

PCP — Lecture 05

Fall 2020 September 7, 2020

Last Time - Operations, conversions and reading from a user

- There are operators for `int`, `float`, `double`, and `decimal`.
- It is possible to convert between those datatypes (sometimes automatically, sometimes explicitly) using “cast” operators.
- `string` has operations too: concatenation and interpolation.
- It is possible to read a `string` from the user.

1 Reading an Int From the User

1.1 Converting a String into an Int

In the `int` class, there is a method called `Parse` that converts a `string` into an `int` if possible and “crashes” otherwise.

For instance, one can use:

```
string test = "32";
int testConversion = int.Parse(test);
```

This converts *the string* "32" into the integer 32 and stores it in the `testConversion` variable, so that the programmer may now treat it like a number (and perform operations on it).

Note that if the string does *not* correspond to a number (e.g. "Hi Mom!"), then the program would ... explode, as the conversion fails. It would simply stop and display an error message.

(In case you are curious, we can also convert an `int` into a string using the `ToString` method, as e.g. in `12.ToString()`.)

1.2 Converting a User input into an int

We can simply combine the `Console.ReadLine()` instruction with the `int.Parse` method to read integers from the user:

```
Console.WriteLine("Please enter the year.");
string answer = Console.ReadLine();
int currentYear = int.Parse(answer);
Console.WriteLine($"Next year we will be in {currentYear + 1}.");
```

We could shorten the previous program by “chaining” the methods:

```
Console.WriteLine("Please enter the year.");
int currentYear = int.Parse(Console.ReadLine());
Console.WriteLine($"Next year we will be in {currentYear + 1}.");
```

Or even, if every line counts and we don't need to access the current year later on in the program:

```
Console.WriteLine("Please enter the year.");
Console.WriteLine($"Next year we will be in {int.Parse(Console.ReadLine()) + 1}.");
```

But, of course, the more that happens on a single line, the more difficult it is to debug it properly.

2 Writing A Class

2.1 Introduction

Let us introduce a couple of key notions for object-oriented programming languages:

- We will be using *classes* and *objects*: A class is the specification (you can think of a cookie cutter, a blueprint, a model), and an object is the “actual thing” (the actual cookie my kid ate, my house, your car, ...).
- To create an object from a class will need to *instantiate* it.
- An object will *be* and *do*, that is, have *data* and *operations*. The class has *attributes* and *methods* (or procedures). When we instantiate, the object has *instance variables*.
- An object hides its structure: to have access to the value of the instance variables, you have to use properties (methods to access those attributes). Classes *encapsulate* the attributes and methods of the object and hides them (and the implementation details) from the other objects.
- Inheritance: a class can inherit from another class, i.e., extend it with new attributes or methods. A class can be extended in different ways, and those extensions can also be extended!

2.2 Code Sample

```
class Rectangle
{
    /*
        A rectangle
        - has a lenght, a width, (attributes)
        - can be given a length, a width, can return its length, its width, and its area
        ↪ (methods).
    */

    // Let us start with the attributes.

    private int length;
    private int width;

    // The key words "public" and "private" are "access modifiers".

    // Now, for the methods.
    // Every method will be of the form:
    // public <return type> <Name>(<type of parameter> <name of parameter>){ <collection of
    ↪ statements>}
```

```

public void SetLength(int lengthParameter)
// Parameters are "local variables".
{
    length = lengthParameter;
}

// This method will simply "take" the argument given, and store it as the length of
↪ the object.
// Of course, a method could perform more advanced operations, like test the value,
↪ change it, compare it against other values, etc.

public int GetLength()
{
    return length;
}

public void Setwidth(int widthParameter)
{
    width = widthParameter;
}
public int GetWidth()
{
    return width;
}

public int ComputeArea()
{
    return length * width;
}
}

```

We will use this class in a separate file which contains a **Main** method that will create a rectangle object and manipulate it.

```

using System;

class Program
{
    static void Main(string[] args)
    {
        Rectangle myRectangle = new Rectangle(); // Instantiation
        myRectangle.SetLength(12); // Calling the classes' object.methodname(argument)
        myRectangle.Setwidth(3);

        Console.WriteLine($"You program's length is {myRectangle.GetLength()} " +
            $"", its width is {myRectangle.GetWidth()} " +
            $"", so its area is {myRectangle.ComputeArea()}.");
    }
}

```

```
    }  
}
```

Sometimes, “Parameters” are called “formal parameters” and “arguments” are called “actual parameters”. Stated differently, a method *has parameters* and *takes arguments*.

2.2.1 Key Notions

- A class has *members* (<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/members>) that can be “data members” (attributes) and “function members” (methods).
- A method can be an “accessor” (if it allows to access the value of an attribute) that either sets (“setters”) or retrieves (“getter”) the value of an attribute. Other kinds of methods, like constructors, will be studied later on.

Some of the new keywords we are using are:

- **public** and **private**, which are access modifiers (everything is private by default).
- **return**, which gives what needs to be returned, according to the return type of the method.
- **new**, which instantiates a class.