Contents

Exceptions

Introduction

- At *execution time* programs can run into unspecified behaviour, such as
  - having to divide by zero,
  - having to access an array at an index greater than its length.
- For example, the following instructions would compile just fine, but *at execution time* the program would "explode":

```
int zero = 0;
Console.WriteLine($"Let's divide by zero: {1 /
 ↪  zero}.");
```

```
int[] test = new int[2];
test[2] = 3;
```

  - In the first case, a "System.DivideByZeroException has been thrown" error message would be displayed.
  - In the second case, a "System.IndexOutOfRangeException has been thrown" error message would be displayed.
  - Those are examples of exceptions thrown by operations[1].
- Methods can also throw exceptions. For example, the following statement:

```
int x = int.Parse("This is not a number.");
```

will display a "System.FormatException has been thrown" error message. This is because the `Parse` method can *throw an exception*[2].

[1] https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/exceptions#215-common-exception-classes

[2] https://learn.microsoft.com/en-us/dotnet/api/system.int32.parse?view=net-8.0#system-int32-parse(system-string)

- Of course, a programmer would not *purposely* introduce such strange instructions in their code, but they may arise after interacting with the "outside world", that is, a user, file, or other external factor.
- C# allows *exception handling*, which are ways of recovering when such exceptions are thrown, so that the program can keep on executing. Stated differently, they instruct the program what to do, for example, if it is asked to perform a division by 0. This is handled by **`catch`** blocks.
- C# also allows **`finally`** block, which contain code executed unconditionally, that is, regardless of if the exception was thrown or not.

Syntax and Rules for `try`...`catch`...`finally` Statements

- In a first approximation, the syntax of a **`try`**...**`catch`**...**`finally`** statement is as follows:

```
try{
    <statement block 1>
}
catch{
    <statement block 2>
}
finally{
    <statement block 3>
}
```

- When executed, `<statement block 1>` will be executed statement by statement. If, at any point, one of the statement in `<statement block 1>` throws an exception, then the rest of the statements in `<statement block 1>` will be discarded and `<statement block 2>` will execute. After all of the statements in `<statement block 1>` have executed, or after all of the statements in `<statement block 2>` have executed, then `<statement block 3>` will execute.
- A simple example is

```
try
{
    Console.WriteLine("Enter a number.");
    Console.WriteLine($"Your number is
 ↪  {int.Parse(Console.ReadLine())}.");
}
catch
```

```
{
    Console.WriteLine("Something was off.");
}
finally
{
    Console.WriteLine("Did it worked?");
}
```

  - If the user enters a string that contains only numbers (say, "10"), then the program will display "Your number is 10." then "Did it worked?".
  - If the user enters a string that does *not* contain only numbers (say, "No"), then the program will display "Something was off." then "Did it worked?".

- Both the **catch** and the **finally** parts of the statement are optional: they can be both present, or only one can occur in the **try** block statement (however, you have to have one or the other).
- A **try** block can have multiple **catch**, as follows:

```
try
{
    Console.WriteLine("Enter a number");
    int uInput = int.Parse(Console.ReadLine());
    Console.WriteLine($"Your number is {uInput}.");
    Console.WriteLine($"Ten divided by your number is
 ↪  {10 / uInput}.");

}
catch (DivideByZeroException)
{
   Console.WriteLine("You tried to divide by zero.");
}
catch (FormatException)
{
    Console.WriteLine("You have tried to convert a
 ↪  string "
      + " containing non-numerical characters to a
       ↪  number.");
}
finally
{
    Console.WriteLine("Did it worked?");
}
```

  - This allows a more fine-grained handling of the exceptions that can be thrown.

  - In the example, if a `DivideByZeroException` exception is thrown, it is because the user entered "0" and the operation `{10 / uInput}` failed. In this case, we can display an appropriate error message ("You tried to divide by zero").
  - In the example, if a `FormatException` exception is thrown, it is because the user entered a string containing non-numerical characters, and we can similarly return an appropriate error message.
  - Writing `catch{…}` is the same as `catch (Exception){…}`: by default, a `catch` block catches all the exceptions that can be thrown, not the exceptions of a particular class. Note that, if specifying multiple `catch` blocks, the order matter, as a `catch (Exception)`, if placed first, will always execute before the `catch` blocks put after.

Exception Class and Objects

- Technically speaking, an exception is an object in a particular class that inherits from the exception class[3].
- We can assign an identifier to it in the `catch` block, to be able to access some of its properties such as the `Message` and a `StackTrace` properties.
- For example, the `IndexOutOfRangeException` object returned when trying to access an array outside of its bound can be named ex and used to display particular information:

```
try
{
    int[] test = new int[10];
    for (int i = 0; i <= test.Length; i++)
    {
        test[i] = i;
    }
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine(ex.Message);
    Console.WriteLine(ex.StackTrace);
}
```

  - When the statement `test[10] = 10;` gets executed, the exception is thrown, named ex, and we display its message

[3] https://learn.microsoft.com/en-us/dotnet/standard/exceptions/exception-class-and-properties

("Index was outside the bounds of the array.") and Stack-Trace ("at Program.Main (System.String() args) (0x0000f) in <path>/Program.cs:<line>", with <path> the path of the Program.cs file, and <line> the line where the error occurs).

Purpose of the `finally` Block

- The difference between

  ```
  try{
      <statement block 1>
  }
  catch{
      <statement block 2>
  }
  finally{
      <statement block 3>
  }
  ```

  and

  ```
  try{
      <statement block 1>
  }
  catch{
      <statement block 2>
  }
  <statement block 3>
  ```

  is that in the second case, `<statement block 3>` may be skipped if `<statement block 1>` or `<statement block 2>` return a value, throw an exception that is not caught, or break the flow of control (using for example **`break`**`;`). In the first case, `<statement block 3>` will *always*[4] get executed, no matter which block gets executed and even if it breaks the control flow or throws another exception.

- For example,

  ```
  static bool GuessGame(string guessP)
  {
      const int valueToGuess = 12;
      try
      {
          int guessV = int.Parse(guessP);
          if (guessV == valueToGuess)
          {
  ```

[4]That is, unless the program crashes or loops forever.

```
                Console.WriteLine("You guessed it!");
                return true;
            }
            else
            {
                Console.WriteLine("Try again!");
                return false;
            }
        }
        catch (FormatException)
        {
            Console.WriteLine("Please, provide a string
    ↪   containing only numbers");
            return false;
        }
        finally
        {
            Console.WriteLine("Thank you for playing!");
        }
    }
```

will always display "Thank you for playing!". If this last statement was *not* in the **finally** block, but was simply inserted after the **try** … **catch** statement, then this message would actually never be displayed.

Scoping in try … catch… finally Statements

- Understanding the scope of statements in **try** … **catch**… **finally** statements can be tricky.
- The general rules are:
  - Variables declared in **try**, **catch** or **finally** blocks will not be accessible outside of them,
  - Variables whose value are set in the **try** block will keep the value they had when the **try** block threw an exception.
- For example, in the following code,

```
int zero = -1;
try
{
    zero = 0;
    int x = 1 / zero;
    zero = 2;
}
catch (DivideByZeroException)
```

```
{
    Console.WriteLine("You tried to divide by " +
 ↪  zero + ".");
    zero = 3;
}
finally
{
    Console.WriteLine("The variable holds " + zero +
 ↪  ".");
}
```

- This program will display

  ```
  You tried to divide by 0.
  The variable holds 3.
  ```

- The variable `x` would not be accessible to the **`catch`** or **`finally`** blocks.
- If we were to remove the `zero = 0;` statement, then the program would display "The variable holds 2.".

When To Use `try` … `catch` and When To Use `TryParse`?

- If something goes wrong in a method, that method can either return some error code or throw an exception.
- Returning an error code means possibly cluttering the signature of the method with some extra parameters, as in the `TryParse` methods.
- `TryParse` is "baking in" a way of signaling that something went wrong because
  1. This type of error is simple, common and predictable,
  2. It decided not to care about *why* the parsing fails (it can be either because the input is **`null`**, because it is not in valid format, or because it produces an overflow).
- However, exceptions can handle those cases differently thanks to different **`catch`** blocks:

  ```
  Console.WriteLine("Test with" +
      "\n\t- nothing (ctrl + d on linux, ctrl + z on
   ↪  windows), " +
      "\n\t- \"No\"," +
      "\n\t-  " + int.MaxValue + "+ 1 = 2147483648.");
  try
  {
      int.Parse(Console.ReadLine());
  ```

```
}
catch (ArgumentNullException)
{
    Console.WriteLine("No argument provided.");
}
catch (FormatException)
{
    Console.WriteLine("The string does not contain
 ↪  only number characters.");
}
catch (OverflowException)
{
    Console.WriteLine("The number is greater than
 ↪  what an integer can store.");
}
```

- So, in summary, `TryParse` is in general better if there is no need to handle the different exceptions differently.

Throwing an Exception

- You can explicitly throw an exception by
  - Creating an `Exception` object,
  - Throwing it, using the **`throw`** keyword.
- For example, the property setter in the following class can explicitly throw an `ArgumentOutOfRangeException` object if we try to create a `Circle` object with a negative diameter:

```
using System;

class Circle
{
  private decimal diameter;
  public decimal Diameter
  {
    get { return diameter; }
    set
    {
      if (value <= 0)
      {
        throw new ArgumentOutOfRangeException();
      }
      else
      {
        diameter = value;
      }
    }
  }

  public Circle(decimal dP)
  {
    Diameter = dP;
  }

  public override string ToString()
  {
    return "Diameter: " + diameter;
  }
}
```

- To use this class properly, every time the `Diameter` value is set (using the set accessor, possibly *via* the constructor), a **`try`** … **`catch`** statement should be used to handle a possible exception, with possibly a loop around it, as follows:

```
using System;

class Program
{
  static void Main(string[] args)
  {
    Circle circle1 = new Circle(1);
    Console.WriteLine(circle1);

    try
    {
      circle1 = new Circle(-10);
      Console.WriteLine("circle1: " + circle1);
    }
    catch (ArgumentOutOfRangeException)
    {
      Console.WriteLine($"Error: value was out of
 ↪  range.");
    }
    Console.WriteLine(circle1);

    bool circle_modified = false;
    do
    {
      try
      {
        Console.WriteLine("Enter the circle new
↪   diameter.");
        int uInput = int.Parse(Console.ReadLine());
        circle1.Diameter = uInput;
        Console.WriteLine(circle1);
        circle_modified = true;
      }
      catch (ArgumentOutOfRangeException)
      {
        Console.WriteLine(
          $"Error: value was out of range."
        );
      }
      catch
      {
        Console.WriteLine("Something went wrong.");
        throw;
      }
    } while (!circle_modified);
  }
}
```

- In the last **`catch`** block above, the **`throw`**`;` (without argument) will pass the exception to the calling environment. It is indeed possible to catch the exception, do something with it (typically, log it or display an error message), and then "pass" that exception to the surrounding environment.