xUnit.net: xUnit.net is a free, open source, community-focused unit testing tool for the .NET Framework https://xunit.github.io/

NUnit: widely used unit testing framework for .NET. https://github.com/nunit