Lesson 29 - More on OOP II

Enum

The **enum** keyword is used to declare an enumeration: a type that consists of a set of named constants called the enumerator list. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. For example, in the following enumeration, Sun is 0, Mon is 1, Tue is 2, and so on:

```
enum Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
```

You can also assign your own enumerator values:

```
enum Days { Sun, Mon, Tue=4, Wed, Thu, Fri, Sat };
```

In the example above, the enumeration will start from 0, then Mon is 1, Tue is 4, Wed is 5, and so on. The value of the next item in an Enum is one increment of the previous value. Note that the values are comma separated. You can refer to the values in the Enum with the dot syntax. In order to assign Enum values to int variables, you have to specify the type in parentheses:

```
int x = (int)Days.Tue;
Console.WriteLine(x);
```

> Basically, Enums define variables that represent members of a fixed set. Some sample Enum uses include month names, days of the week, cards in a deck, etc.

Enums are often used with switch statements:

```
enum TrafficLights { Green, Red, Yellow };
```

```
TrafficLights x = TrafficLights.Red;

switch(x)
{
        case TrafficLights.Green:
                Console.WriteLine("Go");
                break;
```

```
        case TrafficLights.Red:
                Console.WriteLine("Stop!");
                break;
        case TrafficLights.Yellow:
                Console.WriteLine("Caution!");
                break;
        default:
                Console.WriteLine("Error");
                break;
}
```

Exceptions

An **exception** is a problem that occurs during program execution. Exceptions cause abnormal termination of the program. An exception can occur for many different reasons. Some examples:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications.
- Insufficient memory and other issues related to physical resources.

For example, the following code will produce an exception when run because we request an index which does not exist:

```
int[] arr = new int[] { 4, 5, 8 };
Console.WriteLine(arr[8]);
```

> As you can see, exceptions are caused by user error, programmer error, or physical resource issues. However, a well-written program should handle all possible exceptions.

Exception Handling

C# provides a flexible mechanism called the `try-catch` statement to handle exceptions so that a program won't crash when an error occurs. The `try` and `catch` blocks are used similar to:

```
try
{
        int[] arr = new int[] { 4, 5, 8 };
        Console.WriteLine(arr[8]);
}
catch (Exception e)
{
        Console.WriteLine("An error ocurred");
}
```

The code that might generate an exception is placed in the `try` block. If an exception occurs, the catch blocks is executed without stopping the program. The type of exception you want to catch appears in parentheses following the keyword `catch`. We use the general `Exception` type to handle all kinds of exceptions. We can also use the exception object `e` to access the exception details, such as the original error message (`e.Message`):

```
try
{
        int[] arr = new int[] { 4, 5, 8 };
        Console.WriteLine(arr[8]);
}
catch (Exception e)
{
        Console.WriteLine(e.Message);
}
```

> You can also catch and handle different exceptions separately.

Handling Multiple Exceptions

A single `try` block can contain multiple `catch` blocks that handle different exceptions separately. Exception handling is particularly useful when dealing with user input. For example, for a program that requests user input of two numbers and then outputs their quotient, be sure that you handle division by zero, in case your user enters 0 as the second number.

```
int x, y;

try
{
        x = Convert.ToInt32(Console.Read());
        y = Convert.ToInt32(Console.Read());
        Console.WriteLine(x/y);
}
catch (DivideByZeroException e)
{
        Console.WriteLine("Cannot divide by 0");
}
catch (Exception e)
{
        Console.WriteLine("An error ocurred");
}
```

The above code handles the `DivideByZeroException` separately. The last catch handles all the other exceptions that might occur. If multiple exceptions are handled, the Exception type must be defined last. Now, if the user enters 0 for the second number, "Cannot divide by 0"

will be displayed. If, for example, the user enters non-integer values, "An error occurred" will be displayed.

> The following exception types are some of the most commonly used: FileNotFoundException, FormatException, IndexOutOfRangeException, InvalidOperationException, OutOfMemoryException.

finally

An optional `finally` block can be used after the `catch` blocks. The `finally` block is used to execute a given set of statements, whether an exception is thrown or not.

```
int result = 0;
int num1 = 8;
int num2 = 4;

try
{
        result = num1 / num2;
}
catch (DivideByZeroException e)
{
        Console.WriteLine("Error");
}
finally
{
        Console.WriteLine(result);
}
```

> The `finally` block can be used, for example, when you work with files or other resources. These should be closed or released in the `finally` block, whether an exception is raised or not.