7SENG011W
Object Oriented Programming

Computer Programs, Data Types and Variables, Selection Statements

Dr Francesco Tusa

Readings

- The topics we will discuss today can be found in the books
- Hands-On Object-Oriented Programming with C#
  - Chapter: Overview Of C# As A Language
- Programming C# 10.0
  - Chapter: Introducing C#
  - Chapter: Basic Coding In C#
- C# online documentation
  - Operators and Expressions
  - Selection statements

Outline

- Computer programs and .NET
- More on Types, Variables and Conversions
- Selection statements and blocks

Outline

- Computer programs and .NET
- More on Types, Variables and Conversions
- Selection statements and blocks

Computer Program

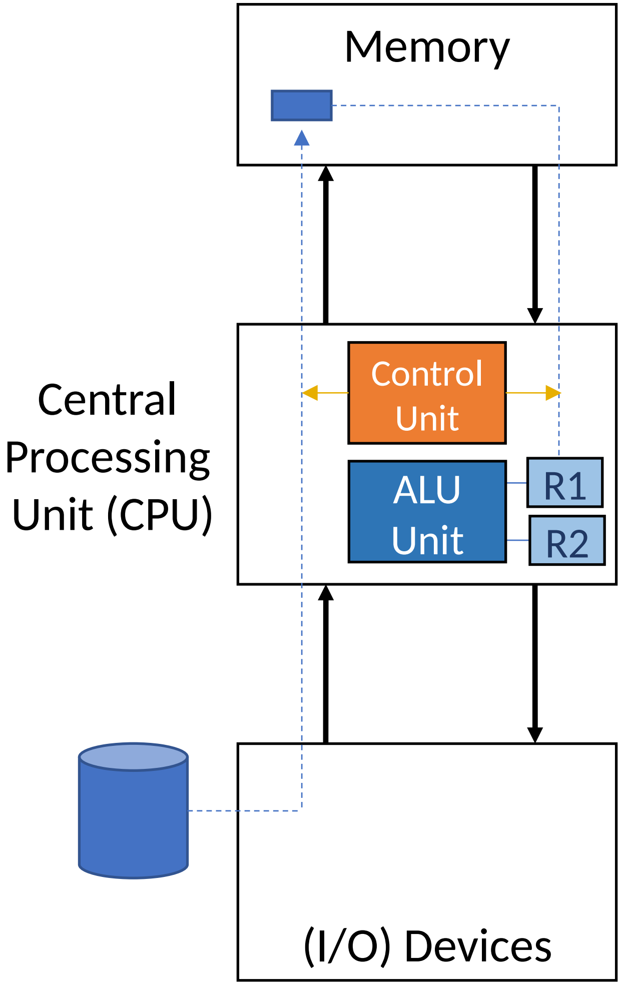
- Ordered set of **instructions** that tells a computer how to carry out a particular task or solve a problem

- *Programs* are *algorithms* written using particular “**programming languages**”

- Programs are translated and executed by the CPU according to one of the models described later

Ada Lovelace

The Computer architecture: Von Neumann



instruction: 011101001
0V 5V

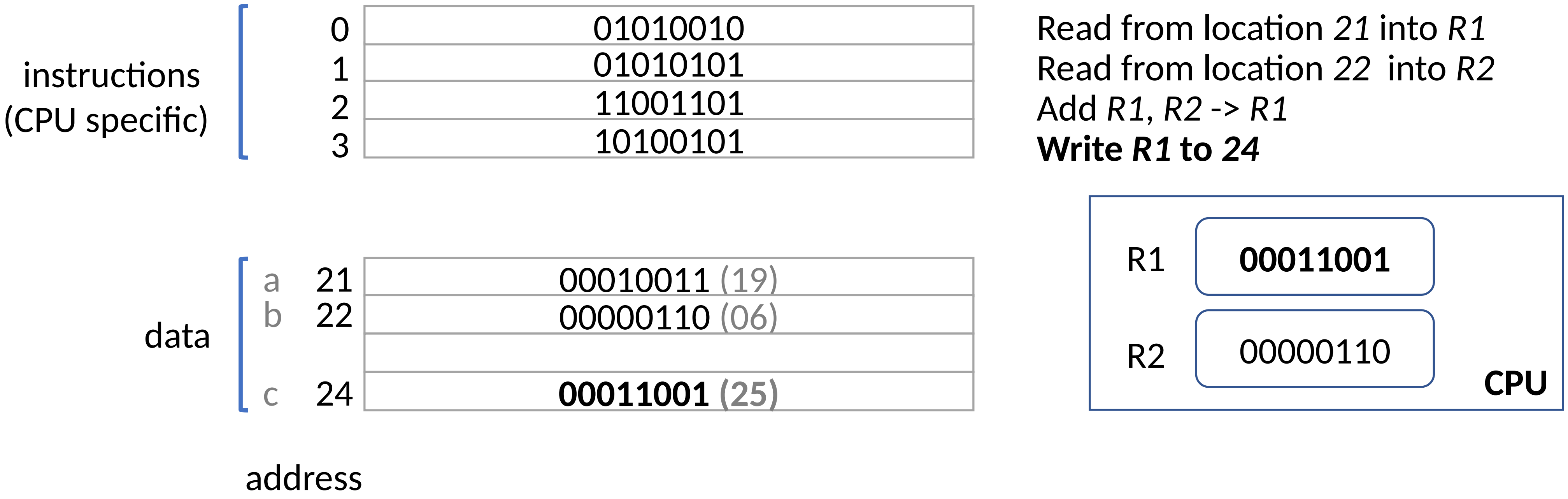
ALU: Arithmetic Logic Unit
R1, R2: Internal Registries

Last Week: integer math

- int variables store integer values
- The *int* type represents an integer: zero, positive or negative whole number
- The + symbol is the addition operations when applied to *int* operands

```
int a = 19;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

CPU Instructions



instructions
(CPU specific)

0	01010010
1	01010101
2	11001101
3	10100101

data

a	21	00010011 (19)
b	22	00000110 (06)
c	24	**00011001** (25)

address

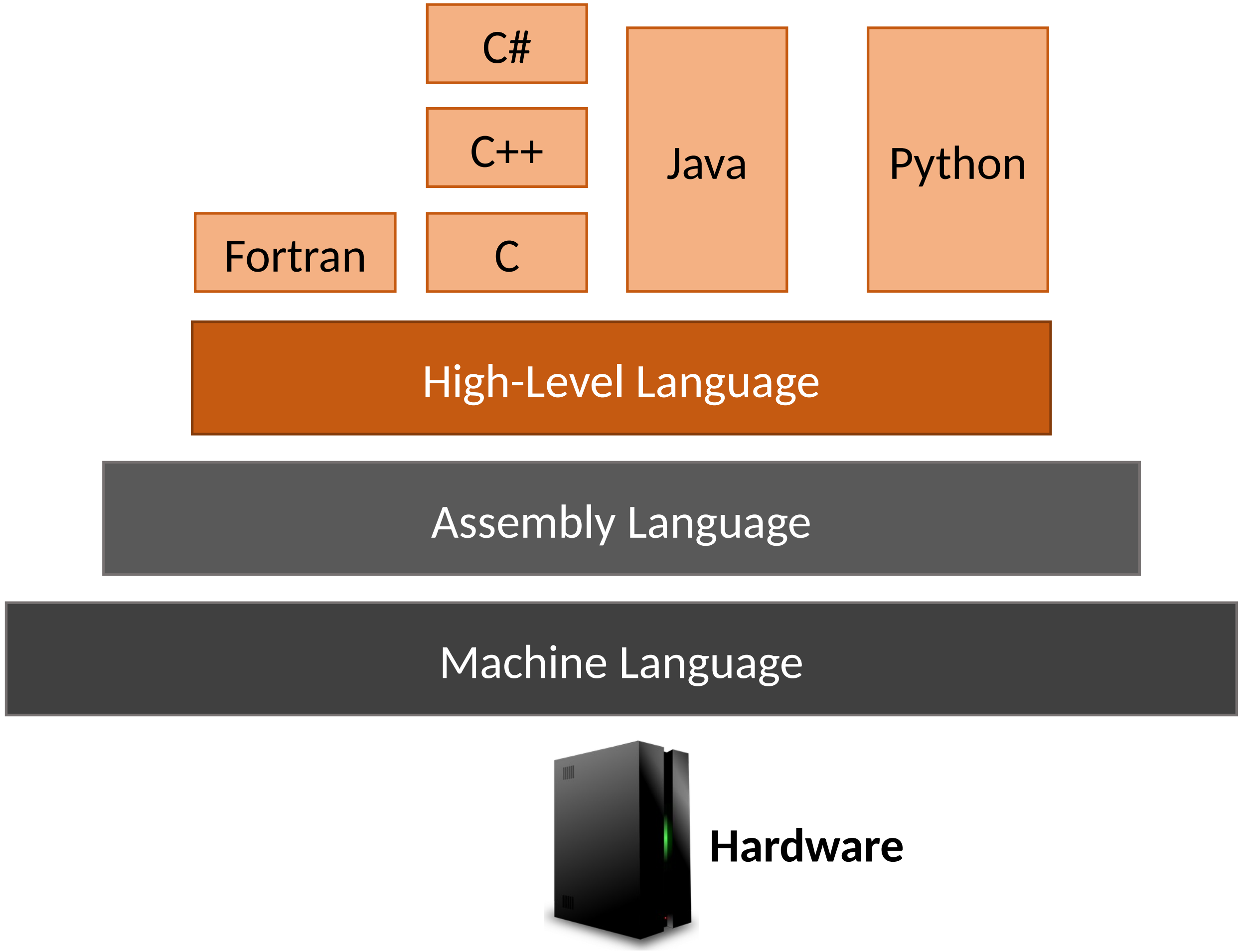
Read from location *21* into *R1*
Read from location *22* into *R2*
Add *R1*, *R2* -> *R1*
Write *R1* to *24*

R1	**00011001**
R2	00000110

CPU

Programming Languages

- Allow writing computers’ instructions in a way closer to natural languages
- Easier to understand for humans than CPU instructions
- High-level instructions need to be translated for the CPU to understand them



C#

C++

Java

Python

Fortran

C

High-Level Language

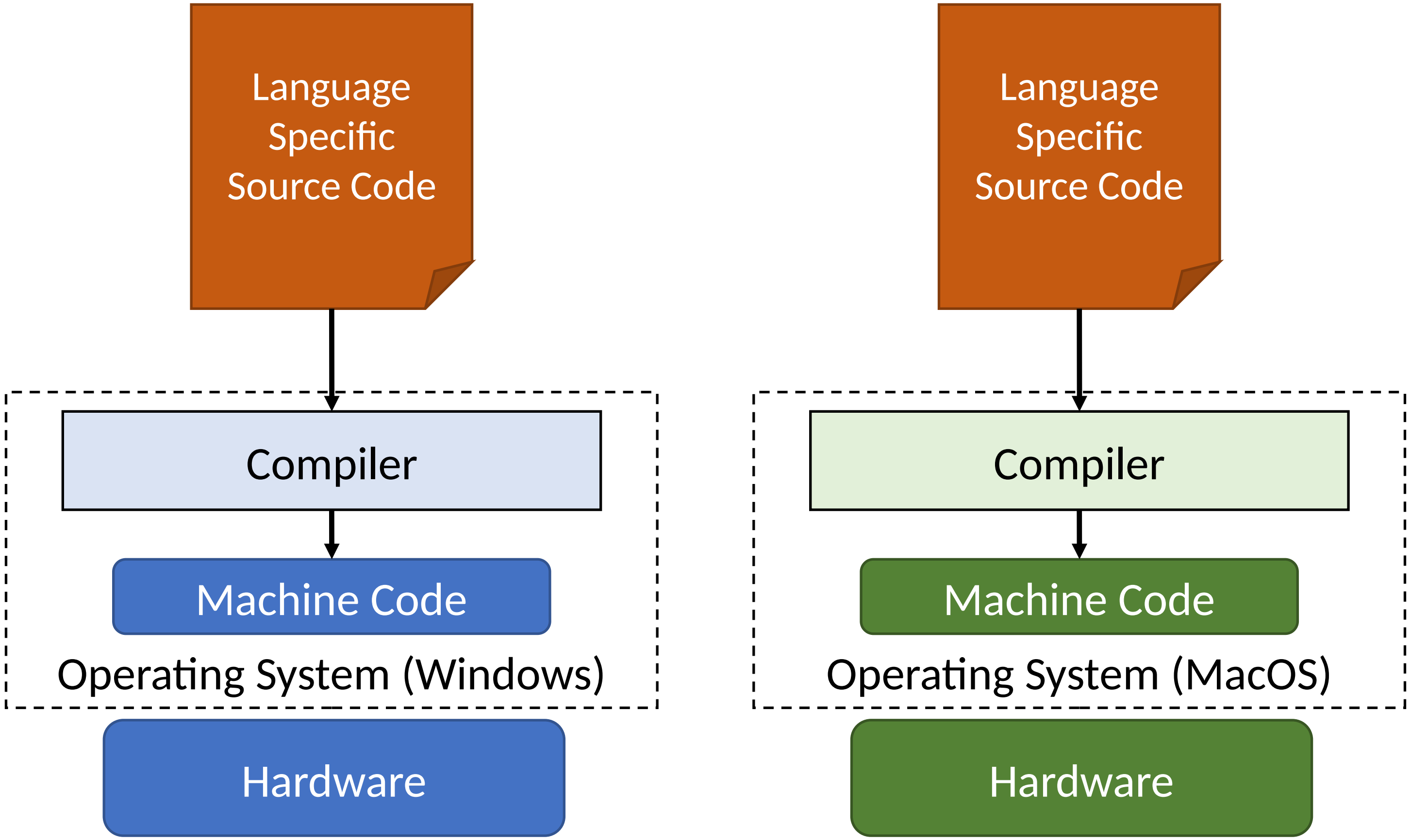
Assembly Language

Machine Language

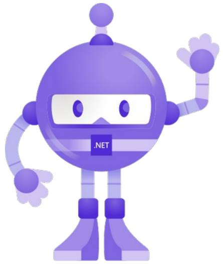
Hardware

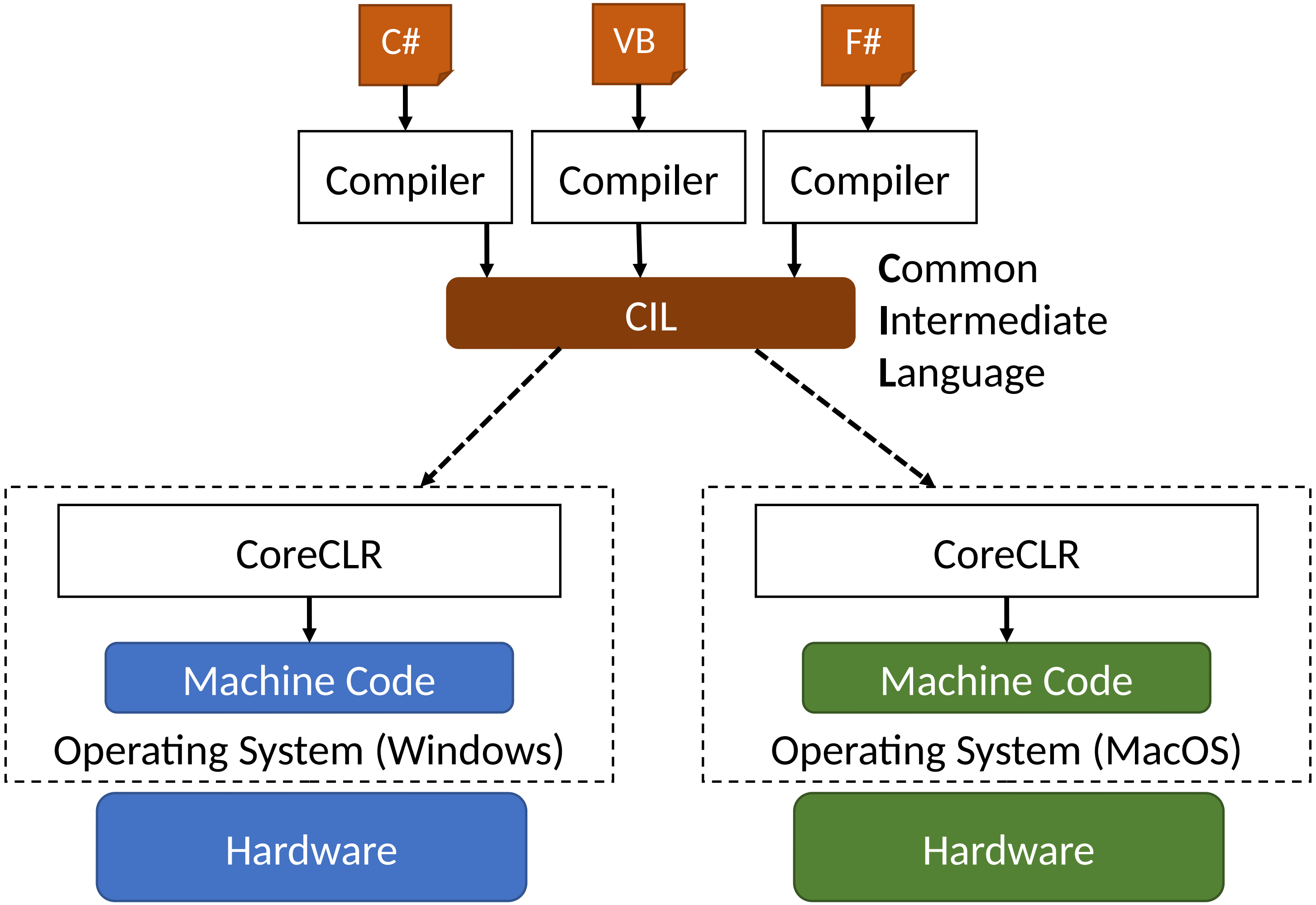
Native Applications

for instance, C
or C++ code


Language Specific Source Code
Compiler
Machine Code
Operating System (Windows)
Hardware
Language Specific Source Code
Compiler
Machine Code
Operating System (MacOS)
Hardware

Managed Code: .NET





Common **I**ntermediate **L**anguage

Common **L**anguage **R**untime
Application Virtual Machine
.NET

Outline

- Computer programs and .NET
- More on Types, Variables and Conversions
- Selection statements and blocks

C# Program Structure

```
class CLASSNAME
{
        static void Main(string[] args) // program entry-point
        {
        STATEMENTS
        }
}
```

Variables

Named ***memory location*** that stores a value of one ***type***

Form:

TYPE NAME;

Example:

```
string aFriend;
int a;
char c;
```

Types

Kinds of values that can be stored and manipulated – ***primitive types***

- **int**: Integer (0, 1, -47) 32 bits
- **double**: Real number (3.14, 1.0, -2.1) 64 bits floating-point (FP)
- **decimal**: Finance applications requiring higher precision 128 bits FP
- **char**: Single character ('a', 'd') 16 bits UTF-16
- **bool**: Truth value (**true** or **false**)

- **string**: Text ("hello", "James") – built with primitive types

Types

Kinds of values that can be stored and manipulated – ***non-primitive***

- **int**: Integer (0, 1, -47) 32 bits
- **double**: Real number (3.14, 1.0, -2.1) 64 bits floating-point (FP)
- **decimal**: Finance applications requiring higher precision 128 bits FP
- **char**: Single character ('a', 'd') 16 bits UTF-16
- **bool**: Truth value (**true** or **false**)

- **string**: Text ("hello", "James") – built with primitive types

Assignment

- Use **=** to give variables a value (*other variable or literal)*

  ```
  string aFriend;
  aFriend = "James"; // use "double quotes"
  ```

- Can be combined with a variable declaration

  ```
  double pi = 3.14;
  char c = 'c'; // use 'single quotes'
  bool isSeptember = true;
  ```

- *3.14*, *true*, *"James"*, 'c' are **literals** – they *literally* mean what they represent

Mismatched Types

- C# is a *statically typed* language
- Variables have a data type determined at *compile time*
- C# compiler verifies that types *always match*

```
string one = 1 // Error!
```

```
Error CS0029: Cannot implicitly convert type ‘int’ to ‘string’
```

Expression

- Instruction built by combining different **variables** and **literals** via **operators**
- *Assignment* is a simple expression (=)
- **Arithmetic expressions:**
  - can be written using *arithmetic operators*
  - produce a *single* value as *result* when evaluated

Operators

Symbols to build ***arithmetic expressions***

- Assignment: **=**
- Addition: **+**
- Subtraction: **-**
- Multiplication: *
- Division: /

- Remainder: **%**

```
int reminder = 10 % 3; // 1
```

Order of Operations

Follows standard math rules:

1. Parentheses ()
2. Multiplication and division
3. Addition and subtraction

```
double y = 3 / (2 + 1); // y = 1
double x = 3 / 2 + 1; // x = 2
int z = 10 % 3 * 2 + 1; // z = 1 * 2 + 1 = 3
```

Division

- Division (“/”) operates differently on *integers* and on *doubles*

```
int num = 10;
int den = 4;
int result = num / den;
// result = 2
```

```
double num = 10;
double den = 4;
double result = num / den;
// result = 2.5
```

Type conversions

```
int a = 2; // a = 2
double a = 2; // a = 2.0 (implicit conversion)

int a = 18.7; // ERROR
int a = (int) 18.7; // a = 18 (explicit cast conversion)

double a = 2/3; // a = 0.0 (implicit conversion)
```

Type conversions

- C# *promotes* values of types with a *narrower* range into the *larger* one before performing the calculations
- For example, double can represent any value that int can, and many that it cannot, so double is the more expressive type
- Promotion: *automatic* conversion

```
double a = 2; // a = 2.0 (implicit conversion)
```

- Narrowing: *cast* conversion

```
int a = (int) 18.7; // a = 18 (explicit cast conversion)
```

More on Conversion by Casting

- Division (“/”) operates differently on integers and on doubles

```
int num = 10;
int den = 4;
int result = num / den;
```

```
double num = 10;
double den = 4;
double result = num / den;
```

```
int num = 10;
int den = 4;
double result = (double) num / den;
```

Outline

- Computer programs and .NET
- More on Types, Variables and Conversions
- Selection statements and blocks

Blocks

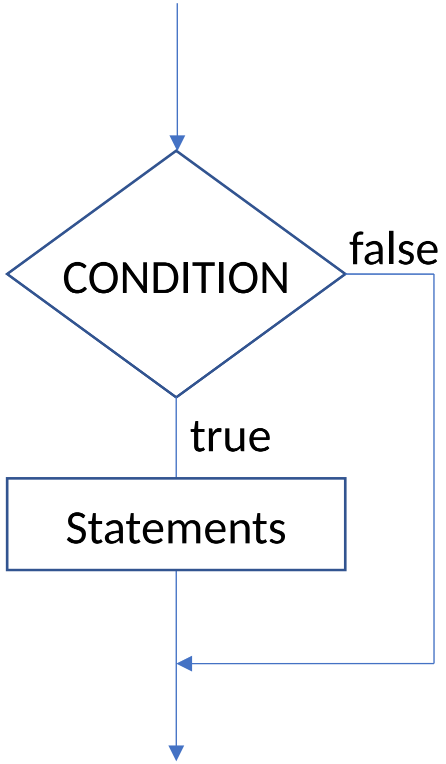
- A block is a region of code delimited by a pair of braces *{}*
- Program *Main* entry-point
- Instructions within an *if* or *else* statement
- Blocks can be *nested*

Selection statements: **if**

```
if (CONDITION) {
        STATEMENTS
}
```

block

- ***CONDITION*** is a *logical expression*
- It combines *relational* and *logical operators*
- Result is either *true* or *false* when evaluated



Relational operators

- Can be applied to compare variables, literals and create ***relational expressions*** whose result is either **true** or **false**

x > y: x is greater than y
x < y: x is smaller than y
x >= y: x is greater than or equal to y

x <= y: x is smaller than or equal to y

x == y: x equals y (equality: ==, assignment: =)

x != y: x not equal y

Selection statements: **if**

```
class Example
{
        static void Main(string[] args)
        {
        // values are assigned to x and y
        int x = 10;
        int y = 5;

        if (x > y)
        {
        Console.WriteLine("x is greater than y");
        }
        }
}
```

Logical operators

- Allow the definition of **logical expressions**

- **Binary logical operators** *(AND, OR)* combine:
  - two *relational expressions*
  - *a relational expression* with a *bool variable* or *literal*
- **Unary logical operators** *(NOT)* applied to *single* operand

Logical operators

&&: logical AND

true if both operands are true, *false* otherwise

```
int x = 3;
int y = 4
if (x > 0 && y < 5) {
        // do something useful
}
```

Logical operators

&&: logical AND

true if both operands are true, *false* otherwise

```
int x = 3;
int y = 4
if (x > 0 && y < 5) {
        // do something useful
}
```

both expressions are true, the AND is true

Logical operators

||: logical OR

true if at least one of the operands is true, false otherwise

```
int x = 3;
int y = 6
if (x > 0 || y < 5) {
        // do something useful
}
```

Logical operators

||: logical OR

true if at least one of the operands is true, false otherwise

```
int x = 3;
int y = 6
if (x > 0 || y < 5) {
        // do something useful
}
```

x>0 is true, y<5 is false; the OR is true

Logical operators

!: logical NOT

- *unary* operator that *changes* the value of its operand
- if the operand is *true*, the result is *false*; if the operand is *false*, the result is *true*

```
int x = 3;
int y = 4
if ( x > 0 && !(y < 5) ) {
        // do something useful
}
```

Logical operators

!: logical NOT

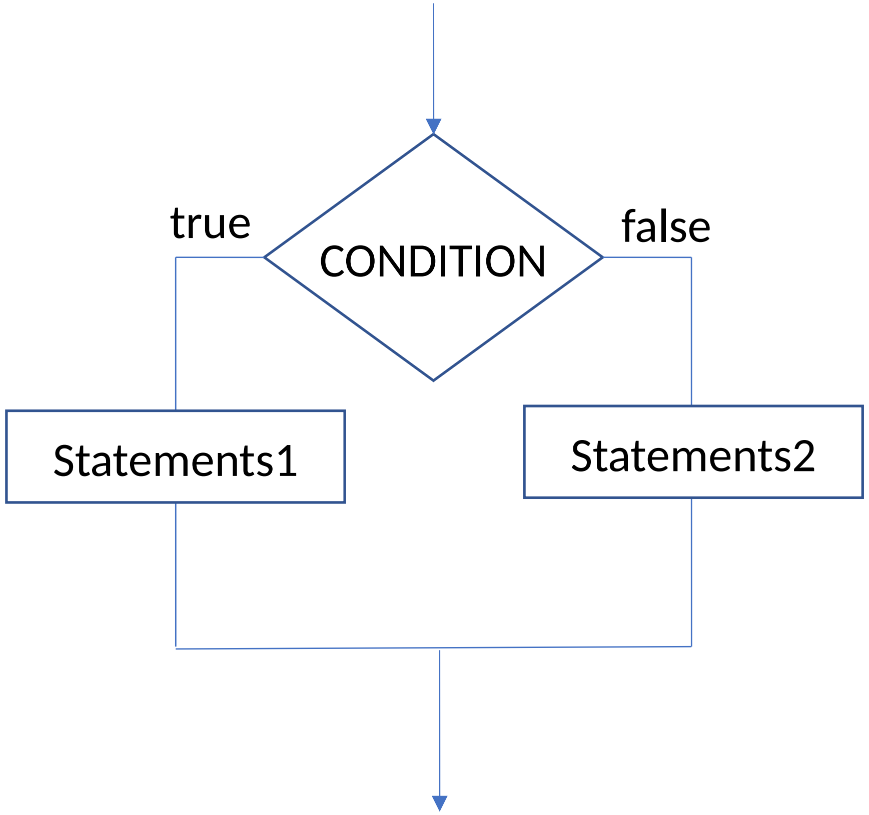
- *unary* operator that *changes* the value of its operand
- if the operand is *true*, the result is *false*; if the operand is *false*, the result is *true*

```
int x = 3;
int y = 4
if ( x > 0 && !(y < 5) ) {
        // do something useful
}
```

y < 5 is *true* but when the **!** is applied the result is *false*
the whole expression is false because of the &&

Selection statements: **if-else**

```
if (CONDITION) {
        STATEMENTS1
} else {
        STATEMENTS2
}
```



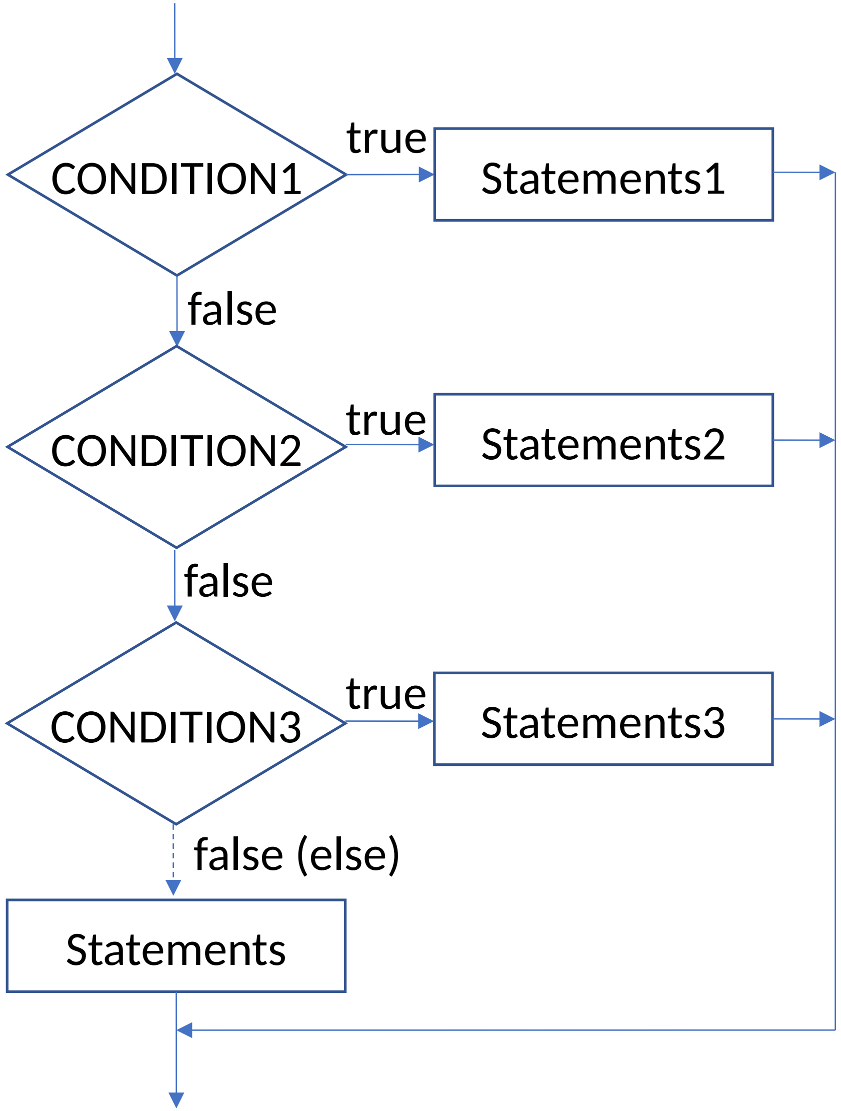
Selection statements: **if-else**

```
class Example2
{
        static void Main(string[] args)
        {
        int temperature = ... // value is assigned to temperature

        if (temperature > 30)
        {
        Console.WriteLine("Weather is hot");
        } else {
        Console.WriteLine("Weather is not hot");
        }
        }
}
```

Selection statements: **if-else-if**

```
if (CONDITION1) {
        STATEMENTS1
} else if (CONDITION2) {
        STATEMENTS2
} else if (CONDITION3) {
        STATEMENTS3
} else {
        STATEMENTS
}
```



Selection statements: if-else-if

```
class Example3
{
        static void Main(string[] args)
        {
                int temperature = … // value is assigned to temperature

                if (temperature >= 30)
                {
                  Console.WriteLine("Weather is hot");
                } else if (temperature > 20) {
                  Console.WriteLine("Weather is warm");
                } else {
                  Console.WriteLine("Weather is cold");
                }
        }
}
```

Blocks

- A block is a region of code delimited by a pair of braces *{}*
- Program *Main* entry-point
- Instructions within an *if* or *else* statement
- Blocks can be *nested*

Nested blocks

```
class Example
{
    static void Main(string[] args)
    { // beginning of Main block

        int x = 10, y = 5; // values are assigned to x and y
        int sum = x + y;

        if (sum < 20)
        { // beginning of nested block
            Console.WriteLine(sum + " is less than 20");
        } // end of nested block
    } // end of Main block
}
```

Nested blocks

```
class Example
{
        static void Main(string[] args)
        { // beginning of Main block

                int x = 10, y = 5; // values are assigned to x and y
                int sum = x + y;

                if (sum < 20)
                { // beginning of nested block
                        Console.WriteLine(sum + " is less than 20");
                } // end of nested block
        } // end of Main block
}
```

The code inside this block can access the variable (*sum*) declared in the parent block

Nested blocks: variable not in scope

```
class Example
{
    static void Main(string[] args)
    { // beginning of Main block

        int x = 10, y = 5; // values are assigned to x and y
        int sum = x + y;

        if (sum < 20)
        { // beginning of nested block
            Console.WriteLine(sum + “ is less than 20”);
            int willNotWork = sum * 5;
        } // end of nested block
        Console.WriteLine(“ willNotWork is ” + willNotWork);
    } // end of Main block
}
```

willNotWork only exists **within** the if block

Code indentation style

- It is important that the instructions belonging to a block are properly *indented*
- This improves the *readability* of the code
- Visual Studio already helps with code indentation
- **Task**: look for “indentation style” on the web and select the one you prefer
- Use it consistently in your code

Code indentation style

```
class Example2
{
static void Main(string[] args)
{
int num = 10;
int den = 4;
double result = (double) num / den;

if (result > 0)
{
Console.WriteLine(“Result is > 0”);
} else if (result == 0) {
Console.WriteLine(“Result is 0”);
} else {
// what should we print here?
}
}
}
```

Please NEVER do this!

Exercise (homework)

- A sensor collects temperature measurements *T* (in Celsius)
- Using an appropriate selection statement, write a program that prints different messages on the screen:
  - Normal: T <= 24 C
  - Warning: 24 C < T <= 30 C
  - Critical: T > 30 C
- *T* should be typed in from the keyboard
  - Hint: use Convert.ToInt32() and Console.ReadLine()