CSCI 1301 Book

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

June 7, 2021 (12:40:28 PM)

Table of Contents

# Introduction to Computers and Programming

## Principles of Computer Programming

* Computer hardware changes frequently - from room-filling machines with punch cards and tapes to modern laptops and tablets
* With these changes - the capabilities of computers increase rapidly (storage, speed, graphics, etc.)
* Computer programming languages also change
  + Better programming language theory leads to new programming techniques
  + Improved programming language implementations
  + New languages are created - old ones updated
* There are hundreds of programming languages, why?
  + Different tools for different jobs
    - Some languages are better suited for certain jobs
    - Python for scripting, Javascript for web pages
  + Personnel preference and popularity
* 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

## Programming Language Concepts

* Machine language
  + Computers are made of electronic circuits
    - Circuits are components connected by wires
    - Some wires carry data - e.g. numbers
    - Some carry control signals - e.g. do an add or a subtract operation
  + Instructions are settings on these control signals
    - A setting is represented as a 0 or 1
    - A machine language instruction is a group of settings - For example: 1000100111011000
  + Most CPUs use one of two languages: x86 or ARM
* Assembly language
  + Easier way for humans to write machine-language instructions
  + Instead of 1s and 0s, it uses letters and “words” to represent an instruction.
    - Example x86 instruction: MOV BX, AX which makes a copy of data stored in a component called AX and places it in one called BX
  + **Assembler**: Translates assembly language instructions to machine language instructions
    - For example: MOV BX, AX translates into 1000100111011000
    - One assembly instruction = one machine-language instruction
    - x86 assembly produces x86 machine code
  + Computers can only execute the machine code
* High-level language
  + Hundreds including C#, C++, Java, Python, etc.
  + Most programs are written in a high-level language since:
    - More human-readable than assembly language
    - High-level concepts such as processing a collection of items are easier to write and understand
    - Takes less code since each statement might be translated into several assembly instructions
  + **Compiler**: Translates high-level language to machine code
    - Finds “spelling” errors but not problem-solving errors
    - Incorporates code libraries – commonly used pieces of code previously written such as Math.Sqrt(9)
    - Optimizes high-level instructions – your code may look very different after it has been optimized
    - Compiler is specific to both the source language and the target computer
  + Compile high-level instructions into machine code then run 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 the high-level statements to be translated then the program is run
* 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 by *interpreting* the intermediate language instructions - translates one at a time to machine code
    - faster since translation step is partially done and only its last step is done when running the program
  + 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, for a non-managed language, you must write a compiler for each platform
    - Writing a compiler is more complicated and more work than writing an interpreter thus an interpreter is a quicker (and cheaper) way to put your language on different platforms
    - 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)

## 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

## 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
* Interpreted 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)

#### Programming workflow

“Flowchart demonstrating roles and tasks of a programmer, beta tester and user in the creation of programs.”

“Flowchart demonstrating roles and tasks of a programmer, beta tester and user in the creation of programs.”

* 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

# C# Fundamentals

## 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
  + [8th most “loved” language on StackOverflow](https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved)
  + .NET is the [2nd most used “other” library/framework](https://insights.stackoverflow.com/survey/2020#technology-other-frameworks-libraries-and-tools)
  + More insights on its evolution can be found [in this blog post](https://dottutorials.net/stats-surveys-about-net-core-future-2020/#stackoverflow-surveys).

## 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:
  + “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

# First Program

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

#### 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.

## 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

## 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)

## 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
    - Must be unique (you cannot re-use the same identifier twice in the same scope – a concept we will discuss later)
  + 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](https://en.wikipedia.org/wiki/Camel_case) (or its variation [Pascal case](https://www.c-sharpcorner.com/UploadFile/8a67c0/C-Sharp-coding-standards-and-naming-conventions/)) 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”

## Write and WriteLine

* The WriteLine method
  + We saw this in the “Hello World” program: Console.WriteLine("Hello World!"); results in “Hello World!” being displayed in the terminal
  + In general, Console.WriteLine("text"); will display the text in the terminal, then *start a new line*
  + This means a second Console.WriteLine will display its text on the next line of the terminal. For example, this program:
  + using System;  
      
    class Welcome  
    {  
     static void Main()  
     {  
     Console.WriteLine("Hello");  
     Console.WriteLine("World!");  
     }  
    }
  + will display the following output in the terminal:
  + Hello  
    World!
* Methods with multiple statements
  + Note that our two-line example has a Main method with multiple statements
  + In C#, each statement must end in a semicolon
  + Class and method declarations are not statements
  + Each line of code in your .cs file is not necessarily a statement
  + A single invocation/call of the WriteLine method is a statement
* The Write method
  + Console.WriteLine("text") prints the text, then starts a new line in the terminal – it effectively “hits enter” after printing the text
  + Console.Write("text") just prints the text, without starting a new line. It’s like typing the text without hitting “enter” afterwards.
  + Even though two Console.Write calls are two statements, and appear on two lines, they will result in the text being printed on just one line. For example, this program:
  + using System;  
      
    class Welcome  
    {  
     static void Main()  
     {  
     Console.Write("Hello");  
     Console.Write("World!");  
     }  
    }
  + will display the following output in the terminal:
  + HelloWorld!
  + Note that there is no space between “Hello” and “World!” because we didn’t type one in the argument to Console.Write
* Combining Write and WriteLine
  + We can use both WriteLine and Write in the same program
  + After a call to Write, the “cursor” is on the same line after the printed text; after a call to WriteLine the “cursor” is at the beginning of the next line
  + This program:
  + using System;  
      
    class Welcome  
    {  
     static void Main()  
     {  
     Console.Write("Hello ");  
     Console.WriteLine("World!");  
     Console.Write("Welcome to ");  
     Console.WriteLine("CSCI 1301!");  
     }  
    }
  + will display the following output in the terminal:
  + Hello world!  
    Welcome to CSCI 1301!

## Escape Sequences

* Explicitly writing a new line
  + So far we’ve used WriteLine when we want to create a new line in the output
  + The **escape sequence** \n can also be used to create a new line – it represents the “newline character,” which is what gets printed when you type “enter”
  + This program will produce the same output as our two-line “Hello World” example, with each word on its own line:
  + using System;  
      
    class Welcome  
    {  
     static void Main()  
     {  
     Console.Write("Hello\nWorld!\n");  
     }  
    }
* Escape sequences in detail
  + An **escape sequence** uses “normal” letters to represent “special”, hard-to-type characters
  + \n represents the newline character, i.e. the result of pressing “enter”
  + \t represents the tab character, which is a single extra-wide space (you usually get it by pressing the “tab” key)
  + \" represents a double-quote character that will get printed on the screen, rather than ending the text string in the C# code.
    - Without this, you couldn’t write a sentence with quotation marks in a Console.WriteLine, because the C# compiler would assume the quotation marks meant the string was ending
    - This program won’t compile because in quotes is not valid C# code, and the compiler thinks it is not part of the string:
    - class Welcome  
      {  
       static void Main()  
       {  
       Console.WriteLine("This is "in quotes"");  
       }  
      }
    - This program will display the sentence including the quotation marks:
    - using System;  
        
      class Welcome  
      {  
       static void Main()  
       {  
       Console.WriteLine("This is \"in quotes\"");  
       }  
      }
  + Note that all escape sequences begin with a backslash character (\)
  + General format is \[key letter] – the letter after the backslash is like a “keyword” indicating which special character to display
  + If you want to put an actual backslash in your string, you need the escape sequence \\, which prints a single backslash
    - This will result in a compile error because \U is not a valid escape sequence:
    - Console.WriteLine("Go to C:\Users\Edward");
    - This will display the path correctly:
    - Console.WriteLine("Go to C:\\Users\\Edward");

# Datatypes and Variables

## Datatype Basics

* Recall the basic structure of a program
  + Program receives input from some source, uses input to make decisions, produces output for the outside world to see
  + In other words, the program reads some data, manipulates data, and writes out new data
  + In C#, data is stored in objects during the program’s execution, and manipulated using the methods of those objects
* This data has **types**
  + Numbers (the number 2) are different from text (the word “two”)
  + Text data is called “strings” because each letter is a **character** and a word is a *string of characters*
  + Within “numeric data,” there are different types of numbers
    - Natural numbers (ℕ): 0, 1, 2, …
    - Integers (ℤ): … -2, -1, 0, 1, 2, …
    - Real numbers (ℝ): 0.5, 1.333333…, -1.4, etc.
* Basic Datatypes in C#
  + C# uses keywords to name the types of data
  + Text data:
    - string: a string of characters, like "Hello world!"
    - char: a single character, like 'e' or 't'
  + Numeric data:
    - int: An integer, as defined previously
    - uint: An *unsigned* integer, in other words, a natural number (positive integers only)
    - float: A “floating-point” number, which is a real number with a fractional part, such as 3.85
    - double: A floating-point number with “double precision” – also a real number, but capable of storing more significant figures
    - decimal: An “exact decimal” number – also a real number, but has fewer rounding errors than float and double (we’ll explore the difference later)

## Literals and Variables

* Literals and their types
  + A **literal** is a data value written in the code
  + A form of “input” provided by the programmer rather than the user; its value is fixed throughout the program’s execution
  + Literal data must have a type, indicated by syntax:
    - string literal: text in double quotes, like "hello"
    - char literal: a character in single quotes, like 'a'
    - int literal: a number without a decimal point, with or without a minus sign (e.g. 52)
    - long literal: just like an int literal but with the suffix l or L, e.g. 4L
    - double literal: a number with a decimal point, with or without a minus sign (e.g. -4.5)
    - float literal: just like a double literal but with the suffix f or F (for “float”), e.g. 4.5f
    - decimal literal: just like a double literal but with the suffix m or M(for “deciMal”), e.g. 6.01m
* Variables overview
  + Variables store data that can *vary* (change) during the program’s execution
  + They have a type, just like literals, and also a name
  + You can use literals to write data that gets stored in variables
  + Sample program with variables:
  + using System;  
      
    class MyFirstVariables  
    {  
     static void Main()  
     {  
     // Declaration  
     int myAge;  
     string myName;  
     // Assignment  
     myAge = 29;  
     myName = "Edward";  
     // Displaying  
     Console.WriteLine($"My name is {myName} and I am {myAge} years old.");  
     }  
    }
  + This program shows three major operations you can do with variables.
    - First it **declares** two variables, an int-type variable named “myAge” and a string-type variable named “myName”
    - Then, it **assigns** values to each of those variables, using literals of the same type. myAge is assigned the value 29, using the int literal 29, and myName is assigned the value “Edward”, using the string literal "Edward"
    - Finally, it **displays** the current value of each variable by using the Console.WriteLine method and **string interpolation**, in which the values of variables are inserted into a string by writing their names with some special syntax (a $ character at the beginning of the string, and braces around the variable names)

## Variable Operations

* Declaration
  + This is when you specify the *name* of a variable and its *type*
  + Syntax: type\_keyword variable\_name;
  + Examples: int myAge;, string myName;, double winChance;
  + A variable name is an identifier, so it should follow the rules and conventions
    - Can only contain letters and numbers
    - Must be unique among all variable, method, and class names
    - Should use CamelCase if it contains multiple words
  + Note that the variable’s type is not part of its name: two variables cannot have the same name *even if* they are different types
  + Multiple variables can be declared in the same statement: string myFirstName, myLastName; would declare *two* strings called respectively myFirstName and myLastName.
* Assignment
  + The act of changing the value of a variable
  + Uses the symbol =, which is the *assignment operator*, not a statement of equality – it does not mean “equals”
  + Direction of assignment is **right to left**: the variable goes on the left side of the = symbol, and its new value goes on the right
  + Syntax: variable\_name = value;
  + Example: myAge = 29;
  + Value *must* match the type of the variable. If myAge was declared as an int-type variable, you cannot write myAge = "29"; because "29" is a string
* Initialization (Declaration + Assignment)
  + Initialization statement combines declaration and assignment in one line (it’s just a shortcut, not a new operation)
  + Creates a new variable and also gives it an initial value
  + Syntax: type variable\_name = value;
  + Example: string myName = "Edward";
  + Can only be used once per variable, since you can only declare a variable once
* Assignment Details
  + Assignment replaces the “old” value of the variable with a “new” one; it’s how variables *vary*
    - If you initialize a variable with int myAge = 29; and then write myAge = 30;, the variable myAge now store the value 30
  + You can assign a variable to another variable: just write a variable name on both sides of the = operator
    - This will take a “snapshot” of the current value of the variable on the right side, and store it into the variable on the left side
    - For example, in this code:
    - int a = 12;  
      int b = a;  
      a = -5;
    - the variable b gets the value 12, because that’s the value that a had when the statement int b = a was executed. Even though a was then changed to -5 afterward, b is still 12.
* Displaying
  + When you want to print a mixture of values and variables with Console.WriteLine, we should convert all of them to a string
  + **String interpolation**: a mechanism for converting a variable’s value to a string and inserting it into the main string
    - Syntax: $"text {variable} text" – begin with a $ symbol, then put variable’s name inside brackets within the string
    - Example: $"I am {myAge} years old"
    - When this line of code is executed, it reads the variable’s current value, converts it to a string (29 becomes "29"), and inserts it into the surrounding string
    - Displayed: I am 29 years old
  + When string interpolation converts a variable to a string, it must call a “string conversion” method supplied with the data type (int, double, etc.). All built-in C# datatypes come with string conversion methods, but when you write your own data types (classes), you’ll need to write your own string conversions – string interpolation won’t magically “know” how to convert MyClass variables to strings

On a final note, observe that you can write statements mixing multiple declarations and assignments, as in int myAge = 10, yourAge, ageDifference; that declares three variables of type int and set the value of the first one. It is generally recommended to separate those instructions in different statements as you begin, to ease debugging and have a better understanding of the “atomic steps” your program should perform.

## Variables in Memory

* A variable names a memory location
  + Data is stored in memory (RAM), so a variable “stores data” by storing it in memory
  + Declaring a variable reserves a memory location (address) and gives it a name
  + Assigning to a variable stores data to the memory location (address) named by that variable
* Numeric datatypes have different sizes
  + Amount of memory used/reserved by each variable depends on the variable’s type
  + Amount of memory needed for an integer data type depends on the size of the number
    - int uses 4 bytes of memory, can store numbers in the range
    - long uses 8 bytes of memory can store numbers in the range
    - short uses 2 bytes of memory, can store numbers in the range
    - sbyte uses only 1 bytes of memory, can store numbers in the range
  + Unsigned versions of the integer types use the same amount of memory, but can store larger positive numbers
    - byte uses 1 byte of memory, can store numbers in the range
    - ushort uses 2 bytes of memory, can store numbers in the range
    - uint uses 4 bytes of memory, can store numbers in the range
    - ulong uses 8 bytes of memory, can store numbers in the range
    - This is because in a signed integer, one bit (digit) of the binary number is needed to represent the sign (+ or -). This means the actual number stored must be 1 bit smaller than the size of the memory (e.g. 31 bits out of the 32 bits in 4 bytes). In an unsigned integer, there is no “sign bit”, so all the bits can be used for the number.
  + Amount of memory needed for a floating-point data type depends on the precision (significant figures) of the number
    - float uses 4 bytes of memory, can store positive or negative numbers in a range of approximately , with 7 significant figures of precision
    - double uses 8 bytes of memory, and has both a wider range ( to ) and more significant figures (15 or 16)
    - decimal uses 16 bytes of memory, and has 28 or 29 significant figures of precision, but it actually has the smallest range ( to ) because it stores decimal fractions exactly
  + Difference between binary fractions and decimal fractions
    - float and double store their data as binary (base 2) fractions, where each digit represents a power of 2
      * The binary number 101.01 represents , or 5.25 in base 10
    - More specifically, they use binary scientific notation: A mantissa (a binary integer), followed by an exponent assumed to be a power of 2, which is applied to the mantissa
      * 10101e-10 means a mantissa of 10101 (i.e. 21 in base 10) with an exponent of -10 (i.e.  in base 10), which also produces the value 101.01 or 5.25 in base 10
    - Binary fractions can’t represent all base-10 fractions, because they can only represent fractions that are negative powers of 2. is not a negative power of 2 and can’t be represented as a sum of , , , etc.
    - This means some base-10 fractions will get “rounded” to the nearest finite binary fraction, and this will cause errors when they are used in arithmetic
    - On the other hand, decimal stores data as a base-10 fraction, using base-10 scientific notation
    - This is slower for the computer to calculate with (since computers work only in binary) but has no “rounding errors” with fractions that include 0.1
    - Use decimal when working with money (since money uses a lot of 0.1 and 0.01 fractions), double when working with non-money fractions

**Summary of numeric data types and sizes:**

| Type | Size | Range of Values | Precision |
| --- | --- | --- | --- |
| sbyte | 1 bytes |  | N/A |
| byte | 1 bytes |  | N/A |
| short | 2 bytes |  | N/A |
| ushort | 2 bytes |  | N/A |
| int | 4 bytes |  | N/A |
| uint | 4 bytes |  | N/A |
| long | 8 bytes |  | N/A |
| ulong | 8 bytes |  | N/A |
| float | 4 bytes |  | 7 digits |
| double | 8 bytes |  | 15-16 digits |
| decimal | 16 bytes |  | 28-29 digits |

* Value and reference types: different ways of storing data in memory
  + Variables name memory locations, but the data that gets stored at the named location is different for each type
  + For a **value type** variable, the named memory location stores the exact data value held by the variable (just what you’d expect)
  + Value types: all the numeric types (int, float, double, decimal, etc.), char, and bool
  + For a **reference type** variable, the named memory location stores a *reference* to the data, not the data itself
    - The contents of the memory location named by the variable are the address of another memory location
    - The *other* memory location is where the variable’s data is stored
    - To get to the data, the computer first reads the location named by the variable, then uses that information (the memory address) to find and read the other memory location where the data is stored
  + Reference types: string, object, and all objects you create from your own classes
  + Assignment works differently for reference types
    - Assignment always copies the value in the variable’s named memory location - but in the case of a reference type that’s just a memory address, not the data
    - Assigning one reference-type variable to another copies the memory address, so now both variables “refer to” the same data
    - Example:
    - string word = "Hello";  
      string word2 = word;
    - Both word and word2 contain the same memory address, pointing to the same memory location, which contains the string “Hello”. There is only one copy of the string “Hello”; word2 doesn’t get its own copy.

## Overflow 🛡️

* Assume a car has a 4-digit odometer, and currently, it shows 9999. What does the odometer show if you drive the car another mile? As you guess, it shows 0000 while it should show 10000. The reason is the odometer does not have a counter for the fifth digit. Similarly, in C#, when you do arithmetic operations on integral data, the result may not fit in the corresponding data type. This situation is called overflow error.
* In an unsigned data type variable with bits, we can store the numbers ranged from 0 to . In signed data type variables, the high order bit represents the sign of the number as follows:
  + 0 means zero or a positive value
  + 1 means a negative value
* With the remaining bits, we can represent states. Hence, considering the sign bit, we can store a number from to in the variable.
* In many programming languages like C, overflow error raise an exceptional situation that crashes the program if it is not handled. But, in C#, the extra bits are just ignored, and if the programmer does not care about such a possibility, it can lead to a severe security problem.
* For example, assume a company gives loans to its employee. Couples working for the company can get loans separately, but the total amount can not exceed $10,000. The underneath program looks like it does this job, but there is a risk of attacks. (This program uses notions you have not studied yet, but that should not prevent you from reading the source code and executing it.)
* using System;  
    
  class Program  
  {  
   static void Main()  
   {  
   uint n1, n2;  
    
   Console.WriteLine("Enter the requested loan amount for the first person:");  
   n1 = uint.Parse(Console.ReadLine());  
    
   Console.WriteLine("Enter the requested loan amount for the second person:");  
   n2 = uint.Parse(Console.ReadLine());  
    
   if(n1 + n2 < 10000)  
   {  
   Console.WriteLine($"Pay ${n1} for the first person");  
   Console.WriteLine($"Pay ${n2} for the second person");  
   }  
   else  
   {  
   Console.WriteLine("Error: the sum of loans exceeds the maximum allowance.");  
   }  
   }  
  }
* If the user enters 2 and 4,294,967,295, we expect to see the error message (“Error: the sum of loans exceeds the maximum allowance.”). However, this is not what will happened, and the request will be accepted even if it should not have. The reason can be explained as follows:
  + uint is a 32-bit data type.
  + The binary representation of 2 and 4,294,967,295 are 00000000000000000000000000000010 and 11111111111111111111111111111111.
  + Therefore, the sum of these numbers should be 100000000000000000000000000000001, which needs 33 bits.
  + Nevertheless, there is only 32 bits available for the result, and the extra bits will be dropped, and the result looks like 00000000000000000000000000000001, which is less than 10,000.

## Underflow 🛡️

* Sometimes, the result of arithmetic operations over floating-point numbers is smaller than what can be stored in the corresponding data type. This problem is known as the underflow problem.
* In C#, in case of an underflow problem, the result will be zero.
* using System;  
    
  class Program  
  {  
   static void Main()  
   {  
   float myNumber;  
   myNumber = 1E-45f;  
   Console.WriteLine(myNumber); //outputs 1.401298E-45  
   myNumber = myNumber / 10;  
   Console.WriteLine(myNumber); //outputs 0  
   myNumber = myNumber \* 10;  
   Console.WriteLine(myNumber); //outputs 0  
   }  
  }

# Operators

## Arithmetic Operators

Variables or literals of numeric data types (int, double, etc.) can be used to do math. All the usual arithmetic operations are available in C#:

| Operation | C# Operator | Algebraic Expression | C# Expression |
| --- | --- | --- | --- |
| Addition | + |  | myVar + 7 |
| Subtraction | - |  | myVar - 7 |
| Multiplication | \* |  | myVar \* 7 |
| Division | / | , | myVar / 7 |
| Remainder (a.k.a. modulo) | % |  | myVar % 7 |

Note: the “remainder” or “modulo” operator represents the remainder after doing integer division between its two operands. For example, because *with remainder* .

* Arithmetic and variables
  + The result of an arithmetic expression (like those shown in the table) is a numeric value
    - For example, the C# expression 3 \* 4 has the value 12, which is int data
  + A numeric value can be assigned to a variable of the same type, just like a literal: int myVar = 3 \* 4; initializes the variable myVar to contain the value 12
  + A numeric-type variable can be used in an arithmetic expression
  + When a variable is used in an arithmetic expression, its current value is read, and the math is done on that value
  + Example:
  + int a = 4;  
    int b = a + 5;  
    a = b \* 2;
    - To execute the second line of the code, the computer will first evaluate the expression on the right side of the = sign. It reads the value of the variable a, which is 4, and then computes the result of 4 + 5, which is 9. Then, it assigns this value to the variable b (remember assignment goes right to left).
    - To execute the third line of code, the computer first evaluates the expression on the right side of the = sign, which means reading the value of b to use in the arithmetic operation. b contains 9, so the expression is 9 \* 2, which evaluates to 18. Then it assigns the value 18 to the variable a, which now contains 18 instead of 4.
  + A variable can appear on both sides of the = sign, like this:
  + int myVar = 4;  
    myVar = myVar \* 2;
  + This looks like a paradox because myVar is assigned to itself, but it has a clear meaning because assignment is evaluated right to left. When executing the second line of code, the computer evaluates the right side of the = before doing the assignment. So it first reads the current (“old”) value of myVar, which is 4, and computes 4 \* 2 to get the value 8. Then, it assigns the new value to myVar, overwriting its old value.
* Compound assignment operators
  + The pattern of “compute an expression with a variable, then assign the result to that variable” is common, so there are shortcuts for doing it
  + The **compound assignment operators** change the value of a variable by adding, subtracting, etc. from its current value, equivalent to an assignment statement that has the value on both sides:

| * Statement | * Equivalent |
| --- | --- |
| * x += 2; | * x = x + 2; |
| * x -= 2; | * x = x - 2; |
| * x \*= 2; | * x = x \* 2; |
| * x /= 2; | * x = x / 2; |
| * x %= 2; | * x = x % 2; |

## Implicit and Explicit Conversions Between Datatypes

* Assignments from different types
  + The “proper” way to initialize a variable is to assign it a literal of the same type:
  + int myAge = 29;  
    double myHeight = 1.77;  
    float radius = 2.3f;
  + Note that 1.77 is a double literal, while 2.3f is a float literal
  + If the literal is not the same type as the variable, you will sometimes get an error – for example, float radius = 2.3 will result in a compile error – but sometimes, it appears to work fine: for example float radius = 2; compiles and runs without error even though 2 is an int value.
  + In fact, the value being assigned to the variable **must** be the same type as the variable, but some types can be **implicitly converted** to others
* Implicit conversions
  + Implicit conversion allows variables to be assigned from literals of the “wrong” type: the literal value is first implicitly converted to the right type
    - In the statement float radius = 2;, the int value 2 is implicitly converted to an equivalent float value, 2.0f. Then the computer assigns 2.0f to the radius variable.
  + Implicit conversion also allows variables to be assigned from other variables that have a different type:
  + int length = 2;  
    float radius = length;
* When the computer executes the second line of this code, it reads the variable length to get an int value 2. It then implicitly converts that value to 2.0f, and then assigns 2.0f to the float-type variable radius.
  + Implicit conversion only works between *some* data types: a value will only be implicitly converted if it is “safe” to do so without losing data
  + Summary of possible implicit conversions:

| * + Type | * + Possible Implicit Conversions |
| --- | --- |
| * + short | * + int, long, float, double, decimal |
| * + int | * + long, float, double, decimal |
| * + long | * + float, double, decimal |
| * + ushort | * + uint, int, ulong, long, decimal, float, double |
| * + uint | * + ulong, long, decimal, float, double |
| * + ulong | * + decimal, float, double |
| * + float | * + double |

* + In general, a data type can only be implicitly converted to one with a *larger range* of possible values
  + Since an int can store any integer between and , but a float can store any integer between and (as well as fractional values), it’s always safe to store an int value in a float
  + You *cannot* implicitly convert a float to an int because an int stores fewer values than a float – it can’t store fractions – so converting a float to an int will **lose data**
  + Note that all integer data types can be implicitly converted to float or double
  + Each integer data type can be implicitly converted to a larger integer type: short int long
  + Unsigned integer data types can be implicitly converted to a *larger* signed integer type, but not the *same* signed integer type: uint long, but **not** uint int
    - This is because of the “sign bit”: a uint can store larger values than an int because it doesn’t use a sign bit, so converting a large uint to an int might lose data
* Explicit conversions
  + Any conversion that is “unsafe” because it might lose data will not happen automatically: you get a compile error if you assign a double variable to a float variable
  + If you want to do an unsafe conversion anyway, you must perform an **explicit conversion** with the **cast operator**
  + Cast operator syntax: ([type name]) [variable or value] – the cast is “right-associative”, so it applies to the variable to the right of the type name
  + Example: (float) 2.8 or (int) radius
  + Explicit conversions are often used when you (the programmer) know the conversion is actually “safe” – data won’t actually be lost
    - For example, in this code, we know that 2.886 is within the range of a float, so it’s safe to convert it to a float:
    - float radius = (float) 2.886;
    - The variable radius will be assigned the value 2.886f.
    - For example, in this code, we know that 2.0 is safe to convert to an int because it has no fractional part:
    - double length = 2.0;  
      int height = (int) length;
    - The variable height will be assigned the value 2.
  + Explicit conversions only work if there exists code to perform the conversion, usually in the standard library. The cast operator isn’t “magic” – it just calls a method that is defined to convert one type of data (e.g. double) to another (e.g. int).
    - All the C# numeric types have explicit conversions to each other defined
    - string does not have explicit conversions defined, so you cannot write int myAge = (int) "29";
  + If the explicit conversion is truly unsafe (will lose data), data is lost in a specific way
    - Casting from floating-point (e.g. double) types to integer types: fractional part of number is *truncated* (ignored/dropped)
    - In int length = (int) 2.886;, the value 2.886 is truncated to 2 by the cast to int, so the variable length gets the value 2.
    - Casting from more-precise to less-precise floating point type: number is *rounded* to nearest value that fits in less-precise type:
    - decimal myDecimal = 123456789.999999918m;  
      double myDouble = (double) myDecimal;  
      float myFloat = (float) myDouble;
    - In this code, myDouble gets the value 123456789.99999993, while myFloat gets the value 123456790.0f, as the original decimal value is rounded to fit types with fewer significant figures of precision.
    - Casting from a larger integer to a smaller integer: the most significant *bits* are truncated – remember that numbers are stored in binary format
    - This can cause weird results, since the least-significant *bits* of a number in binary don’t correspond to the least significant *digits* of the equivalent base-10 number
    - Casting from another floating point type to decimal: Either value is stored precisely (no rounding), or *program crashes* with System.OverflowException if value is larger than decimal’s maximum value:
    - decimal fromSmall = (decimal) 42.76875;  
      double bigDouble = 2.65e35;  
      decimal fromBig = (decimal) bigDouble;
    - In this code, fromSmall will get the value 42.76875m, but the program will crash when attempting to cast bigDouble to a decimal because is larger than decimal’s maximum value of
    - decimal is more precise than the other two floating-point types (thus doesn’t need to round), but has a smaller range (only , vs. )

Summary of implicit and explicit conversions for the numeric datatypes:

“Implicit and Explicit Conversion Between Datatypes”

“Implicit and Explicit Conversion Between Datatypes”

Refer to [the “Result Type of Operations” chart from the cheatsheet](../datatypes_in_csharp.html#result-type-of-operations) for more detail.

## Arithmetic on Mixed Data Types

* Math operators for each data type
  + The math operators (+, -, \*, /) are defined separately for each data type: There is an int version of + that adds ints, a float version of + that adds floats, etc.
  + Each operator expects to get two values of the same type on each side, and produces a result of that same type. For example, 2.25 + 3.25 uses the double version of +, which adds the two double values to produce a double-type result, 5.5.
  + Most operators have the same effect regardless of their type, except for /
  + The int/short/long version of / does **integer division**, which returns only the quotient and drops the remainder: In the statement int result = 21 / 5;, the variable result gets the value 4, because is 4 with a remainder of 1. If you want the fractional part, you need to use the floating-point version (for float, double, and decimal): double fracDiv = 21.0 / 5.0; will initialize fracDiv to 4.2.
* Implicit conversions in math
  + If the two operands/arguments to a math operator are not the same type, they must become the same type – one must be converted
  + C# will first try implicit conversion to “promote” a less-precise or smaller value to a more precise, larger type
  + Example: with the expression double fracDiv = 21 / 2.4;
    - Operand types are int and double
    - int is smaller/less-precise than double
    - 21 gets implicitly converted to 21.0, a double value
    - Now the operands are both double type, so the double version of the / operator gets executed
    - The result is 8.75, a double value, which gets assigned to the variable fracDiv
  + Implicit conversion also happens in assignment statements, which happen *after* the math expression is computed
  + Example: with the expression double fraction = 21 / 5;
    - Operand types are int and int
    - Since they match, the int version of / gets executed
    - The result is 4, an int value
    - Now this value is assigned to the variable fraction, which is double type
    - The int value is implicitly converted to the double value 4.0, and fraction is assigned the value 4.0
* Explicit conversions in math
  + If the operands are int type, the int version of / will get called, even if you assign the result to a double
  + You can “force” floating-point division by explicitly converting one operand to double or float
  + Example:
  + int numCookies = 21;  
    int numPeople = 6;  
    double share = (double) numCookies / numPeople;
  + Without the cast, share would get the value 3.0 because numCookies and numPeople are both int type (just like the fraction example above). With the cast, numCookies is converted to the value 21.0 (a double), which means the operands are no longer the same type. This will cause numPeople to be implicitly converted to double in order to make them match, and the double version of / will get called to evaluate 21.0 / 6.0. The result is 3.5, so share gets assigned 3.5.
  + You might also *need* a cast to ensure the operands are the same type, if implicit conversion doesn’t work
  + Example:
  + decimal price = 3.89;  
    double shares = 47.75;  
    decimal total = price \* (decimal) shares;
  + In this code, double can’t be implicitly converted to decimal, and decimal can’t be explicitly converted to double, so the multiplication price \* shares would produce a compile error. We need an explicit cast to decimal to make both operands the same type (decimal).

## Order of Operations

* Math operations in C# follow PEMDAS from math class: Parentheses, Exponents, Multiplication, Division, Addition, Subtraction
  + Multiplication/division are evaluated together, as are addition/subtraction
  + Expressions are evaluated left-to-right
  + Example: int x = 4 = 10 \* 3 - 21 / 2 - (3 + 3);
    - Parentheses: (3 + 3) is evaluated, returns 6
    - Multiplication/Division: 10 \* 3 is evaluated to produce 30, then 21 / 2 is evaluated to produce 10 (left-to-right)
    - Addition/Subtraction: 4 + 30 - 10 - 6 is evaluated, result is 18
* Cast operator is higher priority than all binary operators
  + Example: double share = (double) numCookies / numPeople;
    - Cast operator is evaluated first, converts numCookies to a double
    - Division is evaluated next, but operand types don’t match
    - numPeople is implicitly converted to double to make operand types match
    - Then division is evaluated, result is 21.0 / 6.0 = 3.5
* Parentheses always increase priority, even with casts
  + An expression in parentheses gets evaluated before the cast “next to” it
  + Example:
  + int a = 5, b = 4;  
    double result = (double) (a / b);
  + The expression in parentheses gets evaluated first, then the result has the (double) cast applied to it. That means a / b is evaluated to produce 1, since a and b are both int type, and then that result is cast to a double, producing 1.0.

# Reading Input, Displaying Output, and Concatenation

## Output with Variables

* Converting from numbers to strings
  + As we saw in a previous lecture (Datatypes and Variables), the Console.WriteLine method needs a string as its argument
  + If the variable you want to display is not a string, you might think you could cast it to a string, but that won’t work – there is no explicit conversion from string to numeric types
    - This code:
    - double fraction = (double) 47 / 6;  
      string text = (string) fraction;
    - will produce a compile error
  + You *can* convert numeric data to a string using string interpolation, which we’ve used before in Console.WriteLine statements:
  + int x = 47, y = 6;  
    double fraction = (double) x / y;  
    string text = $"{x} divided by {y} is {fraction}";
  + After executing this code, text will contain “47 divided by 6 is 7.8333333”
  + String interpolation can convert any expression to a string, not just a single variable. For example, you can write:
  + Console.WriteLine($"{x} divided by {y} is {(double) x / y}");  
    Console.WriteLine($"{x} plus 7 is {x + 7}");
  + This will display the following output:
  + 47 divided by 6 is 7.8333333  
    47 plus 7 is 54
  + Note that writing a math expression inside a string interpolation statement does not change the values of any variables. After executing this code, x is still 47, and y is still 6.
* The ToString() method
  + String interpolation doesn’t “magically know” how to convert numbers to strings – it delegates the task to the numbers themselves
  + This works because all data types in C# are objects, even the built-in ones like int and double
    - Since they are objects, they can have methods
  + **All** objects in C# are guaranteed to have a method named ToString(), whose return value (result) is a string
  + Meaning of ToString() method: “Convert this object to a string, and return that string”
  + This means you can call the ToString() method on any variable to convert it to a string, like this:
  + int num = 42;  
    double fraction = 33.5;  
    string intText = num.ToString();  
    string fracText = fraction.ToString();
  + After executing this code, intText will contain the string “42”, and fracText will contain the string “33.5”
  + String interpolation calls ToString() on each variable or expression within braces, asking it to convert itself to a string
    - In other words, these three statements are all the same:
    - Console.WriteLine($"num is {num}");  
      Console.WriteLine($"num is {intText}");  
      Console.WriteLine($"num is {num.ToString()}");
    - Putting num within the braces is the same as calling ToString() on it.

## String Concatenation

* Now that we’ve seen ToString(), we can introduce another operator: the concatenation operator
* Concatenation basics
  + Remember, the + operator is defined separately for each data type. The “double + double” operator is different from the “int + int” operator.
  + If the operand types are string (i.e. string + string), the + operator performs concatenation, not addition
  + You can concatenate string literals or string variables:
  + string greeting = "Hi there, " + "John";  
    string name = "Paul";  
    string greeting2 = "Hi there, " + name;
  + After executing this code, greeting will contain “Hi there, John” and greeting2 will contain “Hi there, Paul”
* Concatenation with mixed types
  + Just like with the other operators, both operands (both sides of the +) must be the same type
  + If one operand is a string and the other is not a string, the ToString() method will automatically be called to convert it to a string
  + Example: In this code:
  + int bananas = 42;  
    string text = "Bananas: " + bananas;
  + The + operator has a string operand and an int operand, so the int will be converted to a string. This means the computer will call bananas.ToString(), which returns the string “42”. Then the string + string operator is called with the operands “Bananas:” and “42”, which concatenates them into “Bananas: 42”.
* Output with concatenation
  + We now have two different ways to construct a string for Console.WriteLine: Interpolation and concatenation
  + Concatenating a string with a variable will automatically call its ToString() method, just like interpolation will. These two WriteLine calls are equivalent:
  + int num = 42;  
    Console.WriteLine($"num is {num}");  
    Console.WriteLine("num is " + num);
  + It’s usually easier to use interpolation, since when you have many variables the + signs start to add up. Compare the length of these two equivalent lines of code:
  + Console.WriteLine($"The variables are {a}, {b}, {c}, {d}, and {e}");  
    Console.WriteLine("The variables are " + a + ", " + b + ", " + c + ", " + d + ", and " + e);
  + Be careful when using concatenation with numeric variables: the meaning of + depends on the types of its two operands
    - If both operands are numbers, the + operator does addition
    - If both operands are strings, the + operator does concatenation
    - If *one* argument is a string, the other argument will be converted to a string using ToString()
    - Expressions in C# are always evaluated **left-to-right**, just like arithmetic
    - Therefore, in this code:
    - int var1 = 6, var2 = 7;  
      Console.WriteLine(var1 + var2 + " is the result");  
      Console.WriteLine("The result is " + var1 + var2);
    - The first WriteLine will display “13 is the result”, because var1 and var2 are both ints, so the first + operator performs addition on two ints (the resulting number,13, is then converted to a string for the second + operator). However, the second WriteLine will display “The result is 67”, because both + operators perform concatenation: The first one concatenates a string with var1 to produce a string, and then the second one concatenates this string with var2
    - If you want to combine addition and concatenation in the same line of code, use parentheses to make the order and grouping of operations explicit. For example:
    - int var1 = 6, var2 = 7;  
      Console.WriteLine((var1 + var2) + " is the result");  
      Console.WriteLine("The result is " + (var1 + var2));
    - In this code, the parentheses ensure that var1 + var2 is always interpreted as addition.

## Reading Input from the User

* Input and output in CLI
  + Our programs use a command-line interface, where input and output come from text printed in a “terminal” or “console”
  + We’ve already seen that Console.WriteLine prints text on the screen to provide output to the user
  + The equivalent method for reading input is Console.ReadLine(), which waits for the user to type some text in the console and then returns it
  + In general, the Console class represents the command-line interface
* Using Console.ReadLine()
  + This method is the “inverse” of Console.WriteLine, and the way you use it is also the “inverse”
  + While Console.WriteLine takes an argument, which is the text you want to display on the screen, Console.ReadLine() takes no arguments because it doesn’t need any input from your program – it will always do the same thing
  + Console.WriteLine has no “return value” - it doesn’t give any output back to your program, and the only effect of calling it is that text is displayed on the screen
  + Console.ReadLine() does have a return value, specifically a string, just like ToString(). This means you can use the result of this method to assign a string variable.
  + The string that Console.ReadLine() returns is **one line of text** typed in the console. When you call it, the computer will wait for the user to type some text and then press “Enter”, and everything the user typed before pressing “Enter” gets returned from Console.ReadLine()
  + Example usage:
  + using System;  
      
    class PersonalizedWelcomeMessage  
    {  
     static void Main()  
     {  
     string firstName;  
     Console.WriteLine("Enter your first name:");  
     firstName = Console.ReadLine();  
     Console.WriteLine($"Welcome, {firstName}!");  
     }  
    }
  + This program first declares a string variable named firstName. On the second line, it uses Console.WriteLine to display a message (instructions for the user). On the third line, it calls the Console.ReadLine() method, and assigns its return value (result) to the firstName variable. This means the program waits for the user to type some text and press “Enter”, and then stores that text in firstName. Finally, the program uses string interpolation in Console.WriteLine to display a message including the contents of the firstName variable.
* Parsing user input
  + Casting cannot be used to convert numeric data *to or from* string data
  + When converting numeric data to string data, we use the ToString() method:
  + int myAge = 29;  
    //This does not work:  
    //string strAge = (string) myAge;  
    string strAge = myAge.ToString();
  + Similarly, we use a method to convert strings to numbers:
  + string strAge = "29";  
    //This does not work:  
    //int myAge = (int) strAge;  
    int myAge = int.Parse(strAge);
  + The int.Parse method takes a string as an argument, and returns an int containing the numeric value written in that string
  + Each built-in numeric type has its own Parse method
    - int.Parse("42") returns the value 42
    - long.Parse("42") returns the value 42L
    - double.Parse("3.65") returns the value 3.65
    - float.Parse("3.65") returns the value 3.65f
  + The Parse methods are useful for converting user input to useable data. Remember, Console.ReadLine() will always return a string, even if you asked the user to enter a number.
  + Example of parsing user input:
  + Console.WriteLine("Please enter the year.");  
    string userInput = Console.ReadLine();  
    int curYear = int.Parse(userInput);  
    Console.WriteLine($"Next year it will be {curYear + 1}");
  + In order to do arithmetic with the user’s input (i.e. add 1), it must be a numeric type (i.e. int), not a string.
  + The Parse methods *assume* that the string they are given as an argument (i.e. the user input) actually contains a valid number. If not, they will make the program crash
    - If the string does not contain a number at all – e.g. int badIdea = int.Parse("Hello"); – the program will fail with the error System.FormatException
    - If the string contains a number that cannot fit in the desired datatype, the program will fail with the error System.OverflowException. For example, int.Parse("52.5") will cause this error because an int cannot contain a fraction, and int.Parse("3000000000") will fail because 3000000000 is larger than (the maximum value an int can store)

# Classes, Objects, and UML

## Class and Object Basics

* Classes vs. Objects
  + A **class** is a specification, blueprint, or template for an object; it is the code that describes what data the object stores and what it can do
  + An **object** is a single instance of a class, created using its “template.” It is running code, with specific values stored in each variable
  + To **instantiate** an object is to create a new object from a class
* Object design basics
  + Objects have **attributes**: data stored in the object. This data is different in each instance, although the type of data is defined in the class.
  + Objects have **methods**: functions that use or modify the object’s data. The code for these functions is defined in the class, but it is executed on (and modifies) a specific object
* Encapsulation: An important principle in class/object design
  + Attribute data is stored in **instance variables**, a special kind of variable
  + Called “instance” because each instance, i.e. object, has its own copy of them
  + **Encapsulation** means instance variables (attributes) are “hidden” inside an object: other code cannot access them directly
    - Only the object’s own methods can access the instance variables
    - Other code must “ask permission” from the object in order to read or write the variables
  + **Accessor** method: a method written specifically to allow other code to access instance variables (i.e. attributes)

## Writing Our First Class

* Designing the class
  + Our first class will be used to represent rectangles; each instance (object) will represent one rectangle
  + Attributes of a rectangle:
    - Length
    - Width
  + Methods that will use the rectangle’s attributes
    - Get length
    - Get width
    - Set length
    - Set width
    - Compute the rectangle’s area
  + Note that the first four are **accessor** methods because they allow other code to read (get) or write (set) the rectangle’s instance variables

The Rectangle class:

class Rectangle  
{  
 private int length;  
 private int width;  
  
 public void SetLength(int lengthParameter)  
 {  
 length = lengthParameter;  
 }  
 public int GetLength()  
 {  
 return length;  
 }  
 public void SetWidth(int widthParameter)  
 {  
 width = widthParameter;  
 }  
 public int GetWidth()  
 {  
 return width;  
 }  
 public int ComputeArea()  
 {  
 return length \* width;  
 }  
}

Let’s look at each part of this code in order.

* Attributes
  + Each attribute (length and width) is stored in an instance variable
  + Instance variables are declared similarly to “regular” variables, but with one additional feature: the **access modifier**
  + Syntax: [access modifier] [type] [variable name]
  + The access modifier can be either public or private
  + An access modifier of private is what enforces encapsulation: when you use this access modifier, it means the instance variable cannot be accessed by any code outside the Rectangle class
  + The C# compiler will give you an error if you write code that attempts to use a private instance variable anywhere other than a method of that variable’s class
* SetLength method - our first accessor method
  + This method will allow code outside the Rectangle class to modify a Rectangle object’s “length” attribute
  + Note that the header of this method has an access modifier, just like the instance variable
  + In this case the access modifier is public because we *want* to allow other code to call the SetLength method
  + Syntax of a method declaration: [access modifier] [return type] [method name]([parameters])
  + This method has one **parameter**, named lengthParameter, whose type is int. This means the method must be called with one **argument** that is int type.
    - Similar to how Console.WriteLine must be called with one argument that is string type – the Console.WriteLine declaration has one parameter that is string type.
    - Note that it’s declared just like a variable, with a type and a name
  + A parameter works like a variable: it has a type and a value, and you can use it in expressions and assignment
  + When you call a method with a particular argument, like 15, the parameter is assigned this value, so within the method’s code you can assume the parameter value is “the argument to this method”
  + The body of the SetLength method has one statement, which assigns the instance variable length to the value contained in the parameter lengthParameter. In other words, whatever argument SetLength is called with will get assigned to length
  + This is why it’s called a “setter”: SetLength(15) will set length to 15.
* GetLength method
  + This method will allow code outside the Rectangle class to read the current value of a Rectangle object’s “length” attribute
  + The **return type** of this method is int, which means that the value it returns to the calling code is an int value
  + Recall that Console.ReadLine() returns a string value to the caller, which is why you can write string userInput = Console.ReadLine(). The GetLength method will do the same thing, only with an int instead of a string
  + This method has no parameters, so you don’t provide any arguments when calling it. “Getter” methods never have parameters, since their purpose is to “get” (read) a value, not change anything
  + The body of GetLength has one statement, which uses a new keyword: return. This keyword declares what will be returned by the method, i.e. what particular value will be given to the caller to use in an expression.
  + In a “getter” method, the value we return is the instance variable that corresponds to the attribute named in the method. GetLength returns the length instance variable.
* SetWidth method
  + This is another “setter” method, so it looks very similar to SetLength
  + It takes one parameter (widthParameter) and assigns it to the width instance variable
  + Note that the return type of both setters is void. The return type void means “this method does not return a value.” Console.WriteLine is an example of a void method we’ve used already.
  + Since the return type is void, there is no return statement in this method
* GetWidth method
  + This is the “getter” method for the width attribute
  + It looks very similar to GetLength, except the instance variable in the return statement is width rather than length
* The ComputeArea method
  + This is *not* an accessor method: its goal is not to read or write a single instance variable
  + The goal of this method is to compute and return the rectangle’s area
  + Since the area of the rectangle will be an int (it’s the product of two ints), we declare the return type of the method to be int
  + This method has no parameters, because it doesn’t need any arguments. Its only “input” is the instance variables, and it will always do the same thing every time you call it.
  + The body of the method has a return statement with an expression, rather than a single variable
  + When you write return [expression], the expression will be evaluated first, then the resulting value will be used by the return command
  + In this case, the expression length \* width will be evaluated, which computes the area of the rectangle. Since both length and width are ints, the int version of the \* operator runs, and it produces an int result. This resulting int is what the method returns.

## Using Our Class

* We’ve written a class, but it doesn’t do anything yet
  + The class is a blueprint for an object, not an object
  + To make it “do something” (i.e. execute some methods), we need to instantiate an object using this class
  + The code that does this should be in a separate file (e.g. Program.cs), not in Rectangle.cs
* Here is a program that uses our Rectangle class:

using System;  
  
class Program  
{  
 static void Main(string[] args)  
 {  
 Rectangle myRectangle = new Rectangle();  
 myRectangle.SetLength(12);  
 myRectangle.SetWidth(3);  
 int area = myRectangle.ComputeArea();  
 Console.WriteLine("Your rectangle's length is " +  
 $"{myRectangle.GetLength()}, and its width is " +  
 $"{myRectangle.GetWidth()}, so its area is {area}.");  
 }  
}

* Instantiating an object
  + The first line of code creates a Rectangle object
  + The left side of the = sign is a variable declaration – it declares a variable of type Rectangle
    - Classes we write become new data types in C#
  + The right side of the = sign assigns this variable a value: a Rectangle object
  + We **instantiate** an object by writing the keyword new followed by the name of the class (syntax: new [class name]()). The empty parentheses are required, but we’ll explain why later.
  + This statement is really an initialization statement: It declares and assigns a variable in one line
  + The value of the myRectangle variable is the Rectangle object that was created by new Rectangle()
* Calling setters on the object
  + The next two lines of code call the SetLength and SetWidth methods on the object
  + Syntax: [object name].[method name]([argument]). Note the “dot operator” between the variable name and the method name.
  + SetLength is called with an argument of 12, so lengthParameter gets the value 12, and the rectangle’s length instance variable is then assigned this value
  + Similarly, SetWidth is called with an argument of 3, so the rectangle’s width instance variable is assigned the value 3
* Calling ComputeArea
  + The next line calls the ComputeArea method and assigns its result to a new variable named area
  + The syntax is the same as the other method calls
  + Since this method has a return value, we need to do something with the return value – we assign it to a variable
  + Similar to how you must do something with the result (return value) of Console.ReadLine(), i.e. string userInput = Console.ReadLine()
* Calling getters on the object
  + The last line of code displays some information about the rectangle object using string interpolation
  + One part of the string interpolation is the area variable, which we’ve seen before
  + The other interpolated values are myRectangle.GetLength() and myRectangle.GetWidth()
  + Looking at the first one: this will call the GetLength method, which has a return value that is an int. Instead of storing the return value in an int variable, we put it in the string interpolation brackets, which means it will be converted to a string using ToString. This means the rectangle’s length will be inserted into the string and displayed on the screen

## Flow of Control with Objects

* Consider what happens when you have multiple objects in the same program, like this:
* class Program  
  {  
   static void Main(string[] args)  
   {  
   Rectangle rect1;  
   rect1 = new Rectangle();  
   rect1.SetLength(12);  
   rect1.SetWidth(3);  
   Rectangle rect2 = new Rectangle();  
   rect2.SetLength(7);  
   rect2.SetWidth(15);  
   }  
  }
  + First, we declare a variable of type Rectangle
  + Then we assign rect1 a value, a new Rectangle object that we instantiate
  + We call the SetLength and SetWidth methods using rect1, and the Rectangle object that rect1 refers to gets its length and width instance variables set to 12 and 3
  + Then we create another Rectangle object and assign it to the variable rect2 . This object has its own copy of the length and width instance variables, not 12 and 3
  + We call the SetLength and SetWidth methods again, using rect2 on the left side of the dot instead of rect1. This means the Rectangle object that rect2 refers to gets its instance variables set to 7 and 15, while the other Rectangle remains unmodified
* The same method code can modify different objects at different times
  + Calling a method transfers control from the current line of code (i.e. in Program.cs) to the method code within the class (Rectangle.cs)
  + The method code is always the same, but the specific object that gets modified can be different each time
  + The variable on the left side of the dot operator determines which object gets modified
  + In rect1.SetLength(12), rect1 is the **calling object**, so SetLength will modify rect1
    - SetLength begins executing with lengthParameter equal to 12
    - The instance variable length in length = lengthParameter refers to rect1’s length
  + In rect2.SetLength(7), rect2 is the calling object, so SetLength will modify rect2
    - SetLength begins executing with lengthParameter equal to 7
    - The instance variable length in length = lengthParameter refers to rect2’s length
* Accessing object members
  + The “dot operator” that we use to call methods is technically the **member access operator**
  + A **member** of an object is either a method or an instance variable
  + When we write objectName.methodName(), e.g. rect1.SetLength(12), we are using the dot operator to access the “SetLength” member of rect1, which is a method; this means we want to call (execute) the SetLength method of rect1
  + We can also use the dot operator to access instance variables, although we usually don’t do that because of encapsulation
  + If we wrote the Rectangle class like this:
  + class Rectangle  
    {  
     public int length;  
     public int width;  
    }
  + Then we could write a Main method that uses the dot operator to access the length and width instance variables, like this:
  + static void Main(string[] args)  
    {  
     Rectangle rect1 = new Rectangle();  
     rect1.length = 12;  
     rect1.width = 3;  
    }
* But this code violates encapsulation, so we won’t do this.
* Method calls in more detail
  + Now that we know about the member access operator, we can explain how method calls work a little better
  + When we write rect1.SetLength(12), the SetLength method is executed with rect1 as the calling object – we’re accessing the SetLength member of rect1 in particular (even though every Rectangle has the same SetLength method)
  + This means that when the code in SetLength uses an instance variable, i.e. length, it will automatically access rect1’s copy of the instance variable
  + You can imagine that the SetLength method “changes” to this when you call rect1.SetLength():
  + public void SetLength(int lengthParameter)  
    {  
     rect1.length = lengthParameter;  
    }
  + Note that we use the “dot” (member access) operator on rect1 to access its length instance variable.
  + Similarly, you can imagine that the SetLength method “changes” to this when you call rect2.SetLength():
  + public void SetLength(int lengthParameter)  
    {  
     rect2.length = lengthParameter;  
    }
  + The calling object is automatically “inserted” before any instance variables in a method
  + The keyword this is an explicit reference to “the calling object”
    - Instead of imagining that the calling object’s name is inserted before each instance variable, you could write the SetLength method like this:
    - public void SetLength(int lengthParameter)  
      {  
       this.length = lengthParameter;  
      }
    - This is valid code (unlike our imaginary examples) and will work exactly the same as our previous way of writing SetLength
    - When SetLength is called with rect1.SetLength(12), this becomes equal to rect1, just like lengthParameter becomes equal to 12
    - When SetLength is called with rect2.SetLength(7), this becomes equal to rect2 and lengthParameter becomes equal to 7
* Methods don’t always change instance variables
  + Using a variable in an expression means *reading* its value
  + A variable only changes when it’s on the left side of an assignment statement; this is *writing* to the variable
  + A method that uses instance variables in an expression, but doesn’t assign to them, will not modify the object
  + For example, consider the ComputeArea method:
  + public int ComputeArea()  
    {  
     return length \* width;  
    }
  + It reads the current values of length and width to compute their product, but the product is returned to the method’s caller. The instance variables are not changed.
  + After executing the following code:
  + Rectangle rect1 = new Rectangle();  
    rect1.SetLength(12);  
    rect1.SetWidth(3);  
    int area = rect1.ComputeArea();
  + rect1 has a length of 12 and a width of 3. The call to rect1.ComputeArea() computes , and the area variable is assigned this return value, but it does not change rect1.
* Methods and return values
  + Recall the basic structure of a program: receive input, compute something, produce output
  + A method has the same structure: it *receives input* from its parameters, *computes* by executing the statements in its body, then *produces output* by returning a value
    - For example, consider this method defined in the Rectangle class:
    - public int LengthProduct(int factor)  
      {  
       return length \* factor;  
      }
    - Its input is the parameter factor, which is an int. In the method body, it computes the product of the rectangle’s length and factor. The method’s output is the resulting product.
  + The return statement specifies the output of the method: a variable, expression, etc. that produces some value
  + A method call can be used in other code as if it were a value. The “value” of a method call is the method’s return value.
    - In previous examples, we wrote int area = rect1.ComputeArea();, which assigns a variable (area) a value (the return value of ComputeArea())
    - The LengthProduct method can be used like this:
    - Rectangle rect1 = new Rectangle();  
      rect1.SetLength(12);  
      int result = rect1.LengthProduct(2) + 1;
    - When executing the third line of code, the computer first runs the LengthProduct method with argument (input) 2, which computes the product . Then it uses the return value of LengthProduct, which is 24, to evaluate the expression rect1.LengthProduct(2) + 1, producing a result of 25. Finally, it assigns the value 25 to the variable result.
  + When writing a method that returns a value, the value in the return statement **must** be the same type as the method’s return type
    - If the value returned by LengthProduct is not an int, we’ll get a compile error
    - This won’t work:
    - public int LengthProduct(double factor)  
      {  
       return length \* factor;  
      }
    - Now that factor has type double, the expression length \* factor will need to implicitly convert length from int to double in order to make the types match. Then the product will also be a double, so the return value does not match the return type (int).
    - We could fix it by either changing the return type of the method to double, or adding a cast to int to the product so that the return value is still an int
  + Not all methods return a value, but all methods must have a return type
    - The return type void means “nothing is returned”
    - If your method does not return a value, its return type *must* be void. If the return type is not void, the method *must* return a value.
    - This will cause a compile error because the method has a return type of int but no return statement:
    - public int SetLength(int lengthP)  
      {  
       length = lengthP;  
      }
    - This will cause a compile error because the method has a return type of void, but it attempts to return something anyway:
    - public void GetLength()  
      {  
       return length;  
      }

## Introduction to UML

* UML is a specification language for software
  + UML: Unified Modeling Language
  + Describes design and structure of a program with graphics
  + Does not include “implementation details,” such as code statements
  + Can be used for any programming language, not just C#
  + Used in planning/design phase of software creation, before you start writing code
  + Process: Determine program requirements Make UML diagrams Write code based on UML Test and debug program
* UML Class Diagram elements

| * **ClassName** |
| --- |
| * - attribute: type |
| * ————————————————– |
| * + SetAttribute(attributeParameter: type): void |
| * + GetAttribute(): type |
| * + Method(paramName: type): type |

* + Top box: Class name, centered
  + Middle box: Attributes (i.e. instance variables)
    - On each line, one attribute, with its name and type
    - Syntax: [+/-] [name]: [type]
    - Note this is the opposite order from C# variable declaration: type comes after name
    - Minus sign at beginning of line indicates “private member”
  + Bottom box: Operations (i.e. methods)
    - On each line, one method header, including name, parameters, and return type
    - Syntax: [+/-] [name]([parameter name]: [parameter type]): [return type]
    - Also backwards compared to C# order: parameter types come after parameter names, and return type comes after method name instead of before it
    - Plus sign at beginning of line indicates “public”, which is what we want for methods
* UML Diagram for the Rectangle class

| * **Rectangle** |
| --- |
| * - width: int |
| * - length: int |
| * ———————————— |
| * + SetLength(lengthParameter: int) |
| * + GetLength(): int |
| * + SetWidth(widthParameter: int) |
| * + GetWidth(): int |
| * + ComputeArea(): int |

* + Note that when the return type of a method is void, we can omit it in UML
  + In general, attributes will be private (- sign) and methods will be public (+ sign), so you can expect most of your classes to follow this pattern (-s in the upper box, +s in the lower box)
  + Note that there is no code or “implementation” described here: it doesn’t say that ComputeArea will multiply length by width
* Writing code based on a UML diagram
  + Each diagram is one class, everything within the box is between the class’s header and its closing brace
  + For each attribute in the attributes section, write an instance variable of the right name and type
    - See “- width: int”, write private int width;
    - Remember to reverse the order of name and type
  + For each method in the methods section, write a method header with the matching return type, name, and parameters
    - Parameter declarations are like the instance variables: in UML they have a name followed by a type, in C# you write the type name first
  + Now the method bodies need to be filled in - UML just defined the interface, now you need to write the implementation

## Variable Scope

* Instance variables are different from local variables
  + Instance variables: Stored (in memory) with the object, shared by all methods of the object. Changes made within a method persist after method finishes executing.
  + Local variables: Visible to only one method, not shared. Disappear after method finishes executing. Variables we’ve created before in the Main method (they’re local to the Main method!).
  + Example: In class Rectangle, we have these two methods:
  + public void SwapDimensions()  
    {  
     int temp = length;  
     length = width;  
     width = temp;  
    }  
    public int GetLength()  
    {  
     return length;  
    }
    - temp is a local variable within SwapDimensions, while length and width are instance variables
    - The GetLength method can’t use temp; it is visible only to SwapDimensions
    - When SwapDimensions changes length, that change is persistent – it will still be different when GetLength executes, and the next call to GetLength after SwapDimensions will return the new length
    - When SwapDimensions assigns a value to temp, it only has that value within the current call to SwapDimensions – after SwapDimensions finishes, temp disappears, and the next call to SwapDimensions creates a new temp
* Each variable has a **scope**
  + Variables exist only in limited **time** and **space** within the program
  + Outside those limits, the variable cannot be accessed – e.g. local variables cannot be accessed outside their method
  + Scope of a variable: The region of the program where it is accessible/visible
    - A variable is “in scope” when it is accessible
    - A variable is “out of scope” when it doesn’t exist or can’t be accessed
  + Time limits to scope: Scope begins *after* the variable has been declared
    - This is why you can’t use a variable before declaring it
  + Space limits to scope: Scope is within the same *code block* where the variable is declared
    - Code blocks are defined by curly braces: everything between matching { and } is in the same code block
    - Instance variables are declared in the class’s code block (they are inside class Rectangle’s body, but not inside anything else), so their scope extends to the entire class
    - Code blocks nest: A method’s code block is inside the class’s code block, so instance variables are also in scope within each method’s code block
    - Local variables are declared inside a method’s code block, so their scope is limited to that single method
  + The scope of a parameter (which is a variable) is the method’s code block - it’s the same as a local variable for that method
  + Scope example:
  + public void SwapDimensions()  
    {  
     int temp = length;  
     length = width;  
     width = temp;  
    }  
    public void SetWidth(int widthParam)  
    {  
     int temp = width;  
     width = widthParam;  
    }
    - The two variables named temp have different scopes: One has a scope limited to the SwapDimensions method’s body, while the other has a scope limited to the SetWidth method’s body
    - This is why they can have the same name: variable names must be unique *within the variable’s scope*. You can have two variables with the same name if they are in different scopes.
    - The scope of instance variables length and width is the body of class Rectangle, so they are in scope for both of these methods
* Be careful when mixing instance and local variables
  + This code is legal (compiles) but doesn’t do what you want:
  + class Rectangle  
    {  
     private int length;  
     private int width;  
     public void UpdateWidth(int newWidth)  
     {  
     int width = 5;  
     width = newWidth;  
     }  
    }
  + The instance variable width and the local variable width have different scopes, so they can have the same name
  + But the instance variable’s scope (the class Rectangle) *overlaps* with the local variable’s scope (the method UpdateWidth)
  + If two variables have the same name and overlapping scopes, the variable with the *closer* or *smaller* scope **shadows** the variable with the *farther* or *wider* scope: the name will refer *only* to the variable with the smaller scope
  + In this case, that means width inside UpdateWidth refers only to the local variable named width, whose scope is smaller because it is limited to the UpdateWidth method. The line width = newWidth actually changes the local variable, not the instance variable named width.
  + Since instance variables have a large scope (the whole class), they will always get shadowed by variables declared within methods
  + You can prevent shadowing by using the keyword this, like this:
  + class Rectangle  
    {  
     private int length;  
     private int width;  
     public void UpdateWidth(int newWidth)  
     {  
     int width = 5;  
     this.width = newWidth;  
     }  
    }
  + Since this means “the calling object”, this.width means “access the width member of the calling object.” This can only mean the instance variable width, not the local variable with the same name
  + Incidentally, you can also use this to give your parameters the same name as the instance variables they’re modifying:
  + class Rectangle  
    {  
     private int length;  
     private int width;  
     public void SetWidth(int width)  
     {  
     this.width = width;  
     }  
    }
  + Without this, the body of the SetWidth method would be width = width;, which doesn’t do anything (it would assign the parameter width to itself).

## Constants

* Classes can also contain constants
* Syntax: [public/private] const [type] [name] = [value];
* This is a named value that never changes during program execution
* Safe to make it public because it can’t change – no risk of violating encapsulation
* Can only be built-in types (int, double, etc.), not objects
* Can make your program more readable by giving names to “magic numbers” that have some significance
* Convention: constants have names in ALL CAPS
* Example:
* class Calendar  
  {  
   public const int MONTHS = 12;  
   private int currentMonth;  
   //...  
  }
* The value “12” has a special meaning here, i.e. the number of months in a year, so we use a constant to name it.
* Constants are accessed using the name of the class, not the name of an object – they are the same for every object of that class. For example:
* Calendar myCal = new Calendar();  
  decimal yearlyPrice = 2000.0m;  
  decimal monthlyPrice = yearlyPrice / Calendar.MONTHS;

## Reference Types: More Details

* Data types in C# are either value types or reference types
  + This difference was introduced in an earlier lecture (Datatypes and Variables)
  + For a **value type** variable (int, long, float, double, decimal, char, bool) the named memory location stores the exact data value held by the variable
  + For a **reference type** variable, such as string, the named memory location stores a *reference to the value*, not the value itself
  + All objects you create from your own classes, like Rectangle, are reference types
* Object variables are references
  + When you have a variable for a reference type, or “reference variable,” you need to be careful with the assignment operation
  + Consider this code:
  + using System;  
    class Program  
    {  
     static void Main(string[] args)  
     {  
     Rectangle rect1 = new Rectangle();  
     rect1.SetLength(8);  
     rect1.SetWidth(10);  
     Rectangle rect2 = rect1;  
     rect2.SetLength(4);  
     Console.WriteLine($"Rectangle 1: {rect1.GetLength()} "  
     + $"by {rect1.GetWidth()}");  
     Console.WriteLine($"Rectangle 2: {rect2.GetLength()} "  
     + $"by {rect2.GetWidth()}");  
     }  
    }
  + The output is:
  + Rectangle 1: 4 by 10  
    Rectangle 2: 4 by 10
  + The variables rect1 and rect2 actually refer to the same Rectangle object, so rect2.SetLength(4) seems to change the length of “both” rectangles
  + The assignment operator copies the contents of the variable, but a reference variable contains a *reference* to an object – so that’s what gets copied (in Rectangle rect2 = rect1), not the object itself
  + In more detail:
    - Rectangle rect1 = new Rectangle() creates a new Rectangle object somewhere in memory, then creates a reference variable named rect1 somewhere else in memory. The variable named rect1 is initialized with the memory address of the Rectangle object, i.e. a reference to the object
    - rect1.SetLength(8) reads the address of the Rectangle object from the rect1 variable, finds the object in memory, and runs the SetLength method on that object (changing its length to 8)
    - rect1.SetWidth(10) does the same thing, finds the same object, and sets its width to 10
    - Rectangle rect2 = rect1 creates a reference variable named rect2 in memory, but does not create a new Rectangle object. Instead, it initializes rect2 with the same memory address that is stored in rect1, referring to the same Rectangle object
    - rect2.SetLength(4) reads the address of a Rectangle object from the rect2 variable, finds that object in memory, and sets its length to 4 – but this is the exact same Rectangle object that rect1 refers to
* Reference types can also appear in method parameters
  + When you call a method, you provide an argument (a value) for each parameter in the method’s declaration
  + Since the parameter is really a variable, the computer will then assign the argument to the parameter, just like variable assignment
    - For example, when you write rect1.SetLength(8), there’s an implicit assignment lengthParameter = 8 that gets executed before executing the body of the SetLength method
  + This means if the parameter is a reference type (like an object), the parameter will get a copy of the reference, not a copy of the object
  + When you use the parameter to modify the object, you will modify the same object that the caller provided as an argument
  + This means objects can change other objects!
  + For example, imagine we added this method to the Rectangle class:
  + public void CopyToOther(Rectangle otherRect)  
    {  
     otherRect.SetLength(length);  
     otherRect.SetWidth(width);  
    }
  + It uses the SetLength and SetWidth methods to modify its parameter, otherRect. Specifically, it sets the parameter’s length and width to its own length and width.
  + The Main method of a program could do something like this:
  + Rectangle rect1 = new Rectangle();  
    Rectangle rect2 = new Rectangle();  
    rect1.SetLength(8);  
    rect1.SetWidth(10);  
    rect1.CopyToOther(rect2);  
    Console.WriteLine($"Rectangle 2: {rect2.GetLength()} "  
     + $"by {rect2.GetWidth()}");
    - First it creates two different Rectangle objects (note the two calls to new), then it sets the length and width of one object, using rect1.SetLength and rect1.SetWidth
    - Then it calls the CopyToOther method with an argument of rect2. This transfers control to the method and (implicitly) makes the assignment otherRect = rect2
    - Since otherRect and rect2 are now reference variables referring to the same object, the calls to otherRect.SetLength and otherRect.SetWidth within the method will modify that object
    - After the call to CopyToOther, the object referred to by rect2 has a length of 8 and a width of 10, even though we never called rect2.SetLength or rect2.SetWidth

# More Advanced Object Concepts

## Default Values and the ClassRoom Class

* Instance variables get default values
  + In lab, you were asked to run a program like this:
  + using System;  
      
    class Program  
    {  
     static void Main(string[] args)  
     {  
     Rectangle myRect = new Rectangle();  
     Console.WriteLine($"Length is {myRect.GetLength()}");  
     Console.WriteLine($"Width is {myRect.GetWidth()}");  
     }  
    }
  + Note that we create a Rectangle object, but do not use the SetLength or SetWidth methods to assign values to its instance variables. It displays the following output:
  + Length is 0  
    Width is 0
  + This works because the instance variables length and width have a default value of 0, even if you never assign them a value
  + Local variables, like the ones we write in the Main method, do *not* have default values. You must assign them a value before using them in an expression.
    - For example, this code will produce a compile error:
    - int myVar1;  
      int myVar2 = myVar1 + 5;
    - You can’t assume myVar1 will be 0; it has no value at all until you use an assignment statement.
  + When you create (instantiate) a new object, its instance variables will be assigned specific default values based on their type:

| * + **Type** | * + **Default Value** |
| --- | --- |
| * + Numeric types | * + 0 |
| * + string | * + null |
| * + objects | * + null |
| * + bool | * + false |
| * + char | * + '\0' |

* + Remember, null is the value of a reference-type variable that refers to “nothing” - it does not contain the location of any object at all. You can’t do anything with a reference variable containing null.
* A class we’ll use for subsequent examples
  + ClassRoom: Represents a room in a building on campus
  + UML Diagram:

| * + **ClassRoom** |
| --- |
| * + - building: string |
| * + - number: int |
| * + ————————————— |
| * + + SetBuilding(buildingParam : string) |
| * + + GetBuilding(): string |
| * + + SetNumber(numberParameter: int) |
| * + + GetNumber(): int |

* + - There are two attributes: the name of the building (a string) and the room number (an int)
    - Each attribute will have a “getter” and “setter” method
  + Implementation:
  + class ClassRoom  
    {  
     private string building;  
     private int number;  
      
     public void SetBuilding(string buildingParam)  
     {  
     building = buildingParam;  
     }  
     public string GetBuilding()  
     {  
     return building;  
     }  
     public void SetNumber(int numberParam)  
     {  
     number = numberParam;  
     }  
     public int GetNumber()  
     {  
     return number;  
     }  
    }
    - Each attribute is implemented by an instance variable with the same name
    - To write the “setter” for the building attribute, we write a method whose return type is void, with a single string-type parameter. Its body assigns the building instance variable to the value in the parameter buildingParam
    - To write the “getter” for the building attribute, we write a method whose return type is string, and whose body returns the instance variable building
  + Creating an object and using its default values:
  + using System;  
      
    class Program  
    {  
     static void Main(string[] args)  
     {  
     ClassRoom english = new ClassRoom();  
     Console.WriteLine($"Building is {english.GetBuilding()}");  
     Console.WriteLine($"Room number is {english.GetNumber()}");  
     }  
    }
  + This will print the following output:
  + Building is  
    Room number is 0
  + Remember that the default value of a string variable is null. When you use string interpolation on null, you get an empty string.

## Constructors

* Instantiation and constructors
  + Instantiation syntax requires you to write parentheses after the name of the class, like this:
  + ClassRoom english = new ClassRoom();
  + Parentheses indicate a method call, like in Console.ReadLine() or english.GetBuilding()
  + In fact, the instantiation statement new ClassRoom() does call a method: the **constructor**
  + Constructor: A special method used to create an object. It “sets up” a new instance by **initializing its instance variables**.
  + If you don’t write a constructor in your class, C# will generate a “default” constructor for you – this is what’s getting called when we write new ClassRoom() here
  + The default constructor initializes each instance variable to its default value – that’s where default values come from
* Writing a constructor
  + Example for ClassRoom:
  + public ClassRoom(string buildingParam, int numberParam)  
    {  
     building = buildingParam;  
     number = numberParam;  
    }
  + To write a constructor, write a method whose name is *exactly the same* as the class name
  + This method has *no return type*, not even void. It does not have a return statement either
  + For ClassRoom, this means the constructor’s header starts with public ClassRoom
    - You can think of this method as “combining” the return type and name. The name of the method is ClassRoom, and its output is of type ClassRoom, since the return value of new ClassRoom() is always a ClassRoom object
    - You don’t actually write a return statement, though, because new will always return the new object after calling the constructor
  + A custom constructor usually has parameters that correspond to the instance variables: for ClassRoom, it has a string parameter named buildingParam, and an int parameter named numberParam
    - Note that when we write a method with two parameters, we separate the parameters with a comma
  + The body of a constructor must assign values to **all** instance variables in the object
  + Usually this means assigning each parameter to its corresponding instance variable: initialize the instance variable to equal the parameter
    - Very similar to calling both “setters” at once
* Using a constructor
  + An instantiation statement will call a constructor for the class being instantiated
  + Arguments in parentheses must match the parameters of the constructor
  + Example with the ClassRoom constructor:
  + using System;  
      
    class Program  
    {  
     static void Main(string[] args)  
     {  
     ClassRoom csci = new ClassRoom("Allgood East", 356);  
     Console.WriteLine($"Building is {csci.GetBuilding()}");  
     Console.WriteLine($"Room number is {csci.GetNumber()}");  
     }  
    }
  + This program will produce this output:
  + Building is Allgood East  
    Room number is 356
  + The instantiation statement new ClassRoom("Allgood East", 356) first creates a new “empty” object of type ClassRoom, then calls the constructor to initialize it. The first argument, “Allgood East”, becomes the constructor’s first parameter (buildingParam), and the second argument, 356, becomes the constructor’s second parameter (numberParam).
  + After executing the instantiation statement, the object referred to by csci has its instance variables set to these values, even though we never called SetBuilding or SetNumber
* Methods with multiple parameters
  + The constructor we wrote is an example of a method with two parameters
  + The same syntax can be used for ordinary, non-constructor methods, if we need more than one input value
  + For example, we could write this method in the Rectangle class:
  + public void MultiplyBoth(int lengthFactor, int widthFactor)  
    {  
     length \*= lengthFactor;  
     width \*= widthFactor;  
    }
  + The first parameter has type int and is named lengthFactor. The second parameter has type int and is named widthFactor
  + You can call this method by providing two arguments, separated by a comma:
  + Rectangle myRect = new Rectangle();  
    myRect.SetLength(5);  
    myRect.SetWidth(10);  
    myRect.MultiplyBoth(3, 5);
  + The first argument, 3, will be assigned to the first parameter, lengthFactor. The second argument, 5, will be assigned to the second parameter, widthFactor
  + The order of the arguments matters when calling a multi-parameter method. If you write myRect.MultiplyBoth(5, 3), then lengthFactor will be 5 and widthFactor will be 3.
  + The type of each argument must match the type of the corresponding parameter. For example, when you call the ClassRoom constructor we just wrote, the first argument must be a string and the second argument must be an int
* Writing multiple constructors
  + Remember that if you don’t write a constructor, C# generates a “default” one with no parameters, so you can write new ClassRoom()
  + Once you add a constructor to your class, C# will **not** generate a default constructor
    - This means once we write the ClassRoom constructor (as shown earlier), this statement will produce a compile error: ClassRoom english = new ClassRoom();
    - The constructor we wrote has 2 parameters, so now you always need 2 arguments to instantiate a ClassRoom
  + If you still want the option to create an object with no arguments (i.e. new ClassRoom()), you must write a constructor with no parameters
  + A class can have more than one constructor, so it would look like this:
  + class ClassRoom  
    {  
     //...  
     public ClassRoom(string buildingParam, int numberParam)  
     {  
     building = buildingParam;  
     number = numberParam;  
     }  
     public ClassRoom()  
     {  
     building = null;  
     number = 0;  
     }  
     //...  
    }
  + The “no-argument” constructor must still initialize all the instance variables, even though it has no parameters
    - You can pick any “default value” you want, or use the same ones that C# would use (0 for numeric variables, null for object variables, etc.)
  + When a class has multiple constructors, the instantiation statement must decide which constructor to call
  + The instantiation statement will call the constructor whose parameters match the arguments you provide
    - For example, each of these statements will call a different constructor:
    - ClassRoom csci = new ClassRoom("Allgood East", 356);  
      ClassRoom english = new ClassRoom();
    - The first statement calls the two-parameter constructor we wrote, since it has a string argument and an int argument (in that order), and those match the parameters (string buildingParam, int numberParam). The second statement calls the zero-parameter constructor since it has no arguments.
    - If the arguments don’t match any constructor, it’s still an error:
    - ClassRoom csci = new ClassRoom(356, "Allgood East");
    - This will produce a compile error, because the instantiation statement has two arguments in the order int, string, but the only constructor with two parameters needs the first parameter to be a string.

## Writing ToString Methods

* ToString recap
  + String interpolation automatically calls the ToString method on each variable or value
  + ToString returns a string “equivalent” to the object; for example, if num is an int variable containing 42, num.ToString() returns “42”.
  + C# datatypes already have a ToString method, but you need to write a ToString method for your own classes to use them in string interpolation
* Writing a ToString method
  + To add a ToString method to your class, you must write this header: public override string ToString()
  + The access modifier must be public (so other code, like string interpolation, can call it)
  + The return type must be string (ToString must output a string)
  + It must have no parameters (the string interpolation code won’t know what arguments to supply)
  + The keyword override means your class is “overriding,” or providing its own version of, a method that is already defined elsewhere – ToString is defined by the base object type, which is why string interpolation “knows” it can call ToString on any object
    - If you do not use the keyword override, then the pre-existing ToString method (defined by the base object type) will be used instead, which only returns the name of the class
  + The goal of ToString is to return a “string representation” of the object, so the body of the method should use all of the object’s attributes and combine them into a string somehow
  + Example ToString method for ClassRoom:
  + public override string ToString()  
    {  
     return building + " " + number;  
    }
    - There are two instance variables, building and number, and we use both of them
    - A natural way to write the name of a classroom is the building name followed by the room number, like “University Hall 124”, so we concatenate the variables in that order
    - Note that we add a space between the variables
    - Note that building is already a string, but number is an int, so string concatenation will implicitly call number.ToString() – ToString methods can call other ToString methods
    - Another way to write the body would be return $"{building} {number}";
* Using a ToString method
  + Any time an object is used in string interpolation or concatenation, its ToString method will be called
  + You can also call ToString by name using the “dot operator,” like any other method
  + This code will call the ToString method we just wrote for ClassRoom:
  + ClassRoom csci = new ClassRoom("Allgood East", 356);  
    Console.WriteLine(csci);  
    Console.WriteLine($"The classroom is {csci}");  
    Console.WriteLine("The classroom is " + csci.ToString());

## Method Signatures and Overloading

* Name uniqueness in C#
  + In general, variables, methods, and classes must have unique names, but there are several exceptions
  + **Variables** can have the same name if they are in *different scopes*
    - Two methods can each have a local variable with the same name
    - A local variable (scope limited to the method) can have the same name as an instance variable (scope includes the whole class), but this will result in **shadowing**
  + **Classes** can have the same name if they are in *different namespaces*
    - This is one reason C# has namespaces: you can name your classes anything you want. Otherwise, if a library (someone else’s code) used a class name, you would be prevented from using that name
    - For example, imagine you were using a “shapes library” that provided a class named Rectangle, but you also wanted to write your own class named Rectangle
    - The library’s code would use its own namespace, like this:
    - namespace ShapesLibrary  
      {  
       class Rectangle  
       {  
       //instance variables, methods, etc.  
       }  
      }
    - Then your own code could have a Rectangle class in your own namespace:
    - namespace MyProject  
      {  
       class Rectangle  
       {  
       //instance variables, methods, etc.  
       }  
      }
    - You can use both Rectangle classes in the same code, as long as you specify the namespace, like this:
    - MyProject.Rectangle rect1 = new MyProject.Rectangle();  
      ShapesLibrary.Rectangle rect2 = new ShapesLibrary.Rectangle();
  + **Methods** can have the same name if they have *different signatures*; this is called **overloading**
    - We’ll explain signatures in more detail in a minute
    - Briefly, methods can have the same name if they have different parameters
    - For example, you can have two methods named Multiply in the Rectangle class, as long as one has one parameter and the other has two parameters:
    - public void Multiply(int factor)  
      {  
       length \*= factor;  
       width \*= factor;  
      }  
      public void Multiply(int lengthFactor, int widthFactor)  
      {  
       length \*= lengthFactor;  
       width \*= widthFactor;  
      }
    - C# understands that these are different methods, even though they have the same name, because their parameters are different. If you write myRect.Multiply(2) it can only mean the first “Multiply” method, not the second one, because there is only one argument.
    - We have used overloading already when we wrote multiple constructors – constructors are methods too. For example, these two constructors have the same name, but different parameters:
    - public ClassRoom(string buildingParam, int numberParam)  
      {  
       building = buildingParam;  
       number = numberParam;  
      }  
      public ClassRoom()  
      {  
       building = null;  
       number = 0;  
      }
* Method signatures
  + A method’s **signature** has 3 components: its **name**, the **type** of each parameter, and the **order** the parameters appear in
  + Methods are unique if their *signatures* are unique, which is why they can have the same name
  + Signature examples:
    - public void Multiply(int lengthFactor, int widthFactor) – the signature is Multiply(int, int) (name is Multiply, parameters are int and int type)
    - public void Multiply(int factor) – signature is Multiply(int)
    - public void Multiply(double factor) – signature is Multiply(double)
    - These could all be in the same class since they all have different signatures
  + Parameter *names* are not part of the signature, just their types
    - Note that the parameter names are omitted when I write down the signature
    - That means these two methods are not unique and could not be in the same class:
    - public void SetWidth(int widthInMeters)  
      {  
       //...  
      }  
      public void SetWidth(int widthInFeet)  
      {  
       //...  
      }
    - Both have the same signature, SetWidth(int), even though the parameters have different names. You might intend the parameters to be different (i.e. represent feet vs. meters), but any int-type parameter is the same to C#
  + The method’s return type is not part of the signature
    - So far all the examples have the same return type (void), but changing it would not change the signature
    - The signature of public int Multiply(int factor) is Multiply(int), which is the same as public void Multiply(int factor)
    - The signature “begins” with the name of the method; everything “before” that doesn’t count (i.e. public, int)
  + The order of parameters is part of the signature, as long as the types are different
    - Since parameter name is not part of the signature, only the type can determine the order
    - These two methods have different signatures:
    - public int Update(int number, string name)  
      {  
       //...  
      }  
      public int Update(string name, int number)  
      {  
       //..  
      }
    - The signature of the first method is Update(int, string). The signature of the second method is Update(string, int).
    - These two methods have the same signature, and could not be in the same class:
    - public void Multiply(int lengthFactor, int widthFactor)  
      {  
       //...  
      }  
      public void Multiply(int widthFactor, int lengthFactor)  
      {  
       //...  
      }
    - The signature for both methods is Multiply(int, int), even though we switched the order of the parameters – the name doesn’t count, and they are both int type
  + Constructors have signatures too
    - The constructor ClassRoom(string buildingParam, int numberParam) has the signature ClassRoom(string, int)
    - The constructor ClassRoom() has the signature ClassRoom()
    - Constructors all have the same name, but they are unique if their signatures (parameters) are different
* Calling overloaded methods
  + Previously, when you used the dot operator and wrote the name of a method, the name was enough to determine which method to run – myRect.GetLength() would call the GetLength method
  + When a method is overloaded, you must use the entire signature to determine which method gets executed
  + A method call has a “signature” too: the name of the method, and the type and order of the arguments
  + C# will execute the method whose signature matches the signature of the method call
  + Example: myRect.Multiply(4); has the signature Multiply(int), so C# will look for a method in the Rectangle class that has the signature Multiply(int). This matches the method public void Multiply(int factor)
  + Example: myRect.Multiply(3, 5); has the signature Multiply(int, int), so C# will look for a method with that signature in the Rectangle class. This matches the method public void Multiply(int lengthFactor, int widthFactor)
  + The same process happens when you instantiate a class with multiple constructors: C# calls the constructor whose signature matches the signature of the instantiation
  + If no method or constructor matches the signature of the method call, you get a compile error. You still can’t write myRect.Multiply(1.5) if there is no method whose signature is Multiply(double).

## Constructors in UML

* Now that we can write constructors, they should be part of the UML diagram of a class
  + No need to include the default constructor, or one you write yourself that takes no arguments
  + Non-default constructors go in the operations section (box 3) of the UML diagram
  + Similar syntax to a method: [+/-] <<constructor>> [name]([parameter name]: [parameter type])
  + Note that the name will always match the class name
  + No return type, ever
  + Annotation “<<constructor>>” is nice, but not necessary: if the method name matches the class name, it’s a constructor
* Example for ClassRoom:

| * **ClassRoom** |
| --- |
| * - building: string |
| * - number: int |
| * —————————————————————————- |
| * + <<constructor>> ClassRoom(buildingParam: string, numberParam: int) |
| * + SetBuilding(buildingParam : string) |
| * + GetBuilding(): string |
| * + SetNumber(numberParameter: int) |
| * + GetNumber(): int |

## Properties

* Attributes are implemented with a standard “template” of code
  + Remember, “attribute” is the abstract concept of some data stored in an object; “instance variable” is the way that data is actually stored
  + First, declare an instance variable for the attribute
  + Then write a “getter” method for the instance variable
  + Then write a “setter” method for the instance variable
  + With this combination of instance variable and methods, the object has an attribute that can be read (with the getter) and written (with the setter)
  + For example, this code implements a “width” attribute for the class Rectangle:
  + class Rectangle  
    {  
     private int width;  
     public void SetWidth(int value)  
     {  
     width = value;  
     }  
     public int GetWidth()  
     {  
     return width;  
     }  
    }
  + Note that there’s a lot of repetitive or “obvious” code here:
    - The name of the attribute is intended to be “width,” so you must name the instance variable width, and the methods GetWidth and SetWidth, repeating the name three times.
    - The attribute is intended to be type int, so you must ensure that the instance variable is type int, the getter has a return type of int, and the setter has a parameter type of int. Similarly, this repeats the data type three times.
    - You need to come up with a name for the setter’s parameter, even though it also represents the width (i.e. the new value you want to assign to the width attribute). We usually end up naming it “widthParameter” or “widthParam” or “newWidth” or “newValue.”
  + Properties are a “shorthand” way of writing this code: They implement an attribute with less repetition
* Writing properties
  + Declare an instance variable for the attribute, like before
  + A **property declaration** has 3 parts:
    - Header, which gives the property a name and type (very similar to variable declaration)
    - get section, which declares the “getter” method for the property
    - set section, which declares the “setter” method for the property
  + Example code, implementing the “width” attribute for Rectangle (this replaces the code in the previous example):
  + class Rectangle  
    {  
     private int width;  
     public int Width  
     {  
     get  
     {  
     return width;  
     }  
     set  
     {  
     width = value;  
     }  
     }  
    }
  + Header syntax: [public/private] [type] [name]
  + *Convention* (not rule) is to give the property the same name as the instance variable, but capitalized – C# is case sensitive
  + get section: Starts with the keyword get, then a method body inside a code block (between braces)
    - get is like a method header that always has the same name, and its other features are implied by the property’s header
    - Access modifier: Same as the property header’s, i.e. public in this example
    - Return type: Same as the property header’s type, i.e. int in this example (so imagine it says public int get())
    - Body of get section is exactly the same as body of a “getter”: return the instance variable
  + set section: Starts with the keyword set, then a method body inside a code block
    - Also a method header with a fixed name, access modifier, return type, and parameter
    - Access modifier: Same as the property header’s, i.e. public in this example
    - Return type: Always void (like a setter)
    - Parameter: Same type as the property header’s type, name is always “value”. In this case that means the parameter is int value; imagine the method header says public void set(int value)
    - Body of set section looks just like the body of a setter: Assign the parameter to the instance variable (and the parameter is always named “value”). In this case, that means width = value
* Using properties
  + Properties are members of an object, just like instance variables and methods
  + Access them with the “member access” operator, aka the dot operator
    - For example, myRect.Width will access the property we wrote, assuming myRect is a Rectangle
  + A complete example, where the “length” attribute is implemented the “old” way with a getter and setter, and the “width” attribute is implemented with a property:
  + using System;  
      
    class Program  
    {  
     static void Main(string[] args)  
     {  
     Rectangle myRectangle = new Rectangle();  
     myRectangle.SetLength(6);  
     myRectangle.Width = 15;  
     Console.WriteLine("Your rectangle's length is " +  
     $"{myRectangle.GetLength()}, and " +  
     $"its width is {myRectangle.Width}");  
     }  
    }
  + Properties “act like” variables: you can assign to them and read from them
  + Reading from a property will *automatically* call the get method for that property
    - For example, Console.WriteLine($"The width is {myRectangle.Width}"); will call the get section inside the Width property, which in turn executes return width and returns the current value of the instance variable
    - This is equivalent to Console.WriteLine($"The width is {myRectangle.GetWidth()}"); using the “old” Rectangle code
  + Assigning to (writing) a property will *automatically* call the set method for that property, with an argument equal to the right side of the = operator
    - For example, myRectangle.Width = 15; will call the set section inside the Width property, with value equal to 15
    - This is equivalent to myRectangle.SetWidth(15); using the “old” Rectangle code
* Properties in UML
  + Since properties represent attributes, they go in the “attributes” box (the second box)
  + If a property will simply “get” and “set” an instance variable of the same name, you do *not* need to write the instance variable in the box
    - No need to write both the property Width and the instance variable width
  + Syntax: [+/-] <<property>> [name]: [type]
  + Note that the access modifier (+ or -) is for the property, not the instance variable, so it’s + if the property is public (which it usually is)
  + Example for Rectangle, assuming we converted both attributes to use properties instead of getters and setters:

| * + **Rectangle** |
| --- |
| * + + <<property>> Width: int |
| * + + <<property>> Length: int |
| * + ———————————— |
| * + + ComputeArea(): int |

* + We no longer need to write all those setter and getter methods, since they are “built in” to the properties

# Decisions and Decision Structures

Decisions are a constant occurence in daily life. For instance consider an instructor teaching CSCI 1301. At the beginning of class the instructor may

* Ask if there are questions. If a student has a question, then the instructor will answer it, and ask again (“Anything else?”).
* When there are no more questions, the instructor will move on to the next step.
* If there is a quiz scheduled, the next step will be distributing the quiz.
* If there is no quiz scheduled or the quiz is complete (and collected), the instructor may introduce the lecture topic (“Today, we will be discussing Decisions and Decision Structures”) and start the class.
* etc.

This type of “branching” between multiple choices can be represented with an [activity diagram](https://en.wikipedia.org/wiki/Activity_diagram):

“An Activity Diagram on Teaching a Class”

“An Activity Diagram on Teaching a Class”

In C#, we will express

* repetitions (or “loops”) (“As long as there are questions…”) with the while, do…while and for keywords,
* branchings (“If there is a quiz…”) with the if, if…else and switch keywords.

Both structures need a datatype to express the result of a decision (“Is it *true* that there are questions.”, or “Is it *false* that there is a quiz.”) called booleans. Boolean values can be set with conditions, that can be composed in different ways using three operators (“and”, “or” and “not”). For example, “If today is a Monday, and it is not past 10:10 am, and if mid-term has not been given yet, the class will also include a brief reminder about the upcoming first exam.”

# Boolean Variables and Values

## Variables

We can store if something is true or false (“The user has reached the age of majority”, “The switch is on”, “The user is using Windows”, “This computer’s clock indicates that we are in the afternoon”, …) in a (boolean) *flag*, which is simply a variable of type boolean. Note that true and false are the only possible two values for boolean variables: there is no third option!

We can declare, assign, initialize and display a boolean variable (flag) as with any other variable:

bool learning\_how\_to\_program = true;  
Console.WriteLine(learning\_how\_to\_program);

## Operations on Boolean Values

Despite having only two possible values in true and false, we can use the following three operations to construct more complex booleans:

1. “and” (&&, conjunction),
2. “or” (||, disjunction),
3. and “not” (!, negation).

Each has the precise meaning described here:

1. the condition “A and B” is true if and only if A is true, and B is true,
2. “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),
3. “not A” is true if and only if A is false (that is, “not” “flips” the value it is applied to).

The expected results of these operations can be displayed in *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 |

These tables can also be written in -dimensions, as can be seen for [conjunction on wikipedia](https://en.wikipedia.org/wiki/Truth_table#Logical_conjunction_(AND)).

# Equality and Relational Operators

Boolean values can also be set through expressions, or tests, that “evaluate” a condition or series of conditions as true or false. For instance, you can write an expression meaning “variable myAge has the value 12” which will evaluate to true if the value of myAge is indeed 12, and to false otherwise. *To ease your understanding*, we will write “expression true” to indicate that “expression” evaluates to true below, but this is *not* part of C#’s syntax.

Here we use two kinds of operators: - Equality operators test if two values (literal or variable) are the same. This works on all datatypes. - Relational operators test if a value (literal or variable) is greater or smaller (strictly or largely) than an other value or variable.

Relational operators will be primarily used for numerical values.

## Equality Operators

| **Equality Operators** |  |  |
| --- | --- | --- |
| Mathematical Notation | C# Notation | Example |
|  | == | 3 == 4 false |
|  | != | 3!=4 true |

We can test numerical values for equality, as well as string, char and bool!

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

## Relational Operators

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

| **Relational Operators** |  |  |
| --- | --- | --- |
| Mathematical Notation | C# Notation | Example |
|  | > | 3 > 4 false |
|  | < | 3 < 4 true |
|  | >= | 3 >= 4 false |
|  | <= | 3 <= 4 true |

Relational operators can also compare char, but the order is a bit complex (you can find it explained, for instance, [in this stack overflow answer](https://stackoverflow.com/a/14967721/)).

All of the operators explained in this lecture have an order (or precedence) in which they are evaluated. The precedence is as follows:

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

* Operators with higher precedence are on the left and operators with lower precedence are on the right: for instance, in an expression like 2\*3+4, 2\*3 will have higher precedence than 3+4, and thus be evaluated first: 2\*3+4 is to be read as (2\*3)+4 = 6 + 4 = 10 and *not* as 2\*(3+4) = 2\*7 = 14.
* Operators in parenthesis have equal precedence and are evaluated in the order they appear, from left to right: in 1-2+3, 1-2 will be evaluated before 2+3: 1-2+3 is to be read as (1-2)+3 = -1 + 3 = 2 and *not* as 1-(2+3) = 1-5 = 4.

# if Statement

## First Example

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

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

## 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.

# if-else Statements

## Syntax

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

With if-else statements, statement block 1 is executed only if the condition evaluates to true, and block 2 is executed 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 blocks will be executed. Since a condition cannot be true and false at the same time, we also know that at most one block will be executed. Hence, exactly one block will be executed.

# Nested if-else Statements

A <statement block> can actually include an if-else statement itself!



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

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!
* We could remove the braces since each condition corresponds to exactly one statement.
* We could have a similar flavor with only if: if(age > 18 && usCitizen) … else …, but the messages would be less accurate.

# 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!

## Example

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

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

Giving various values for age, charVar and boolFlag, we can see which value would x get in each case.

# ?: 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>.

# Boolean Flags

Do you 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 works 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 characters.  
  
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 there aren’t three answers to this question (you either work here full-time, or you don’t), so we can change the behavior to

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

We’ll study *user input validation*, that allows us to require better answers from users, later on.

But imagine we are at the beginning of a long form, and we will need to re-use that information multiple times. If we use this code fragment, we would need to duplicate all our code. 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 understand the ? operator (from the link in lecture or from 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 long, convoluted code, to a very simple line! We already did this trick last time, but I thought that seeing it again would help.

# Constructing a Value Progressively

In the if statement lab, you are 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 answer 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 “progressively” construct 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 independent, we can test them in two different if statements!

# 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 (<literal 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 behavior:

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;  
}

# Definition and First Example of while loops

## Formal Syntax

while (<condition>)  
{  
 <statement block>  
}  
```~  
  
## 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!  
  
# Five Ways Things Can Go Wrong  
  
It is easy to write \_wrong\_ `loop` statements.   
Let us review some of the "classic" blunders.  
  
## Failing to update the variable occurring in the condition  
  
  
```~  
int number = 0;  
while (number <=5)  
{  
 C.WL("Hi Mom!);  
 C.WL(number);  
}  
```~  
  
Number isn't changed!  
  
## Updating the "wrong" value  
  
```~  
int number1, number = 0;  
while (number <=5)  
{  
 C.WL("Hi Mom!);  
 C.WL(number);  
 number1++;  
}  
```~  
  
## Having an empty body  
  
```~  
int number = 0;   
while (number <=5); // Note the semi-colon here!  
{  
 C.WL("Hi Mom!");  
 C.WL(number);  
 number++;  
}  
```~  
  
## Having an empty body (variation)  
  
```~  
int number = 0;  
while (number <=5)  
 C.WL("Hi Mom!);  
 C.WL(number);  
 number++;  
```~  
  
## 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.  
  
  
# 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());  
}  
```~  
  
# 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 iterations depends on the user.  
  
Count controlled loop  
  
: The number of iterations 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").  
  
   
# 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.  
  
# 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);  
```~  
  
# 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; } }

# Putting it all together!  
  
```{.cs .numberLines .lineAnchors}  
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, o) reason?");  
 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 payment?");  
 decimal down = decimal.Parse(Console.ReadLine());  
  
 Loan myLoan = new Loan(name, type, cscore, need, down);  
 Console.WriteLine(myLoan);  
 }  
  
}

# Arrays

## 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.

## 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 Program.Main()  
// "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 careful, 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];

## 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++;  
}

## 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., for int, etc.)!

# For Loops

## 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

## 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.

## 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());  
}

## 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();  
}

## 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.");  
}

## Iterations

There is another, close, structure 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 + " ");

Difference is w.r.t (with respect 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.

# 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.

## 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 writing are not (they are “non-static”, or “instance”).

| Static Method | Non-static Method |
| --- | --- |
| ClassName.MethodName(arguments) | ObjectName.MethodName(arguments) |
| Math.Pow(2, 5) () | 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 |

# 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));  
 }  
 }