# Object Oriented Programming with Applications
# Lecture 2

Witold Gawlikowicz & David Šiška

School of Mathematics, University of Edinburgh

2018-19[1]

[1] Last updated 19th September 2018

# Lecture 2

- Basic data types & memory & operations
- Flow control: `if`, `else`, `for`, `while`
- Methods a.k.a. functions
- Arrays
- Scope of a variable
- Compiler is your friend

Read:
Wright, P. - Beginning Visual C# 2005 Express Edition. Chapter 3.
Duffy, D. J. and Germani, A. - C# for Financial Markets. Chapter 2, Sections 1–8.

# Basic data types & memory

| Type | Contains | Size |
|---|---|---|
| **bool** | `True`,`False` | 1 bit |
| `byte` | $0, 1, \ldots, 255$ | 8 bits |
| `sbyte` | $-128, -127, \ldots, 0, 1, \ldots, 127$ | 8 bits |
| `short` | $-2^{15}, -2^{15}+1, \ldots, 0, 1, \ldots 2^{15}-1$ | 16 bits |
| `ushort` | $0, 1, \ldots 2^{16}-1$ | 16 bits |
| **int** | $-2^{31}, -2^{31}+1, \ldots, 0, 1, \ldots, 2^{31}-1$ | 32 bits |
| `uint` | $0, 1, \ldots, 2^{32}-1$ | 32 bits |
| `s long` | $-2^{63}, -2^{63}+1, \ldots, 0, 1, \ldots, 2^{63}-1$ | 64 bits |
| `ulong` | $0, 1, \ldots, 2^{64}-1$ | 64 bits |
| `float` | IEEE 754[2] approximation of $\mathbb{R}$ | 32 bits |
| **double** | IEEE 754 approximation of $\mathbb{R}$ ("$2\times$ better") | 64 bits |
| `char` | Unicode characters | 16 bits |
| **string** | Collection of unicode characters | 16 bits/char |

[2] Technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE)

# Basic arithmetic operators

Assume we have:

```
int m = -6; int n = 4; int k;
float x = -6; float y = 4; float z
```

| | Meaning | Usage | Comment |
|---|---|---|---|
| * | Multiplication | `k = m*n; z = x*y;` | |
| / | Division | `k = m/n; z = x/y;` | $k = -1$ but $z = -1.5$ |
| % | Remainder | `k = m%n` | $k = -2$ |
| + | Addition | `k = m+n; z = x+y;` | |
| - | Subtraction | `k = m-n; z = x-y;` | |

Order of precedence (highest to lowest): `* / % + -`.

Use `( )` to enforce desired order of computation.

# Prefix and postfix increment & decrement

Prefix and postfix increment `++` both increase the variable they are applied to by 1 but return different values!

Assume we have:
`int m = -6; int n = 4; int k; int l;`
Prefix / postfix increment:
`k = ++m; l = n++;`
Results in $k = -5$, $m = -5$, $l = 4$, $n = 5$.

Prefix and postfix decrement `--` both decrease the variable they are applied to by 1 but return different values!

# Comparison

| | Meaning | Usage | Comment |
|---|---|---|---|
| `<` | Strictly less than | `x < y` | Returns `true` or `false` |
| `>` | Strictly greater than | `x > y` | Returns `true` or `false` |
| `<=` | Less than or equal | `x <= y` | Returns `true` or `false` |
| `>=` | Greater than or equal | `x >= y` | Returns `true` or `false` |
| `==` | Equals | `x == y` | Returns `true` or `false` |
| `!=` | Not equal | `x != y` | Returns `true` or `false` |

# Basic logical operators on Booleans

Assume we have: `bool p = true; bool q = false; int m = 3; int n = 2;`

| | Meaning | Usage | Comment |
|---|---|---|---|
| `&` | Integer bitwise and, logical and | `p & q; m & n;` | Eager |
| `^` | Integer bitwise XOR, logical XOR | `p ^ q; m ^ n;` | Not $p^q$! |
| `\|` | Integer bitwise or, logical or | `p \| q; m \| n;` | Eager |
| `&&` | Conditional and | `p && q;` | Lazy |
| `\|\|` | Conditional or | `p \|\| q;` | Lazy |

Logically `p & q` and `p && q` are equivalent.

But not computationally. Consider `function1()` that always returns `false`. Consider `function2()` that always returns `true` but also creates a file on the hard disc.
`function1() && function2()` is false and nothing gets written hard disc. `function1() & function2()` is false but a file gets created.

# Aside - floating point representation

Floating point number:

$$\mathbb{Q} \ni x = (-1)^s \cdot c \cdot b^q,$$

where: $s$ is the sign (either 0 or 1, $b$ is the base (either 2 or 10), $c$ is the "significand" and $q$ is an exponent.

Both $c$ and $q$ natural numbers. Their size depends on whether we use `float` or `double`.

e.g. $1.2345 = (-1)^0 \cdot 12345 \cdot 10^{-4}$.

| Type | Sign bits | Significand bits | Exponent bits | Total bits |
|---|---|---|---|---|
| `float` | 1 | 23 | 8 | 32 |
| `double` | 1 | 52 | 11 | 64 |

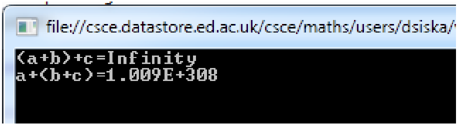
# Aside - the dangers of floating point arithmetic

```
double a = Math.Pow(10,308);
double b = 1.01*Math.Pow(10,308);
double c = -1.001*Math.Pow(10,308);

double s1 = a + b;
s1 = s1 + c;

double s2 = a + (b + c);

Console.WriteLine("(a+b)+c={0}", s1);
Console.WriteLine("a+(b+c)={0}", s2);
```




$$
\begin{aligned}
(a+b)+c = a+(b+c) = a+b+c \\
= 1\cdot 10^{308} + 1.01\cdot 10^{308} - 1.001\cdot 10^{308} = 1.009\cdot 10^{308}.
\end{aligned}
$$

# Aside - the dangers of floating point arithmetic

Lesson learned:

- Floating point arithmetic $\neq$ arithmetic using $\mathbb{R}$.
- Roundoff errors can be managed:
  - Avoid using numbers so big or so small that you are at the edge of floating point representation (e.g. $10^{308}$ is not good even with `double`).
  - Do not add lot of very small to a very big number.
  - Use stable numerical methods.
  - Use numerical *libraries* or established algorithms. *Do not re-invent the wheel.*

# Flow control - if, else if, else

```
if (boolean)
    k = 1;
else if (boolean)
    k = 2;
else
    k = 3;
```

```
if (boolean) {
    k = 1;
    l = -1;
}
else if (boolean) {
    k = 2;
    l = -2;
}
else {
    k = 3;
    l = -3;
}
```

# Flow control - switch

The `switch` statement is a control statement that selects a switch section to execute from a list of candidates.

```
int caseSwitch = 1;
switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1");
        break;
    case 2:
        Console.WriteLine("Case 2");
        break;
    default:
        Console.WriteLine("Default case");
        break;
}
```

# Flow control - for loop

```
for (initializer; condition; iterator) {
    body
}
```

- `initializer` Do something once at the beginning of the loop.
- `condition` Check whether condition is satisfied. Execute "body" and "iterator" if satisfied. Finish if not.
- `iterator` Do something at the end of each iteration.

```
int N = 100;
for (int i = 0; i < N; i++) {
    Console.WriteLine(i);
}
```

# Flow control - while loop

```
while (condition) {
    body
}
```

- `condition` Check whether condition is satisfied. Execute body if it is, stop if it is not.

```
int N = 100;
int i = 0;
while(i < N) {
    Console.WriteLine(i);
    i++;
}
```

# Flow control - do ... while loop

Like while loop but condition checked at the end. So body gets executed at least once.

```
do {
    body
} while (condition);
```

- `condition` Check whether condition is satisfied. Execute body again if it is, stop if it is not.

```
int N = 100;
int i = 0;
do {
    Console.WriteLine(i);
    i++;
} while(i < N);
```

# Flow control - break

Use `break` to break a loop if something special happens.

```
for (int i = 1; i <= 100; i++) {
    if (i == 5) {
        break;
    }
    Console.WriteLine(i);
}
```

# Flow control - continue

Use `continue` to move to the next iteration of `while`, `do` or `for`.

```
for (int i = 1; i <= 10; i++) {
    if (i < 9) {
         continue;
    }
    Console.WriteLine(i);
}

/*
Output:
9
10
*/
```

# Methods

A *method* is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments.

In mathematics the nearest concept is that of a *function*. But the two are not equivalent.

Example

```
static ulong Factorial(ulong n)
{
    if (n==1) return 1;
    else return n*Factorial(n-1);
}

static void Main(string[] args)
{
    Console.WriteLine(Factorial(6));
}

// Output is 720;
```

# Methods - more details

- Use `static` if the method does not change any of the "instance variables of the class"[3].

```
static string Concatenate(string s1, string s2)
{
    return s1 + s2;
}

static void Main(string[] args)
{
    Console.WriteLine(Concatenate("C# ", "is easy!"));
}

// Output is: C# is easy!
```

[3] We'll get back to this later.

# Methods - more details

- Use `void` as a return type if the method does not return a value.
- Use `ref` to force the argument to be passed by reference[4].

```
static void Concatenate(string s1, string s2, ref string output )
{
    output = s1 + s2;
}

static void Main(string[] args)
{
    string output = "";
    Concatenate("I thought C# ", "was easy!", ref output);
    Console.WriteLine(output);
}

// Output is: I thought C# was easy!"
```

[4] We'll get back to this later.

# Arrays

All basic types (and strings) discussed so far can be arranged into arrays.

```
int N = 5;
int[] myArray = new int [N];
for (int i = 0; i<N; i++)
    myArray[i] = i*i;

for (int j = 0; j<N; j++)
    Console.WriteLine(myArray[j]);

/* Output is
0
1
4
9
16
*/
```

# Array example

Get an array of characters from a string

```
static void StringToCharArray(String s, ref char[] charArray, ref int size)
{
    size = s.Length;
    charArray = new char[size];
    for (int i = 0; i < size; i++)
        charArray[i] = s[i];
}

static void Main(string[] args)
{
    string myString = "Arrays are great!";
    char[] myStringAsCharArray = new char[0];
    int size = 0;
    StringToCharArray(myString, ref myStringAsCharArray, ref size);
    for (int i = 0; i < size; i++)
        Console.Write(myStringAsCharArray[i]);
}
```

# Array example - sort

```
static void MySort(double[] numbers)
{
    bool swapped;
    do {
        swapped = false;
        for (int i = 0; i < numbers.Length - 1; i++) {
            if (numbers[i] > numbers[i + 1]) {
                double tmp = numbers[i];
                numbers[i] = numbers[i + 1];
                numbers[i + 1] = tmp;
                swapped = true;
            }
        }
    }
    while (swapped);
}

static void Main(string[] args)
{
    double[] myNumberList = new double[] {1.0,-3.0,-2.0,15.0,12.1,4.2};
    MySort(myNumberList);
}
```

# Scope / context of a variable

A variable is only available inside the block it is declared in and its sub-blocks.
Example:

```
static void Main(string[] args)
{
    int i = 0;
    int N = 10;
    while (i < N) { Console.WriteLine(i); i++; }

    int i = 0;
    while (i < 10) { N = N + i; i++; }
}
```

This will fail to compile as `i` is already declared once in this scope.

# Scope / context of a variable

Example:

```
static void Main(string[] args)
{
    int N = 10; int sum = 0;
    for (int i = 0; i < N; i++)
    {
        sum = sum + i;
        bool isEven = false;
        if (sum \% 2 == 0) isEven = true;
    }
    Console.WriteLine(isEven);
}
```

This will fail to compile as `isEven` only exists in the scope of the for loop.

# Scope / context of a variable

Easy fix:

```
static void Main(string[] args)
{
    int N = 10; int sum = 0;
    bool isEven = false;
    for (int i = 0; i < N; i++)
    {
        sum = sum + i;
        if (sum \% 2 == 0) isEven = true;
    }
    Console.WriteLine(isEven);
}
```

# Scope / context of a variable

Example:

```
using System;

class MyExample
{
    static double AreaOfCircle(double r) { return myPi*r*r; }

    static void Main(string[] args)
    {
        const double myPi = 3.14;
        Console.WriteLine(AreaOfCircle(1));
    }
}
```

This will fail to compile as `myPi` is not declared in the scope of the function `AreaOfCircle`.

# Scope of a variable

```
using System;

class MyExample
{
    const double myPi = 3.14;
    static double AreaOfCircle(double r) { return myPi*r*r; }

    static void Main(string[] args)
    {
        Console.WriteLine(AreaOfCircle(1));
    }
}
```

This is fine as `myPi` is now a member of the class `MyExample`[5]
Hence it is available to all the methods in the same class, in particular the method `AreaOfCircle`.

[5] We'll get back to this!

# Compiler is your (and my) friend!

Modern "strongly typed" programming languages and their compilers are designed to stop us from making stupid mistakes:

- Scope / context: prevents you using a different variable than you think you are.
- Only allowing loss-less implicit conversion: compiler will let you do

  ```
  uint unsigned_k = 10;
  int k = signed_k;
  ```

  compiler won't let you do:

  ```
  int k = 10;
  uint unsigned_k = k; // because you might loose the sign.
  ```

- If you declare a variable to be `const` it won't let you change it.
- Many more as you will find out.

# Summary

We have discussed:

- Basic data types
- Basic operators
- Floating point numbers
- Basic flow control
- Methods
- Arrays
- Scope of variable

# Next week...Lecture 3

Object-Oriented Programming concepts

- Objects, types, classes, methods, properties
- Encapsulation
- Inheritence
- Static methods and classes

Related reading:
Wright, P. - Beginning Visual C# 2005 Express Edition. Chapter 4 and 6.
Duffy, D. J. and Germani, A. - C# for Financial Markets, Chapter 3, and Chapter 4 Sections 4.1–4.5.

# Next week...Lecture 4

- Exceptions
- Data structures
- Basic algorithm complexity
- What makes a good code?

Related reading:
Wright, P. - Beginning Visual C# 2005 Express Edition. Chapter 5.
Duffy, D. J. and Germani, A. - C# for Financial Markets, Chapter 3, and Chapter 4 Sections 4.1–4.5.