CSCI 1301 Book

https://csci-1301.github.io/about#authors

May 21, 2021 (07:33:28 PM)

Contents

1 Introduction to Computers and Programming

1.1 Principles of Computer Programming

- Computer hardware changes frequently - from room-filling machines with punch cards and tapes to modern laptops and tablets
- Capabilities of computers have changed rapidly (storage, speed, graphics, etc.)
- Languages used to program computers have also changed over time
  - Older languages: Fortran, C, C++
  - Newer "compiled" languages: C#, Java, R
  - Newer "interpreted" languages: Python, JavaScript
- This class is about "principles" of computer programming
  - Common principles behind all languages won't change, even though hardware and languages do
  - How to organize and structure data
  - How to express logical conditions and relations
  - How to solve problems with programs

1.2 Programming Language Concepts

- Machine language
  - Computers are made of electronic circuits
  - Basic instructions are encoded by setting wires to “on” or “off”
    - Read data, write data, add, subtract, etc.
  - Binary digits represent on/off state of wires in a circuit
  - Machine language: which sequence of binary digits (circuit state) represents which computer instruction
    - Example instruction: 0010110010101101
  - Most CPUs use one of two languages: x86 or ARM
- Assembly language
  - Easier way for humans to write machine-language instructions
  - Use a sequence of letters/symbols to represent an instruction, instead of 1s and 0s.
    - Example x86 instruction: `movq %rdx, %rbx`
  - **Assembler**: Translates assembly language code to machine instructions
    - One assembly instruction = one machine-language instruction
    - x86 assembly produces x86 machine code
  - Computers can only execute the machine code
- High-level language
  - More human-readable than assembly language
  - Each statement does not need to correspond to a machine instruction
  - Statements represent more “high-level” concepts, such as storing a value in a variable, not “machine-level” concepts like “read these bits from this address”
  - Most languages we program in are high-level (C, C#, Python...)
  - **Compiler**: Translates high-level language to machine code
    - Small programs in high-level language might produce lots of machine code
    - Compiler is specific to both the source language and the target machine code
  - Compile then execute, since computers can only execute machine code
- Compiled vs. Interpreted languages
  - Not all high-level languages use a compiler - some use an interpreter
  - **Interpreter**: Lets a computer “execute” high-level code by translating one statement at a time to machine code
  - Advantage: Less waiting time before you can run the program (no separate “compile” step)
  - Disadvantage: Program runs slower, since you wait for each high-level statement to be translated before the program can continue
- Managed high-level languages (like C#)
  - Combine features of compiled and interpreted languages
  - Compiler translates high-level statements to **intermediate language** instructions, not machine code
    - Intermediate language: Looks like assembly language, but not specific to any CPU
  - **Runtime** executes compiled program by *interpreting* the intermediate language instructions - translates one at a time to machine code
  - Advantages of managed languages:
    - In a “non-managed” language, a compiled program only works on one OS + CPU combination (**platform**) because it is machine code

        * Managed-language programs can be reused on a different platform without recompiling - intermediate language is not machine code and not CPU-specific
        * Still need to write an intermediate language interpreter for each platform (so it produces the right machine code), but in a non-managed language you must write a compiler for each platform
        * Intermediate-language interpreter is much faster than a high-level language interpreter, so programs run faster than an "interpreted language" like Python
    - This still runs slower than a non-managed language (due to the interpreter), so performance-minded programmers use non-managed compiled languages (e.g. for video games)

1.3 Software Concepts

- Flow of execution in a program
    - Program receives input from some source, e.g. keyboard, mouse, data in files
    - Program uses input to make decisions
    - Program produces output for the outside world to see, e.g. by displaying images on screen, writing text to console, or saving data in files
- Program interfaces
    - **GUI** or Graphical User Interface: Input is from clicking mouse in visual elements on screen (buttons, menus, etc.), output is by drawing onto the screen
    - **CLI** or Command Line Interface: Input is from text typed into "command prompt" or "terminal window," output is text printed at same terminal window
    - This class will use CLI because it's simple, portable, easy to work with – no need to learn how to draw images, just read and write text

1.4 Programming Concepts

- Programming workflow (see flowchart)
    - Writing down specifications
    - Creating the source code
    - Running the compiler
    - Reading the compiler's output, warning and error messages
    - Fixing compile errors, if necessary
    - Running and testing the program
    - Debugging the program, if necessary
- Intepreted language workflow (see flowchart)
    - Writing down specifications
    - Creating the source code
    - Running the program in the interpreter
    - Reading the interpreter's output, determining if there is a syntax (language) error or the program finished executing
    - Editing the program to fix syntax errors
    - Testing the program (once it can run with no errors)
    - Debugging the program, if necessary
    - **Advantages**: Fewer steps between writing and executing, can be a faster cycle
    - **Disadvantages**: All errors happen when you run the program, no distinction between syntax errors (compile errors) and logic errors (bugs in running program)

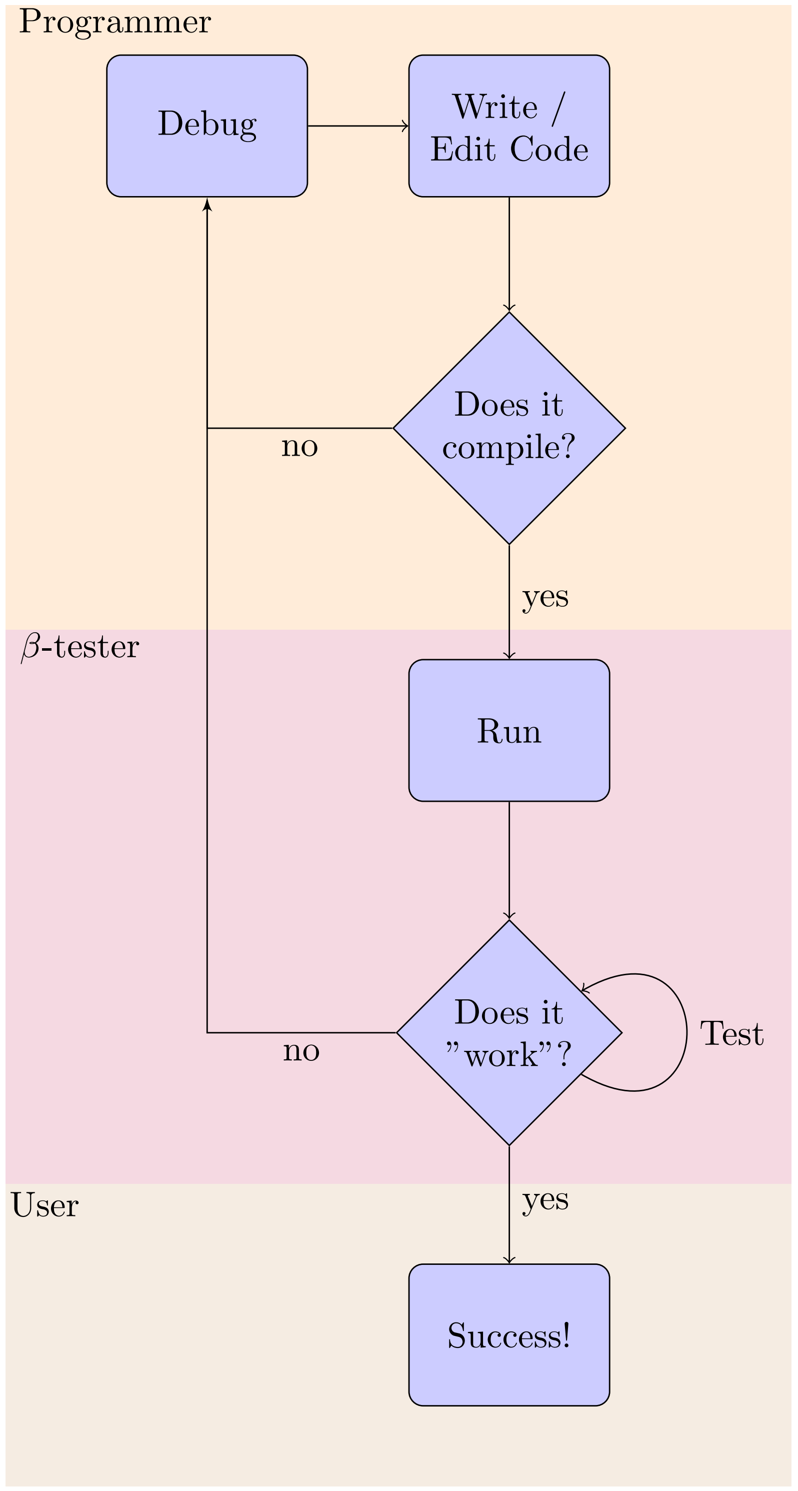

Figure 1: "Flowchart demonstrating roles and tasks of a programmer, beta tester and user in the creation of programs."

1.4.0.1 Programming workflow

- Integrated Development Environment (IDE)
  - Combines a text editor, compiler, file browser, debugger, and other tools
  - Helps you organize a programming project
  - Helps you write, compile, and test code in one place
  - Visual Studio terms:
    * Solution: An entire software project, including source code, metadata, input data files, etc.
    * "Build solution": Compile all of your code
    * "Start without debugging": Run the compiled code
    * Solution location: The folder (on your computer's file system) that contains the solution, meaning all your code and the information needed to compile and run it

2 C# Fundamentals

2.1 Introduction to the C# Language

- C# is a managed language (as introduced in previous lecture)
  - Write in a high-level language, compile to intermediate language, run intermediate language in interpreter
  - Intermediate language is called CIL (Common Intermediate Language)
  - Interpreter is called .NET Runtime
  - Standard library is called .NET Framework, comes with the compiler and runtime
- It is widespread and popular
  - 7th most used language on StackOverflow[1], 5th-most if you discount JavaScript and HTML (which are used for websites, not programs)
  - .NET is the 2nd most used library/framework

2.2 The Object-Oriented Paradigm

- C# is called an "object-oriented" language
  - Programming languages have different *paradigms*: philosophies for organizing code, expressing ideas
  - Object-oriented is one such paradigm, C# uses it
  - Meaning of object-oriented: Program mostly consists of *objects*, which are reusable modules of code
  - Each object contains some data (*attributes*) and some functions related to that data (*methods*)
- Object-oriented terms
  - **Class**: A blueprint or template for an object. Code that defines what kind of data the object will contain and what operations (functions) you will be able to do with that data
  - **Object**: A single instance of a class, containing running code with specific values for the data. Each object is a separate "copy" based on the template given by the class.
  - **Method**: A function that modifies an object. This is code that is defined (written) in the class, but when it runs, it only runs on/for a specific object and modifies that object.
  - **Attribute**: A piece of data stored in an object
- Example objects:

[1](https://insights.stackoverflow.com/survey/2017#technology-programming-languages%3E)

  - “Car” object, represents a car
    * Attributes: Color, wheel size, engine status (on/off/idle), gear position
    * Methods: Press gas or brake pedal, turn key on/off, shift transmission
  - “Audio” object, represents a song being played in a music player
    * Attributes: Sound wave data, current playback position, target speaker device
    * Methods: Play, pause, stop, fast-forward, rewind

3 First Program

Here’s a simple “hello world” program in the C# language:

3.0.0.1 Hello World

```
/* I'm a multi-line comment,
 * I can span over multiple lines!
 */
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, world!"); // I'm an in-line comment.
    }
}
```

Features of this program:

- A multi-line comment: everything between the `/*` and `/*` is considered a *comment*, i.e. text for humans to read. It will be ignored by the C# compiler and has no effect on the program.
- A `using` statement: This imports code definitions from the System *namespace*, which is part of the .NET Framework (the standard library).
  - In C#, code is organized into **namespaces**, which group related classes together
  - If you want to use code from a different namespace, you need a `using` statement to “import” that namespace
  - All the standard library code is in different namespaces from the code you will be writing, so you’ll need `using` statements to access it
- A class declaration
  - Syntax: `class [name of class]`, then { to begin the body of the class, then } to end the body of the class
  - All code between opening { and closing } is part of the class named by the `class [name]` statement
- A method declaration
  - The name of the method is `Main`, and is followed by empty parentheses (we’ll get to those later, but they’re required)
  - Just like with the class declaration, after the name, { begins the body of the method, } ends it
- A statement inside the body of the method
  - This is the part of the program that actually “does something”: It prints a line of text to the console
  - A statement *must* end in a semicolon (the class header and method header aren’t statements)

  - This statement contains a class name (`Console`), followed by a method name (`WriteLine`). It calls the `WriteLine` method in the `Console` class.
  - The **argument** to the `WriteLine` method is the text “Hello, world!”, which is in parentheses after the name of the method. This is the text that gets printed in the console: The `WriteLine` method (which is in the standard library) takes an argument and prints it to the console.
  - Note that the argument to `WriteLine` is inside double-quotes. This means it is a **string**, i.e. textual data, not a piece of C# code. The quotes are required in order to distinguish between text and code.

- An in-line comment: All the text from the `//` to the end of the line is considered a comment, and is ignored by the C# compiler.

3.1 Rules of C# Syntax

- Each statement must end in a semicolon (`;`),
  - Class and method declarations are not statements
  - A method *contains* some statements, but it is not a statement
- All words are case-sensitive
  - A class named `Program` is not the same as one named `program`
  - A method named `writeline` is not the same as one named `WriteLine`
- Braces and parentheses must always be matched
  - Once you start a class or method definition with `{`, you must end it with `}`
- Whitespace – spaces, tabs, and newlines – has almost no meaning
  - There must be at least 1 space between words
  - Spaces are counted exactly if they are inside string data, e.g. `"Hello       world!"`
  - Otherwise, entire program could be written on one line; it would have the same meaning
  - Spaces and new lines are just to help humans read the code
- All C# applications must have a `Main` method
  - Name must match exactly, otherwise .NET runtime will get confused
  - This is the first code to run when the application starts – any other code (in methods) will only run when its method is called

3.2 Conventions of C# Programs

- Conventions: Not enforced by the compiler/language, but expected by humans
  - Program will still work if you break them, but other programmers will be confused
- Indentation
  - After a class or method declaration (header), put the opening `{` on a new line underneath it
  - Then indent the next line by 4 spaces, and all other lines “inside” the class or method body
  - De-indent by 4 spaces at end of method body, so ending `}` aligns vertically with opening `{`
  - Method definition inside class definition: Indent body of method by another 4 spaces
  - In general, any code between `{` and `}` should be indented by 4 spaces relative to the `{` and `}`
- Code files
  - C# code is stored in files that end with the extension “.cs”
  - Each “.cs” file contains exactly one class
  - The name of the file is the same as the name of the class (Program.cs contains `class Program`)

3.3 Reserved Words and Identifiers

- Reserved words: Keywords in the C# language
  - Note they have a distinct color in the code sample and in Visual Studio
  - Built-in commands/features of the language
  - Can only be used for one specific purpose; meaning cannot be changed
  - Examples:
    - `using`
    - `class`
    - `public`
    - `private`
    - `namespace`
    - `this`
    - `if`
    - `else`
    - `for`
    - `while`
    - `do`
    - `return`
- Identifiers: Human-chosen names
  - Names for classes (`Rectangle`, `ClassRoom`, etc.), variables (`age`, `name`, etc.), methods (`ComputeArea`, `GetLength`, etc), namespaces, etc.
  - Some have already been chosen for the standard library (e.g. `Console`, `WriteLine`), but they are still identifiers, not keywords
  - Rules for identifiers:
    - Must not be a reserved word
    - Must contain only letters (`a` → `Z`), numbers (`0` → `9`), and underscore (`_`)– no spaces
    - Must not begin with a number
    - Are case sensitive
  - Conventions for identifiers
    - Should be descriptive, e.g. "`AudioFile`" or "`userInput`" not "`a`" or "`x`"
    - Should be easy for humans to read and type
    - If name is multiple words, use CamelCase[2] (or its variation Pascal case[3]) to distinguish words
    - Class and method names should start with capitals, e.g. "`class AudioFile`"
    - Variable names should start with lowercase letters, then capitalize subsequent words, e.g. "`myFavoriteNumber`"

4 Datatypes

4.1 Datatypes Nomenclature

- Value types
  - Numeric
    - Signed integer (`sbyte`, `short`, *`int`*, `long`)
    - Unsigned integer (`byte`, `ushort`, `uint`, `ulong`)
    - Real number (`float`, *`double`*, `decimal`)
  - Logical (*`bool`*)

[2]https://en.wikipedia.org/wiki/Camel_case
[3]https://www.c-sharpcorner.com/UploadFile/8a67c0/C-Sharp-coding-standards-and-naming-conventions/

    - Character (*char*)

- Reference types
    - String (*string*)
    - Object (object)

(In italics, *the one we will mainly be using.)*

Integers are “whole” numbers (= {…, -1, 0, 1, 2, 3, …}), floating point numbers are real numbers ( ), strings are “text messages”, …

Please refer to the “Datatypes in C#” cheatsheet[4] for more information about datatypes.

4.2 String and Int Variables

Literals are fixed values (“Hi Mom”, 40, 1.2404, …) in the source code.

4.2.0.1 My First Variables

```
using System;

class MyFirstVariables
{
    static void Main()
    {
        // Declaration
        int myAge;
        string myName;

        // Assignment
        myAge = 40;
        myName = "Clément";

        // Displaying
        Console.WriteLine($"I am {myAge} old and my name is {myName}.");
    }
}
```

A variable has a *name* (which must be an identifier), a *type*, a *size*, and a *value*.

Variable Name	Variable Type	Variable Size	Variable Value
`myAge`	`int`	32 bit	`40`
`myName`	`string`	Variable	`"Clément"`

4.3 Variable Initialization

You can declare and assign a variable in one statement using what is called an “initialization statement”.

4.3.0.1 Initialization

[4] https://csci-1301.github.io/datatypes_in_csharp.html

```
string myMessage = "Hey Mom";
int myValue = 12;
```

There is now one additional rule when it comes to chosing a valid identifier for your variable name: you can not take an identifier that was already used. That is, you can have only one variable named `myMessage`: if you want to re-assign a variable, you can not use an initialization again (that would re-declare the variable), you need to use an assignment statement again.

4.4 Remarks

- The value can change (hence the name!) if you re-assign it. The previously stored value is simply wiped out, and lost.
- You can store one variable's value into another, but that value in the other variable won't change when the original variable's value changes:

  ```
  int a = 12;
  int b = a; // b's value is 12
  a = 0; // a's value changed to 0, but b's value is still 12.
  ```

- We can perform basic math operations with numeric datatypes: `+` (sum), `*` (multiplication), `-` (substraction) but also the modulo operation (%), which corresponds to the remainder. More details will be given in lab #3, in homework #2, and during lecture #4.
- There is a difference between

  ```
  int sum = num * 2; // The value of sum is num's value times two
  Console.WriteLine($"{sum}"); // The value of sum is displayed.
  ```

  and

  ```
  Console.WriteLine($"{num * 2}"); // The value of num times two is displayed, but the value of num
  ```

- You can combine multiple declarations, initializations, and even mix both in one statement:

  ```
  int a=0, b, c;  // a, b, c are declared as three int variables, and a's value is set to 0.
  ```

5 Named Constant

A constant is a variable whose value cannot change.

```
const int MONTHS = 12;
const double AVOGADRO = 6.0220e23; // Avogadro Number. Units  1/mol
const double PI = 3.14159265358979;
const double MILES_TO_KM = 1.60934;
```

- Value at to be fixed at declaration (= can only be initialized), and cannot change.
- Name is often ALL CAPS.

For instance, π is defined in the `Math` class and can be accessed as follows:

```
Console.WriteLine(Math.PI);
```

6 Operators

7 Operations on `int` (reminder)

Suppose we are given an `int` variable, `myVar`, with which we can perform the following operations:

Operation	Arithmetic Operator	Algebraic Expression	Expression
Addition	$+$	$x + 7$	`myVar + 7`
Substraction	$-$	$x - 7$	`myVar - 7`
Multiplication	$*$	$x \times 7$	`myVar * 7`
Division	$/$	$x/7$, $x \div 7$	`myVar / 7`
Remainder (a.k.a. modulo)	%	$x \bmod 7$	`myVar % 7`

For the remainder, courtesy of wikipedia[5]:

> For example, the expression "5 mod 2" would evaluate to 1, because 5 divided by 2 has a quotient of 2 and a remainder of 1, while "9 mod 3" would evaluate to 0, because the division of 9 by 3 has a quotient of 3 and a remainder of 0; there is nothing to subtract from 9 after multiplying 3 times 3.

We use = to assing a value to a variable, but we can also use = followed by one of the previous operator to obtain an "augmented assignment operators" or "compound assignment operators":

```
int a = 3, b, c; // Multiple declarations with an assignment.
b = 34 + a;
a = a - 1; // Self-assignment
a -= 1; // Shorthand, this is the same as a = a - 1;
```

8 Implicit and Explicit Conversions Between Numeric Datatypes

As we saw in the cheatsheet[6], we can store e.g. a `float` literal inside a `double` variable (that's an automatic, or *implicit* conversion), but we can not store a `double` literal inside a `float` variable. C# refuses to do this "automatically", because we risk to lose information, but we can "force" it to perform this operation *explicitly* using *cast operators*.

```
float b = 4.7F;
int a = (int) b; // A cast operator is simply the name of the datatype between parenthesis. Here, we co
```

Using casting allows us to go "against" those safe-guards, and can lead to the following complications:

- Storing an imprecise number using a precise datatype (e.g. from `double` to `decimal`).
- Truncating a floating-point number as a "truncated" number (e.g. from `double` to `int`).
- Rounding a precise number to fit a less precise datatype (e.g. from `decimal` to `double`).

Note that *you can*, actually, store a `float` literal inside a `double`, but that you *can not* store a `double` or a `float` literal inside a `decimal`.

[5]https://en.wikipedia.org/wiki/Modulo_operation
[6]https://spots.augusta.edu/caubert/teaching/2020/fall/csci1301/weekly/03/datatypes/index.html#compatibility

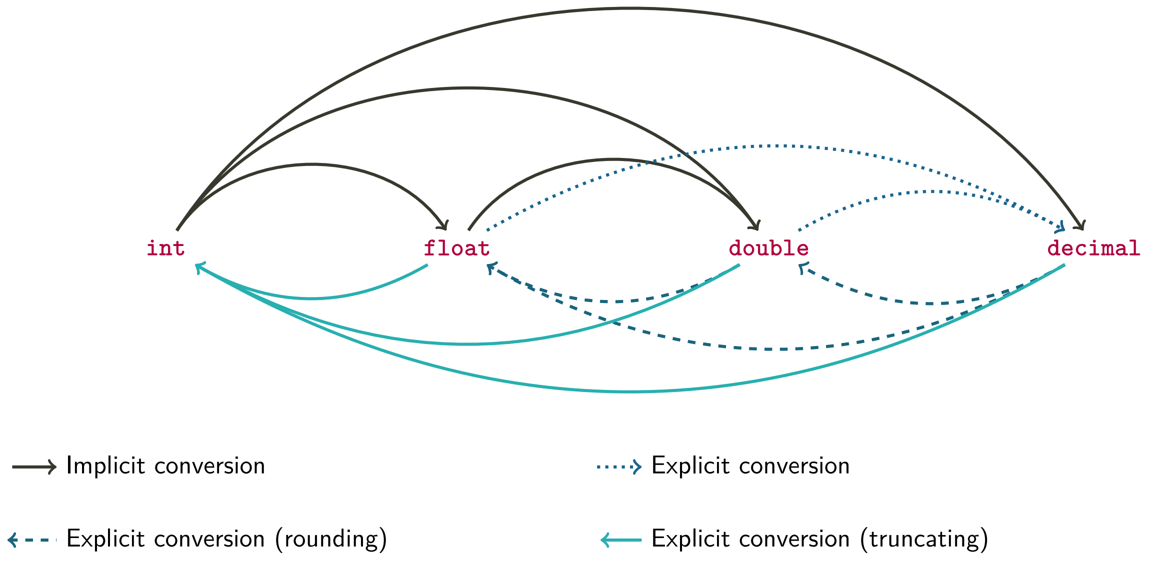
int
float
double
decimal
Implicit conversion
Explicit conversion
Explicit conversion (rounding)
Explicit conversion (truncating)

Figure 2: "Implicit and Explicit Conversion Between Datatypes"

9 Operations on other numerical datatypes

Actually, there is a `+` operator for `float`, `double`, and `decimal`, and the same goes for `-`, `*`, and `%`

Casting can be useful when one wants to divide integers:

```
int pie = 21;
int person = 5;
```

Operation	Result	Type of the multiplication used
`pie / person`	4	`int`
`(float)pie / person`	4.2	`float`
`pie / (float)person`	4.2	`float`
`(float)(pie / person)`	4	`int`

Note that the integer division simply "truncates", and does not round up (that is, `19 / 10` would return 1).

Also note that in "4.2", "4" is the integer, "." (period) is the decimal separator, not to be mixed with "," (comma), the thousands separator.

When performing an operation involving different datatypes, the result type of the operation is the "most precise" datatype if it is allowed (i.e., an implicit conversion can take place), otherwise an error is returned. Refer to the "Result Type of Operations" chart from the cheatsheet[7] for more detail.

10 Increment and Decrement Operators

Increment and decrement operators are used to add or remove one from variables holding numerical values.

	Increment	Decrement
Postfix (after)	a++	a–

11 Operations on Strings

We saw that we could perform interpolation on strings to "fetch" the value of a variable and "insert" it into a string; however, we can use other operations as well. For instance, we can use the + sign for strings as well, to perform an operation called "concatenation":

```
string name = "Bob";
string greetings = $"Welcome, {name}!\nI'll be your guide."; // interpolation

string text = "Hi there" + ", my name is Marie."; // Concatenation. This "join" the strings together.

Console.WriteLine(text); // We can write directly the name of the variable, instead of using interpolat
```

Note that the + sign denotes different operations depending on the type of the operands (`string` or `int`, for instance).

12 Reading, Displaying and Converting Values, Variables, and From the User

12.1 Escape Sequences

Code	Produce
`\n`	A new line[8] (yes, that *is* a character!)
`\t`	A tab character[9] (that is generally interpreted as 4 spaces)
`\"`	Double quotes (")
`\\`	A single backslash (\)

The difference between `Write` and `WriteLine`.

12.2 Reading a String From the User

We can ask *the user* to give their name to the program while it is running as RunTime input. We can use a statement like:

```
string yourFirstName = Console.ReadLine();
```

to store the value entered by the user from the output window in the string variable labeled `yourFirstName`.

12.3 Reading an Int From the User

12.3.1 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:

[8] https://en.wikipedia.org/wiki/Newline
[9] https://en.wikipedia.org/wiki/Tab_key#Tab_characters

```
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()`.)

12.3.2 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.

12.4 Format Specifiers

We can use interpolation to display more nicely numerical values. There are four important format specifiers in C#.

Format specifier	Description
N or n	Formats the string with a thousands separator and a default of two decimal places.
E or e	Formats the number using scientific notation with a default of six decimal places.
C or c	Formats the string as currency. Displays an appropriate currency symbol ($ in the U.S.) next to the number. Separates digits with an appropriate separator character (comma in the U.S.) and sets the number of decimal places to two by default.
P or p	Print percentage

```
Console.WriteLine(
      "\n" + $"{1234.567:N}" // 1,234.57
    + "\n" + $"{1234.5:N}"   // 1,234.50
    + "\n" + $"{1234.567:E}" // 1.234567E+003
```

```
            + "\n" + $"{1234.567:C}" // $1,234.57
            + "\n" + $"{1234.5:C}"   // $1,234.50
            + "\n" + $"{.5:P}"       // 50.00%
    );
```

13 Writing A Class

13.1 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 objet 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!

13.2 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, comp

    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*.

13.2.1 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.

14 Unified Modeling Language

UML is a *specification language* with multiple benefits:

- It is cross-language (you can use it to describe a class written in C#, Java, Python, …),
- It represents only the "surface", and the implementations details are left to programmers,
- It is a language to interact with non-programmer, or with programmers that simply want to use the class without knowing all of its details.

A class is represented as follows:

ClassName
- attribute: `int`
+ SetAttribute(attributeParameter: `int`): `void`
+ GetAttribute(): `int`

Note that `void` is optionnal. For our Rectangle class, this gives:

Rectangle
- width: `int`
- length: `int`
+ SetLength(lengthParameter : `int`): `void`
+ GetLength(): `int`
+ Setwidth(widthParameter: `int`): `void`
+ GetWidth(): `int`
+ ComputeArea(): `int`

15 A Class for ClassRoom

15.1 UML - Specification

ClassRoom
- name: `string`
- number: `int`
+ SetName(nameParameter : `string`): `void`
+ GetName(): `string`
+ SetNumber(numberParameter: `int`): void
+ GetNumber(): `int`

15.2 Implementation

```
using System;

class ClassRoom
{
    private string name;
    private int number;

    public void SetName(string nameParameter)
    {
        name = nameParameter;
    }
    public string GetName()
    {
        return name;
    }

    public void SetNumber(int numberParameter)
    {
        number = numberParameter;
    }
    public int GetNumber()
    {
        return number;
    }
}
```

15.3 Default Values

What if we display the values of the instance variables before setting them?

```
ClassRoom english = new ClassRoom();
Console.WriteLine(english.GetName()); // Nothing!
Console.WriteLine(english.GetNumber()); // 0
```

Indeed, instance variables are different from "usual" variables in that sense that they receive a "default" value when created. This value depends of the variable datatype:

Type	Default
numerical value	`0`
char	`'\x0000'`
bool	`false`
string	`null`

- Note how different it is from the variables we have been using so far, that could not be for instance displayed if their value had not been set.
- We can set a different default value, using, in the class declaration,

```
private string name = "Unknown";
private int number = -1;
```

15.4 Constructors

15.4.1 Custom

A constructor is a method used to create an object. It has to have the same name as the class, and doesn't have a return type.

```
public ClassRoom(string nameParameter, int numberParameter)
{
    name = nameParameter;
    number = numberParameter;
}
```

We use it as follows:

```
ClassRoom math = new ClassRoom("Bertrand", 5);
```

Note:

- the order of the arguments matter,
- the variables, as usual, have a particular scope,
- constructor do not have a return type (not even `void`)

In the UML diagram, we would add:

+ «constructor» ClassRoom(nameParameter: string, numberParameter: int)

Note that we could skip the «constructor» part, can you tell why?

15.4.2 Default

If we implement this constructor, then we lose the "No args", default constructor

```
public ClassRoom() { }
```

We can re-define it, using something like:

```
public ClassRoom() {
    name = "Unknown";
    int = -1;
}
```

16 Signature and Overloading

Every method has a signature made of - its name, - its parameters types (but not the parameter names).

Note that the return type is not part of the method signature in C#.

In a class, all the methods need to have a different signature. You cannot, for example, have these two methods in the same class:

```
int DoSomething(int a, int b);
string DoSomething(int c, int d);
```

It is possible, however, to have two methods with the same name, as long as they have different signatures. If we are in such a situation, then we say that we are *overloading*. We will look at examples of overloading in lab.

17 ToString

A particular method can be used to display information about our objects. It is called `ToString`, and can be defined as follows:

```
public override string ToString()
{
    return "Person: " + Name + " " + Age;
}
```

18 Boolean and Conditions

A condition is either true or false. We can store if something is true or false in a (boolean) *flag*, which is simply a variable of type boolean.

We can declare, assign, initialize and display it as any other variable:

```
bool flag = true;
Console.WriteLine(true);
```

But the only two possible values are `true` and `false`, and we will study three operations on them: "and" (`&&`, the conjonction), "or" (`||`, the disjunction) and "not" (`!`, the negation). They have the expected meaning that the condition "A and B" is true if and only if A is true, and B is true. Similarly, "A or B" is false if and only if A is false, and B is false (that is, it takes only one to make their disjunction true).

We present this behavior with *truth tables*, as follows:

`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

`!true`	`false`
`!false`	`true`

We could also have represented those tables in 2-dimensions, and will do so in lab.

19 Equality and Relational Operators

Equality Operators

Mathematical Notation	C# Notation	Example
$=$	`==`	`3 == 4` → `false`
$\neq$	`!=`	`3!=4` → `true`

We test numerical value for equality, as well as `string`, `char` and `bool`!

```
Console.WriteLine(3 == 4);
Console.WriteLine(myStringVar == "Train");
Console.WriteLine(myCharVar == 'b');
```

We can also test if a value is greater than another, using the following *relational* operators.

Relational Operators

Mathematical Notation	C# Notation	Example
$>$	`>`	`3 > 4` → `false`
$<$	`<`	`3 < 4` → `true`
$\geqslant$	`>=`	`3 >= 4` → `false`
$\leqslant$	`<=`	`3 <= 4` → `true`

We can also compare `char`, but the order is a bit complex (you can find it, for instance, at https://stackoverflow.com/a/14967721/).

The precedence, that we will study in lab, is as follows:

`! (* / %) (+ -) (< > <= >=) (== !=) && ||`

20 if Statement

20.1 First Example

```
Console.WriteLine("Enter your age");
int age = int.Parse(Console.ReadLine());
if (age >= 18)
{
    Console.WriteLine("You can vote!");
}
```

The idea is that the statement `Console.WriteLine("You can vote!");` is exectued only if the condition `(age >= 18)` evaluates to `true`. Otherwise, that statement is simply "skipped".

20.2 Syntax

```
if (<condition>)
{
    <statement block>
}
```

Please observe the following.

- `<Condition>` is something that evaluates to a `bool`. For instance, having a number like in `if(3)` would not compile.
- Note the absence of semicolon after `if (<condition>)`.
- The curly braces can be removed if the statement block is just one statement.
- The following statements (that is, after the `}` that terminates the body of the `if` statement) are executed in any case.

21 if-else Statements

21.1 Syntax

```
if (<condition>)
{
    <statement block 1>
}
else
{
    <statement block 2>
}
```

With `if-else` statements, the idea is that the statement block 1 is exectued only if the condition evaluates to `true`, and that the statement block 2 is exectued only if the condition evaluates to `false`. Note that since a condition is always either true or false, we know that at least one of the block will be executed, and since a condition cannot be true and false at the same time, at most one block will be executed: hence, exactly one block will be executed.

22 Nested if-else Statements

`<statement block>` can actually be an `if-else` statement itself!

```
bool usCitizen = true;
int age = 19;

if (usCitizen == true)
{
    if (age > 18)
    {
        Console.WriteLine("You can vote!");
    }
    else
    {
        Console.WriteLine("You are too young!");
    }
}
else
{
    Console.WriteLine("Sorry, only citizens can vote");
}
```

Note that

- There is a simpler way to write `usCitizen == true`: simply write `usCitizen`!

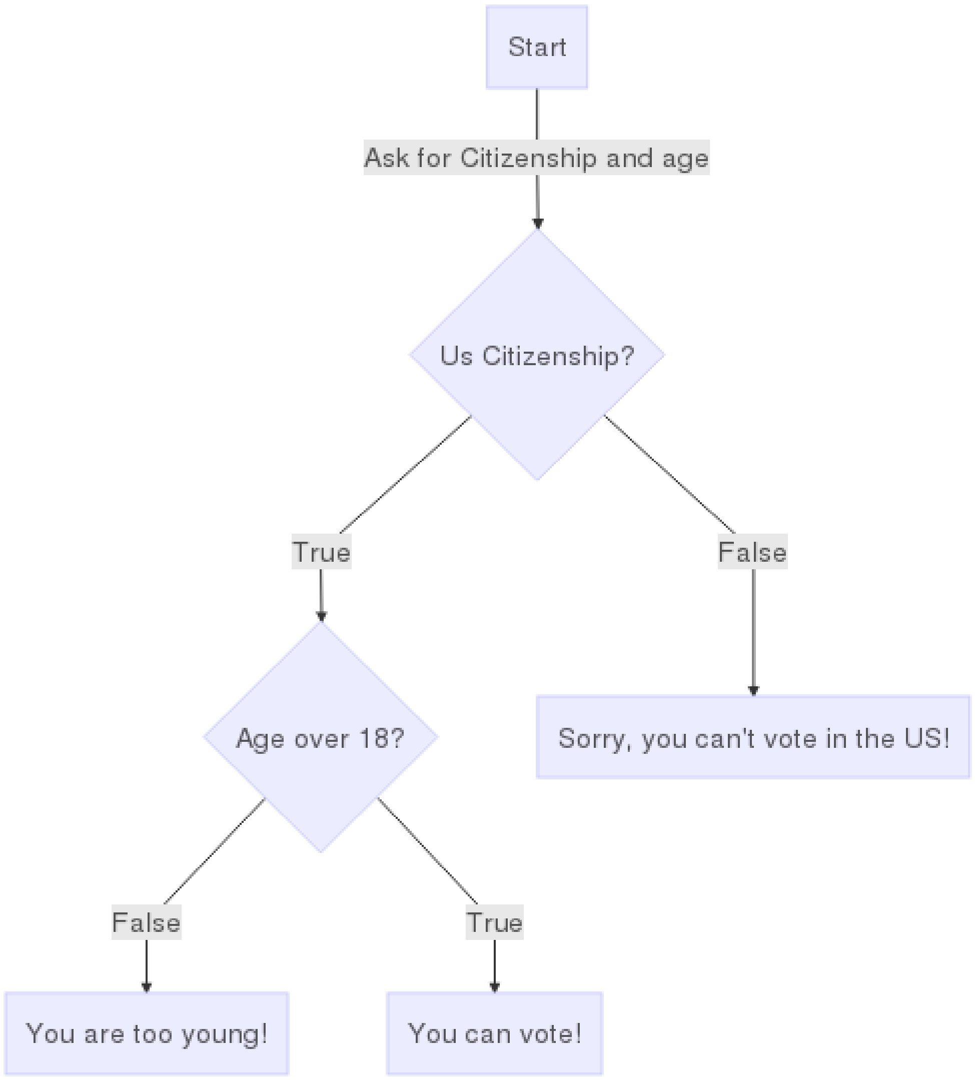

Figure 3: “A flowchart representation of the nested if-else statement”

- We could remove the braces
- We could have a similar flavor with only if: `if(age > 18 && usCitizen) … else …`, but the messages would be less accurate.

23 if-else-if Statements

```
if (<condition 1>)
{
    <statement block> // Executed if condition 1 is true
}
else if (<condition 2>)
{
   <statement block> // Executed if condition 1 is false and condition 2 is true
}
...
else if (<condition N>)
{
    <statement block> // Executed if all the previous conditions are false and condition N is true
}
else
{
    <statement block>  // Executed if all the conditions are false
}
```

Note that the conditions could be really different, not even testing the same thing!

23.1 Example

We can make an example with really different conditions, not overlaping:

```
if (age > 12)
    x = 0;
else if (charVar == 'c')
    x = 1;
else if (boolFlag)
    x = 2;
else
    x = 3;
```

Giving various values to age, charVar and boolFlag, we will see which value would `x` get in each case.

24 ?: Operator

There is an operator for `if else` statements for particular cases (assignment, call, increment, decrement, and new object expressions):

`condition ? first_expression : second_expression;`

```
int price = adult ? 5 : 3;
```

We will have a brief look at it if time allows, otherwise you can read about it at https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-operator.

25 if-else-if Statements

```
if (<condition 1>)
{
    <statement block 1> // Executed if condition 1 is true
}
else if (<condition 2>)
{
   <statement block 2> // Executed if condition 1 is false and condition 2 is true
}
...
else if (<condition N>)
{
    <statement block N> // Executed if all the previous conditions are false and condition N is true
}
else
{
    <statement block N+1>  // Executed if all the conditions are false
}
```

Note that the conditions could be really different, not even testing the same thing!

25.1 Example

We can make an example with really different conditions, not overlaping:

```
if (age > 12)
    x = 0;
else if (charVar == 'c')
    x = 1;
else if (boolFlag)
    x = 2;
else
    x = 3;
```

Try to give various values to `age`, `charVar` and `boolFlag`, and see which value would `x` get in each case.

26 Boolean Flags

Remember that a boolean *flag* is a boolean variable? We can use it to "store" the result of an interaction with a user.

Assume we want to know if the user work full time at some place, we could get started with:

```
Console.WriteLine("Do you work full-time here?");
char ch = Console.ReadKey().KeyChar; // Note that here, passing by, we are using a new method, to read

if (ch == 'y' || ch == 'Y')
     Console.WriteLine("Answered Yes");
else if (ch == 'n' || ch == 'N')
     Console.WriteLine("Answered No");
else
     Console.WriteLine("Said what?");
```

But we can't accomodate this 3-party situation (you either work here full-time, or you don't), so we can change the behaviour to

```
if (ch == 'y' || ch == 'Y')
     Console.WriteLine("Answered Yes");
else
     Console.WriteLine("Answered No");
```

We'll study *user input validation*, that allows to get better answers from the users, later on.

But imagine we are at the beginnig of a long form, and we will need to re-use that information multiple times. With this previous command, we would need to duplicate all our code in two places. Instead, we could "save" the result of our test in a boolean variable, like so:

```
bool fullTime;
if (ch == 'y' || ch == 'Y')
    fullTime = true;
else
    fullTime = false;
```

If you looked at the `?` operator in lab, you can even shorten that statement to:

```
fullTime = (ch == 'y' || ch == 'Y') ? true : false;
```

Why stop here? We could even do

```
fullTime = (ch == 'y' || ch == 'Y');
```

Tada! We went from a long, convoluted code, to a very simple line! We already did this trick last time, but I thought that seeing it again would help.

27 Constructing a Value Progressively

In lab, last time, you were asked the following:

> Ask the user for an integer, and display on the screen "positive and odd" if the number is positive and odd, "positive and even" if the number is positive and even, "negative and odd" if the number is negative and odd, "negative and even" if the number is negative and even, and "You picked 0" if the number is 0.

A possible anwer is:

```
int a;
Console.WriteLine("Enter an integer");
a = int.Parse(Console.ReadLine());
if (a >= 0)
{
    if (a % 2 == 0)
        Console.WriteLine("Positive and even");
    else // if (a % 2 != 0)
        Console.WriteLine("Positive and odd");
}
else
{
    if (a % 2 == 0)
        Console.WriteLine("Negative and even");
    else
```

```
        Console.WriteLine("Negative and odd");
}
```

That is a lot of repetition! We could actually construct "progressively" the message we will be displaying:

```
string msg;
if (a >= 0)
{
    msg = "Positive";
}
else
{
    msg = "Negative";
}
if (a % 2 == 0)
    msg += " and even";
else // if (a % 2 != 0)
    msg += " and odd";
```

Much better! Since the two conditions are actually independant, we can test them in two different `if` statements!

28 Switch Statements

`switch` statements allow to simplify the "matching" of a value against a pre-determined set of values. Its formal syntax is as follows:

```
switch (<variable name>)
{
    case (<literral 1>):
        <statement block 1>
        break;
    case (<literal 2>):
        <statement block 2>
        break;
    ...
    default:
        <statement block n>
        break;
}
```

The (…) are mandatory, the {…} are optional.

- All the literals need to be different.
- The literal and the variable have to be of the same type.
- You can't have case(<variable name>)

For instance, imagine we want to go from a month's number to its name. We could do that with an `if…else if …`:

```
int month = 11;
string monthname;
if (month == 1) monthname = "January";
else if (month == 2) monthname = "February";
// ...
```

```
else if (month == 12) monthname = "December";
else monthname = "Error!";
```

But since we know that "month" will be a value between 1 and 12, or else we have an error, we could also have:

```
switch (month)
{
    case (1):
        monthname = "January";
        break;
    case (2):
        monthname = "February";
        break;
    // ..
    case (12):
        monthname = "December";
        break;
    default:
        monthname = "Error!";
        break;
}
```

Another example, to match a section letter against 4 possibilities, where two actually result in the same behaviour:

```
char section = 'c';
string meet;
switch (section)
{
    case ('a'):
        meet = "MW 1-2PM";
        break;
    case ('b'):
        meet = "TT 1-2PM";
        break;
    case ('c'):
    case ('d'):
 //    case ('a'): Would not compile!
        meet = "F 2-4PM";
        break;
    default:
        meet = "Invalid code";
        break;
}
```

29 Definition and First Example of while loops

29.1 Formal Syntax

```
while (<condition>)
{
    <statement block>
}
```

29.2 Example

```
int number = 0;
while (number <=5)
{
    C.WL("Hi Mom!);
    C.WL(number);
    number++;
}
```

Notes:

- If <`condition`> is `false`: 0 execution of the body.
- If <`condition`> is always true: program loop for ever!
- The conditions under which <`condition`> changes should be given a chance to change in the body of the loop!

30 Five Ways Things Can Go Wrong

It is easy to write *wrong* `loop` statements. Let us review some of the “classic” blunders.

30.1 Failling to update the variable occuring in the condition

```
int number = 0;
while (number <=5)
{
    C.WL("Hi Mom!);
    C.WL(number);
}
```

Number isn’t changed!

30.2 Updating the “wrong” value

```
int number1, number = 0;
while (number <=5)
{
    C.WL("Hi Mom!);
    C.WL(number);
    number1++;
}
```

30.3 Having an empty body

```
int number = 0;
while (number <=5); // Note the semi-colon here!
{
    C.WL("Hi Mom!");
    C.WL(number);
    number++;
}
```

30.4 Having an empty body (variation)

```
int number = 0;
while (number <=5)
    C.WL("Hi Mom!);
    C.WL(number);
    number++;
```

30.5 Going in the wrong direction

```
int number = 0;
while (number >=5)
{
    C.WL("Hi Mom!);
    C.WL(number);
    number++;
}
```

The variable `number` should be decremented, not incremented.

31 User-Input Validation

We can use loops to test what was entered by the user, and ask again if the value does not fit our needs:

```
Console.WriteLine("Please enter a positive number");
int n = int.Parse(Console.ReadLine());
while (n < 0)
{
    Console.WriteLine($"You entered {n}, I asked you for a positive number. Please try again.");
    n = int.Parse(Console.ReadLine());
}
```

32 Vocabulary

Variables and values can have multiple roles, but it is useful to mention three different roles in the context of loops:

Counter Variable that is incremented every time a given event occurs.

```
int i = 0; // i is a counter
while (i < 10){
    Console.WriteLine($"{i}");
    i++;
}
```

Sentinel Value A special value that signals that the loop needs to end.

```
Console.WriteLine("Give me a string.");
string ans = Console.ReadLine();
while (ans != "Quit") // The sentinel value is "Quit".
{
    Console.WriteLine("Hi!");
    Console.WriteLine("Enter \"Quit\" to quit, or anything else to continue.");
    ans = Console.ReadLine();
}
```

Accumulator Variable used to keep the total of several values.

```
int i = 0, total = 0;
while (i < 10){
    total += i; // total is the accumulator.
    i++;
}

Console.WriteLine($"The sum from 0 to {i} is {total}.");
```

We can have an accumulator and a sentinel value at the same time:

```
Console.WriteLine("Enter a number to sum, or \"Done\" to stop and print the total.");
string enter = Console.ReadLine();
int sum = 0;
while (enter != "Done")
{
    sum += int.Parse(enter);
    Console.WriteLine("Enter a number to sum, or \"Done\" to stop and print the total.");
    enter = Console.ReadLine();
}
Console.WriteLine($"Your total is {sum}.");
```

You can have counter, accumulator and sentinel values at the same time!

```
int a = 0;
int sum = 0;
int counter = 0;
Console.WriteLine("Enter an integer, or N to quit.");
string entered = Console.ReadLine();
while (entered != "N") // Sentinel value
{
    a = int.Parse(entered);
    sum += a; // Accumulator
    Console.WriteLine("Enter an integer, or N to quit.");
    entered = Console.ReadLine();
    counter++; // counter
}
Console.WriteLine($"The average is {sum / (double)counter}");
```

We can distinguish between three “flavors” of loops (that are not mutually exclusive):

Sentinel controlled loop The exit condition test if a variable has (or is different from) a specific value.

User controlled loop The number of iteration depends on the user.

Count controlled loop The number of iteration depends on a counter.

Note that a user-controlled loop can be sentinel-controlled (that is the example we just saw), but also count-controlled (“Give me a value, and I will iterate a task that many times”).

33 More Input Validation, Using TryParse

The TryParse method is a complex method that will allow us to parse strings, and to "extract" a number out of them if they contain one, or to be given a way to recover if they don't.

```
Console.WriteLine("Please, enter an integer.");
string message = Console.ReadLine();
int a;
bool res = int.TryParse(message, out a);
if (res)
    {
        Console.WriteLine($"The value entered was an integer: {a}.");
    }
else
    {
        Console.WriteLine("The value entered was not an integer, so 0 is assigned to a.");
    }
Console.WriteLine(a);
```

As you can see, `int.TryParse` takes two arguments, a string and a variable name (prefixed by the "magic" novel keyword `out`) and returns a boolean. You will get a chance to experiment with this code in lab.

34 While Loop With Complex Conditions

```
int c;
string message;
int count;
bool res;

Console.WriteLine("Please, enter an integer.");
message = Console.ReadLine();
res = int.TryParse(message, out c);
count = 0; // The user has 3 tries: count will be 0, 1, 2, and then we default.
while (!res && count < 3)
{
    count++;
    if (count == 3)
        {
        c = 1;
        Console.WriteLine("I'm using the default value 1.");
        }
    else
        {
        Console.WriteLine("The value entered was not an integer.");
        Console.WriteLine("Please, enter an integer.");
        message = Console.ReadLine();
        res = int.TryParse(message, out c);
        }
}
Console.WriteLine("The value is: " + c);
```

35 Combining Methods and Decision Structures

Note that we can have a decision structure inside a method! If we were to re-visit the Rectangle class, we could have a constructor of the following type:

```
public Rectangle(int wP, int lP)
    {
        if (wP <= 0 || lP <= 0)
        {
            Console.WriteLine("Invalid Data, setting everything to 0");
            width = 0;
            length = 0;
        }
        else
        {
        width = wP;
        length = lP;
    }
}
```

36 Putting it all together!

```
using System;

class Loan
{
    private string account;
    private char type;
    private int cscore;
    private decimal amount;
    private decimal rate;

    public Loan()
    {
        account = "Unknown";
        type = 'o';
        cscore = -1;
        amount = -1;
        rate = -1;
    }

    public Loan(string nameP, char typeP, int cscoreP, decimal needP, decimal downP)
    {
        account = nameP;
        type = typeP;
        cscore = cscoreP;
        if (cscore < 300)
        {
            Console.WriteLine("Sorry, we can't accept your application");
            amount = -1;
            rate = -1;
        }
        else
        {
            amount = needP - downP;

            switch (type)
            {
                case ('a'):
                    rate = .05M;
                    break;

                case ('h'):
                    if (cscore > 600 && amount < 1000000M)
                        rate = .03M;
                    else
                        rate = .04M;
                    break;
                case ('o'):
                    if (cscore > 650 || amount < 10000M)
                        rate = .07M;
                    else
                        rate = .09M;
                    break;

            }

        }
    }
    public override string ToString()
    {
        string typeName = "";
        switch (type)
        {
            case ('a'):
                typeName = "an auto";
                break;

            case ('h'):
                typeName = "a house";
                break;
            case ('o'):
                typeName = "another reason";
                break;

        }

        return "Dear " + account + $", you borrowed {amount:C} at {rate:P} for "
            + typeName + ".";
    }
}
```

```
using System;
class Program
{
    static void Main()
    {

        Console.WriteLine("What is your name?");
        string name = Console.ReadLine();

        Console.WriteLine("Do you want a loan for an Auto (A, a), a House (H, h), or for some Other (O,
        char type = Console.ReadKey().KeyChar; ;
        Console.WriteLine();

        string typeOfLoan;

        if (type == 'A' || type == 'a')
        {
            type = 'a';
            typeOfLoan = "an auto";
        }
        else if (type == 'H' || type == 'h')
        {
            type = 'h';
            typeOfLoan = "a house";
        }
        else
        {
            type = 'o';
            typeOfLoan = "some other reason";
        }

        Console.WriteLine($"You need money for {typeOfLoan}, great.\nWhat is your current credit score?
        int cscore = int.Parse(Console.ReadLine());

        Console.WriteLine("How much do you need, total?");
        decimal need = decimal.Parse(Console.ReadLine());

        Console.WriteLine("What is your down paiement?");
        decimal down = decimal.Parse(Console.ReadLine());

        Loan myLoan = new Loan(name, type, cscore, need, down);
        Console.WriteLine(myLoan);
    }

}
```

37 Arrays

37.1 Motivation

Arrays are collection, or grouping, of values held in a single place. They can store multiple values of the same datatype, and are useful, for instance,

- When we want to store a collection of related values,
- When we don't know in advance how many variables we need.

37.2 Declaration and Initialization of Arrays

Declaration and assignment

```
int[] myArray;
myArray = new int[3]; // 3 is the size declarator
// We can now store 3 ints in this array,
// at index 0, 1 and 2

myArray[0] = 10; // 0 is the subscript, or index
myArray[1] = 20;
myArray[2] = 30;

// the following would give an error:
//myArray[3] = 40;
// Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array at P
// "Array bound checking": happen at runtime.
```

As usual, we can combine declaration and assignment on one line:

```
int[] myArray = new int[3];
```

We can even initialize *and* give values on one line:

```
int[] myArray = new int[3] { 10, 20, 30 };
```

And that statement can be rewritten as any of the following:

```
int[] myArray = new int[] { 10, 20, 30 };
int[] myArray = new[] { 10, 20, 30 };
int[] myArray = { 10, 20, 30 };
```

But, we should be carefull, the following would cause an error:

```
int[] myArray = new int[5];
myArray = { 1, 2 ,3, 4, 5}; // ERROR
```

If we use the shorter notation, we *have to* give the values at initialization, we cannot re-use this notation once the array was created.

Other datatype, and even objects, can be stored in arrays:

```
string[] myArray = { "Bob", "Mom", "Train", "Console" };
Rectangle[] arrayOfRectangle = new Rectangle[5];
```

37.3 Custom Size and Values

```
Console.WriteLine("What is size of the array that you want?");
int size = int.Parse(Console.ReadLine());
int[] customArray = new int[size];
```

How can we fill it with values, since we do not know its size? Using iteration!

```
int counter = 0;
while (counter < size)
{
    Console.WriteLine($"Enter the {counter + 1}th value");
    customArray[counter] = int.Parse(Console.ReadLine());
    counter++;
}
```

We can use `length`, a property of our `array`. That is, the integer value `myArray.Length` is the length (= size) of the array, we can access it directly.

To display an array, we need to iterate as well (this time using the `Length` property):

```
int counter2 = 0;
while (counter2 < customArray.Length)
{
    Console.WriteLine($"{counter2}: {customArray[counter2]}.");
    counter2++;
}
```

37.4 Changing the Size

`Array` is actually a class, and it comes with methods!

```
Array.Resize(ref myArray, 4);
myArray[3] = 40;
Array.Resize(ref myArray, 2);
```

`Resize` shrinks (and content is lost) and extends (and store the default value, i.e., 0 for `int`, etc.)!

38 For Loops

38.1 `for` Loops

```
int i = 0;
while (i <= 5)
{
    Console.Write(i + " ");
    i++;
}
```

```
int j = 0;
do
{
    Console.Write(j + " ");
    j++;
} while (j <= 5);
```

```
int k = 0;
for (k = 0; k <= 5; k++)
{
    Console.Write(k + "");
}
```

```
for (int l = 0; l <= 5; l++)
{
    Console.Write(l + "");
}
```

Structure : initialization / condition / update

38.2 Ways Things Can Go Wrong

Don't:

- Increment the counter in the body of the for loop!
- Assume that a variable declared in the header of a for loop will be accessible in the rest of the code. / Use `for` if you want to use the counter for anything else.
- Declare the variable twice.

38.3 For loops With Arrays

`for` loops actually go very well with arrays:

```
for (int i = 0; i < size; i++)
{
    Console.WriteLine($"Enter the {i + 1}th value");
    customArray[i] = int.Parse(Console.ReadLine());
}
```

Remember that we can use the `Length` property of our `array`. The previous code could become (only the first line changed):

```
for (int i = 0; i < customArray.Length; i++)
{
    Console.WriteLine($"Enter the {i + 1}th value");
    customArray[i] = int.Parse(Console.ReadLine());
}
```

38.4 Nested Loops

Of course, exactly as we could nest `if` statements, we can nest looping structures!

```
for (int o = 0; o < 11; o++)
{
    for (int p = 0; p < 11; p++)
        Console.Write($"{o} × {p} = {o * p}  \t ");
    Console.Write();
}
```

38.5 Mixing Control Flows

And we can use `if` statements in the body of `for` loops:

```
for (int m = 0; m < 10; m++)
{
    if (m % 2 == 0) Console.WriteLine("This is my turn.");
    else Console.WriteLine("This is your turn.");
}
```

38.6 Iterations

There is another, close, structrure that allows to iterate over the elements of an array, but can only access them, not change their values (they are "read only").

```
for (int i = 0; i < myArray.Length; i++)
    Console.Write(myArray[i] + " ");

foreach (int i in myArray) // "Read only"
    Console.Write(i + " ");
```

Diffference is w.r.t. to modifying the array "read Vs write". Having `i = 2` in the `foreach` would cause an error!

That last structure is given for the sake of completeness, but it's ok if you'd rather not use it.

39 Static

When we write:

```
Console.WriteLine(Math.PI);
```

The `Math` actually refers to *a class*, and not to *an object*. How is that?

Actually, everything in the `MATH` class is *static* (`public static class Math`), and the `PI` constant is actually public! (`public const double PI`).

Class attribute: can be static or not, public or private, a constant or variable.

```
public const double PI = 3.14159265358979;
```

We also have static methods:

```
Math.Min(x,y);
Math.Max(x,y);
Math.Pow(x,y);
```

A static member (variable, method, etc) belongs to the type of an object rather than to an instance of that type.

39.1 Static Class Members

Class member = methods and fields (attributes)

Motivation: the methods we are using the most (`WriteLine`, `ConsoleRead`) are static, but all the methods we are writting are not (they are "non-static", or "instance").

Static Method	Non-static Method
`ClassName.MethodName(arguments)`	`ObjectName.MethodName(arguments)`
`Math.Pow(2, 5)` ($2^5 = 32$)	`myRectangle.SetLength(5)`

A static class member is associated with the **class** instead of **with the object**.

\	Static Field	Non-static Field
Static method	OK	NO
Non-static method	OK	OK

40 A Static Class for Arrays

```
using System;
    static class Lib
    {
        public static int ValueIsIndex(int[] arrayP)
    {
        int res = 0;
        for (int i = 0; i < arrayP.Length; i++)
            if (arrayP[i] == i) res++;
        return res;
    }

    public static bool AtLeastOneValueIsIndex(int[] arrayP)
    {
        return (ValueIsIndex(arrayP) > 0);
    }

    public static int ValueMatch(int[] arrayP1, int[] arrayP2)
    {
        int res = 0;
        int smallestSize;
        if (arrayP1.Length < arrayP2.Length) smallestSize = arrayP1.Length;
        else smallestSize = arrayP2.Length;
        for (int i = 0; i < smallestSize; i++)
            if (arrayP1[i] == arrayP2[i]) res++;
        return res;
    }
}
```

```
using System;
    class Program
    {
        static void Main(string[] args)
        {
        int[] arrayA = {0, 3, 5, 12, 4, 5, 8 };
        Console.WriteLine(Lib.ValueIsIndex(arrayA));
        Console.WriteLine(Lib.AtLeastOneValueIsIndex(arrayA));

        int[] arrayB = {3, 5, 4, 12, 5, 8 };
```

```
            Console.WriteLine(Lib.ValueIsIndex(arrayB));
            Console.WriteLine(Lib.AtLeastOneValueIsIndex(arrayB));

            Console.WriteLine(Lib.ValueMatch(arrayA, arrayB));
            }
        }
```