TP C#0: There and Back Again

Assignment

Archive

You must submit a zip file with the following architecture :

```
rendu-tp0-firstname.lastname.zip
|-- firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- TP0/
        |-- TP0.sln
        |-- TP0/
            |-- Everything except bin/ and obj/
```

- You shall obviously replace *firstname.lastname* with your login (the format of which usually is *firstname.lastname*).
- The code **MUST** compile.
- In this practical, you are allowed to implement methods other than those specified. They will be considered bonuses. However, you must keep the archive's size as small as possible.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline.
Here is an example (where `$` represents the newline and ␣ a blank space):

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with **NO** extension. To create your `AUTHORS` file simply, you can type the following command in a terminal:

```
echo "* firstname.lastname" > AUTHORS
```

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

1.1 Objectives

During this practical, you will discover and learn to use a (relatively) new language, supported by one of the largest conglomerates on earth: Microsoft's **C#** . You will be working on a new development environment, JetBrains' own **Rider** (farewell Emacs!).

1.2 The C# language

The C# language (pronounced "see sharp") is an object-oriented programming language sponsored and developed by Microsoft.

One of many languages derived from C and C++, C# is also inspired in parts by Java, from which it borrows quite a few concepts and ideas.

Originally a Windows-only delicacy, the C# language can nowadays target Linux and macOS systems as well. This portability is in no small measure thanks to the Mono initiative. Mono is an open source implementation of the .NET Framework.

If you don't know about any of the languages or terms we have introduced in the previous few paragraphs, we invite you to read their Wikipedia page. You will eventually encounter each and every one of them during your first three years at EPITA, so it's best to get accustomed now.

As for C# , it is assuredly **very** different from the OCaml you've come to know and love. For better or worse. In any case, this document serves as an overview of the primitives of the language. We strongly recommend you check out the official documentation **mdsn.com** for more details about the specifics of the language, as well as for an exhaustive list of the available libraries and functions.

1.3 Rider

JetBrain's Rider is a cross-platform IDE for the C# language. If you've ever written some code in a mainstream language before, you might have come across another of JetBrain's IDEs: IntelliJ IDEA for Java, PyCharm for Python, WebStorm for JavaScript, *etc.*

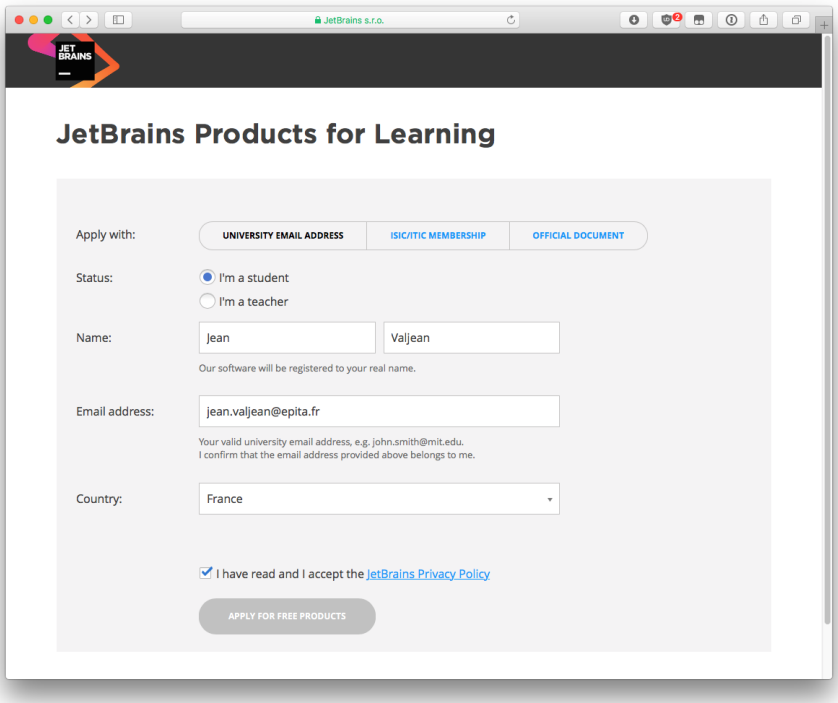
Like all modern IDEs, Rider features an intelligent auto-completion engine, a powerful debugger, as well as a plethora of utilities of questionable value. We certainly won't visit them all during the practicals, but we recommend you explore Rider's capabilities on your own. Getting a good grasp on some of its features will save you a lot of development time.

2 Riding the wave

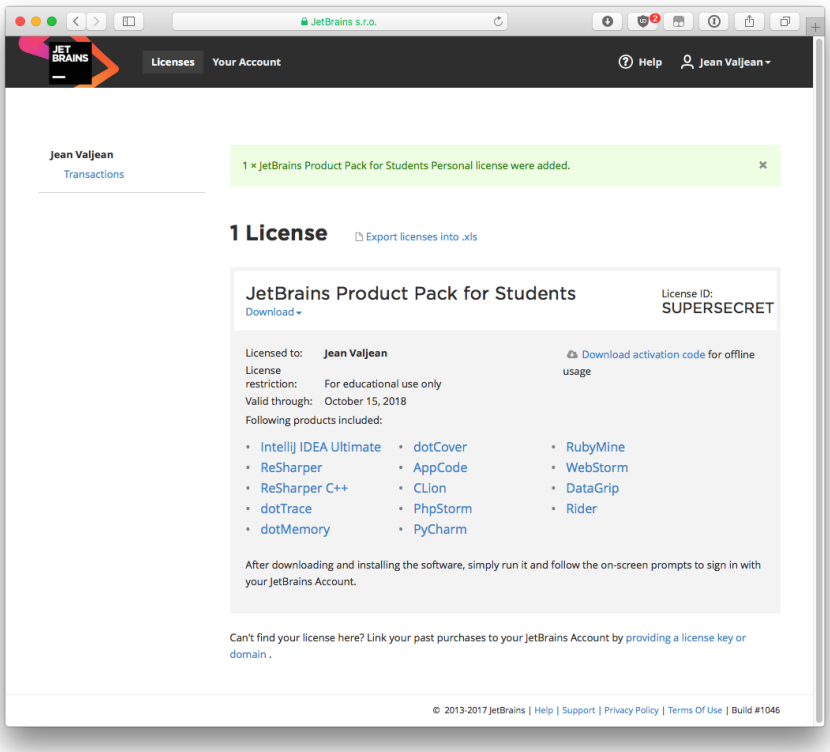
2.1 The Rider License

Turns out Rider is paid software, so you'll need a license in order to use it for a prolonged period of time. Luckily, as a student, you get a free pass.

Go to the URL https://www.jetbrains.com/shop/eform/students and enter your first name, last name and **EPITA email address**, then click "Apply". The form should look something like this, with your own info substituted in:

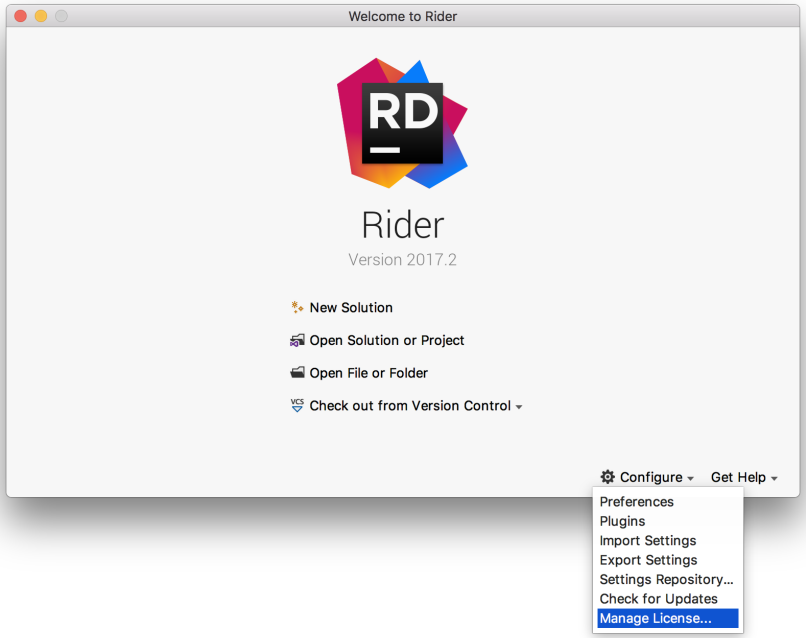


After you submit this form, you'll receive a mail asking you to confirm your License Request. Finally, once you do confirm by clicking on the link embedded in the email, you will receive another email. This time, it will link you to an account creation form with your email address filled in. Pick a secure password and create your account. Now you will be presented with the following screen:

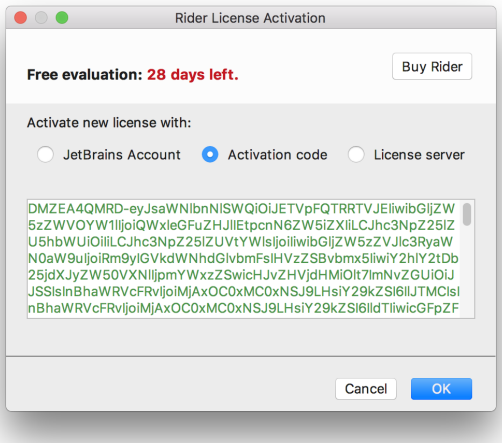


Clicking on the "Download activation code" link will download a plaintext file. Open it and copy its contents to your clipboard.

Now, launch Rider as you would a normal application on your system. If you do not get prompted for a license directly, on the welcoming screen, click on "Configure" and select "Manage License...".



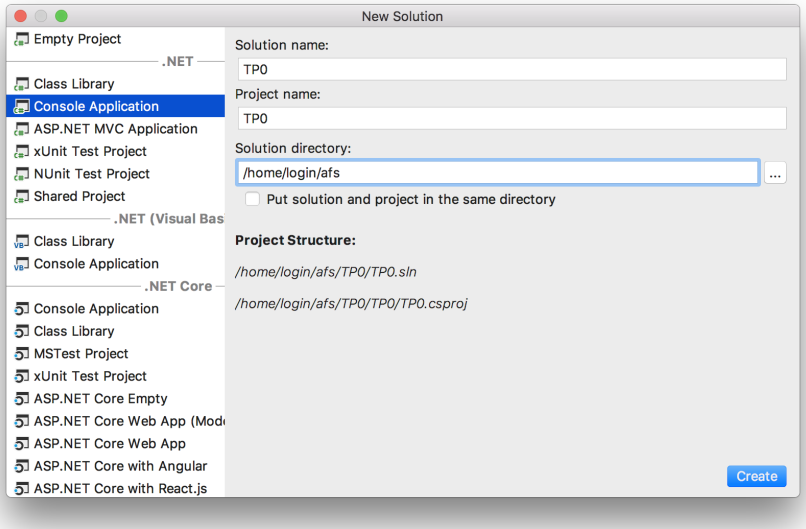
There, you will have an option to paste an activation code, which coincidentally corresponds to exactly what you just have copied to your clipboard. It will be filled in automatically for you.



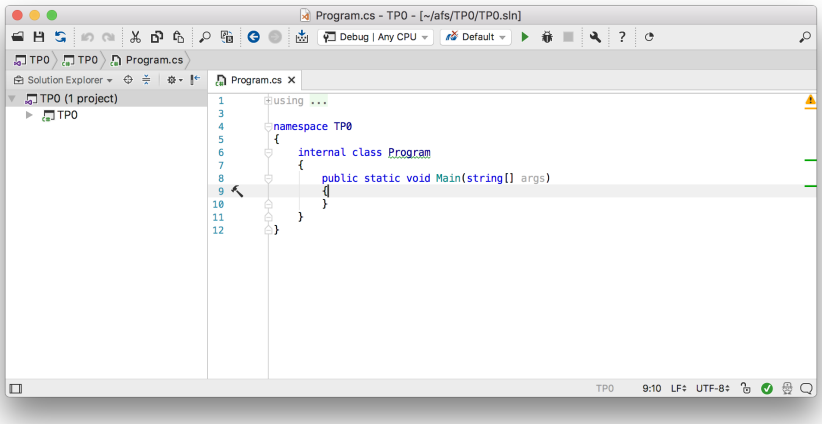
Congrats, that makes one good thing done today! On to the practical now...

2.2 Creating a new solution

On the "Welcome to Rider" screen, click on the "New Solution" link and choose the following settings:



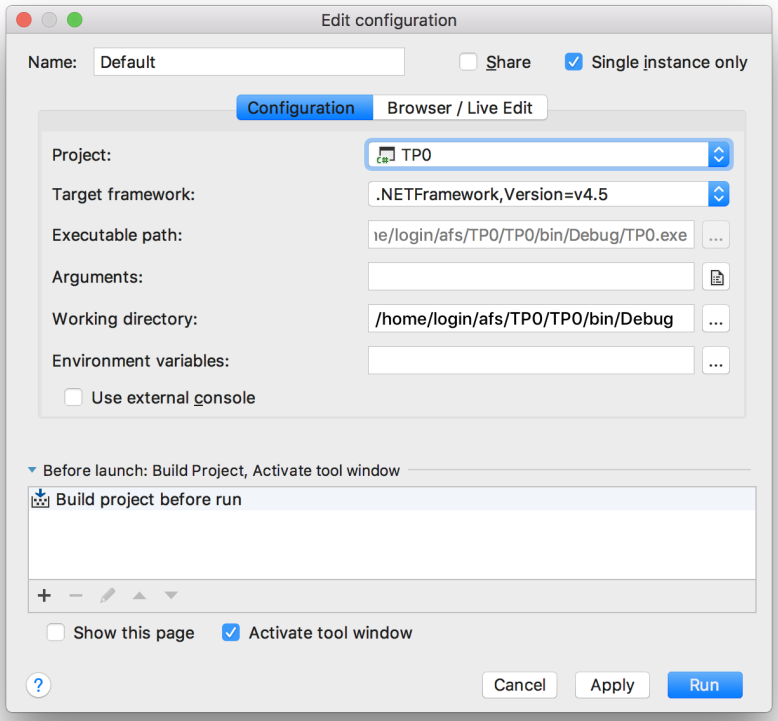
Replace "login" with your own login. Now click "Create". You will be greeted by this screen:



On the left, you will find your solution's hierarchy. Clicking on the little arrows will expand the hierarchy and show the directories and files inside of your project. For now, the only file we care about is "Program.cs": this is where you'll write your code for this practical.

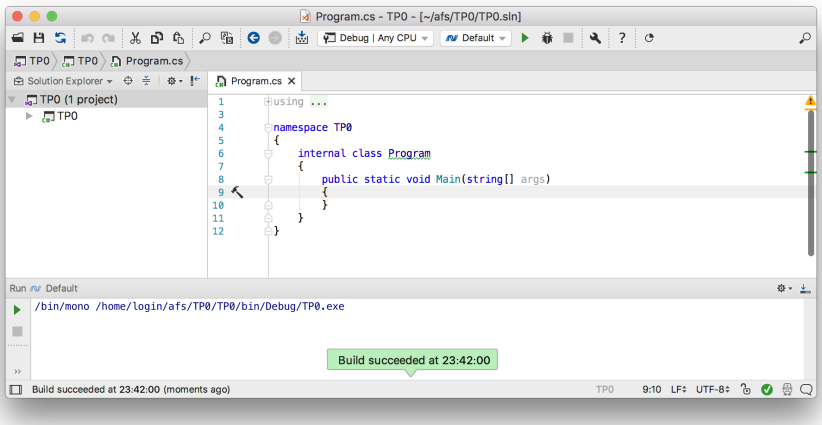
On the right is the editor window. By default, the "Program.cs" file is already opened.

At the top, you will find the toolbar. For now, the only button we care about is the little green arrow pointing right. This is a shortcut to compile and run your code. Hovering over it, you will find the corresponding keyboard shortcut. Click on it.



Pictured above are the default options to compile and run your program. We don't need to change anything there, so just click "Run". The next time you need to run your program, this screen will not appear again.

Finally, we get a new panel at the bottom of the screen with the command that was executed in order to run your program, as well as the console output of the program (nothing yet). This panel is called the *console*, and you will come to know it well during this session and the next.



Now let's learn ourselves some C# !

3 C# 101

3.1 Diving into the C...

The notions you are about to see are among the very basics of the language. You will revisit most of them during the practicals of the following weeks, and dive deeper into some of the more advanced C# principles.

Once again, if anything seems unclear to you or if you wish to know more about a particular subject, you should go visit the official documentation of the language on mdsn.com. This document serves only as a shallow introduction to the language, where we assume you already have a solid grasp of a programming language's basic constructs.

In C# , just like in most languages based on the C syntax:

- Your **program** is a sequence of **instructions**.
- An **instruction** always ends with a semicolon (";").
- **Variables** are defined with a **type** and may be assigned a value.
- A **function** is a set of instructions that takes in **arguments** and may **return** a value.

Here's an example of a simple program that only outputs "kittens" when run from the command line:

```
1  using System;
2
3  namespace TP0
4  {
5      internal class Program
6      {
7          public static void Main(string[] args)
8          {
9              Console.WriteLine("kittens");
10         }
11     }
12 }
```

You can safely copy-and-paste this program into your editor window and run it by clicking the little green arrow.

Unfortunately for everyone, most programs you will be asked to write are slightly more complex.

Let's forget about all the boilerplate code and only focus on what you really need to understand for now:

```
1  public static void Main(string[] args)
2  {
3      Console.WriteLine("kittens");
4  }
```

This part defines a function named `Main` that takes in a list of character strings and returns nothing (`void`). You can forget about the keywords `public` and `static` for now, you will re-encounter them in a later practical.

As for `Console.WriteLine`, as you've probably guessed by now, it is a function that takes in a character string and prints it on its own line in your console.

During this practical, you will be asked to write functions of varying prototypes and behaviors. The prototypes will always be specified. Please respect these prototypes, otherwise your functions **will not be graded**.

Here's an example of how you would write and test your functions in your `Program.cs` file:

```
1  using System;
2
3  namespace TP0
4  {
5      internal class Program
6      {
7          public static void Main(string[] args)
8          {
9              Console.WriteLine(Test(2022));
10         }
11
12         public static int TheAnswer()
13         {
14             return 42;
15         }
16
17         public static bool Test(int answer)
18         {
19             return answer == TheAnswer();
20         }
21     }
22 }
```

The `Main` function will be called at the start of your program. As such, it's the designated place where you will test your functions!

3.2 Declaring variables

The syntax for declaring variables is as follows:

```
1  int yourGrade;   // Declaration
2  yourGrade = -42; // Assignment
3  yourGrade = 0;   // Re-assignment
4  double pi = 3.14; // Declaration + Assignment
```

Here, `int` and `double` represent the **type** of your variables, respectively signed integer and double precision floating point number. In OCaml, you were used to the compiler inferring the type of your variables. There was also no such concept as declaring a variable without giving it a value, or re-assigning a variable without re-declaring it.

Be that as it may, you can opt in to type inference by using the `var` keyword.

```
1  var hey = "that's pretty good"; // `hey` is a string
```

Beware! Your variables declared with the `var` keyword are still statically typed: there is no such thing as changing the type of a variable through re-assignment. Also, you cannot declare a variable with `var` without initializing it to some value.

For instance, the following few lines will result in compiler errors:

```
1  var what;  // Error: Implicitly-typed local variable must be initialized.
2  var pi = 3.14;
3  pi = "3.14";
4  // Error: Cannot convert source type `string` to target type `double`.
```

Here's a list of the most common types you will come to use in C# :

Identifier	Example	Description
`bool`	`true`	A boolean value with only two possible states: `true` or `false`
`byte`	42	Integer from 0 to 255
`short`	420	Integer from -32768 (-2^{15}) to 32767 ($2^{15}-1$)
`int`	-50000	Integer from -2^{31} to $2^{31}-1$
`long`	666L	Integer from -2^{63} to $2^{63}-1$
`ushort`	420	Integer from 0 to $2^{16}-1$
`uint`	50000	Integer from 0 to $2^{32}-1$
`ulong`	666L	Integer from 0 to $2^{64}-1$
`float`	4.2f	Simple precision number from -3,402823e38 to 3,402823e38
`double`	4.2d or 4.2	Double precision number from -1,79769313486232e308 to 1,79769313486232e308
`char`	'a'	Single character
`string`	"epita"	A sequence of zero or more characters

3.2.1 Arithmetic operators

Here's a list of the most common arithmetic operators you will have to use in C# :

Usage	Meaning
`-a`	Negation of `a`
`a + b`	Addition of `a` and `b`
`a - b`	Subtraction of `b` from `a`
`a * b`	Multiplication of `a` and `b`
`a / b`	Division of `a` with `b`
`a % b`	Remainder of the division of `a` with `b` (Modulo)

3.2.2 String properties, methods and operators

In this practical, you will need the following string utilities:

Usage	Result
`s.Length`	Length of the string `s`
`s.Substring(n)`	A copy of the string `s` starting at index `n`
`s[n]`	The character at index `n` in the string `s`

3.3 Conditional branching

3.3.1 The `if/else/else if` statements

As you would expect, C# defines its own syntax for the `if` and `else` construct:

```
1  var cool = "cat";
2  if (cool == "cat")
3  {
4      // Do something interesting.
5  }
6  else
7  {
8      // Do something else.
9  }
```

Both the 'if' and 'else' constructs expect to be followed by either a single statement or a **block**. A block is defined as a sequence of one or more statements enclosed in curly braces ("{" and "}"). Don't forget the curly braces when you want to have more than a single statement in a branch!

As such, the following two code snippets are equivalent:

```
if (1 + 1 == 2)
{
    Console.WriteLine("Sounds good");
}
else
{
    Console.WriteLine("Doesn't look like anything to me");
}
```

```
if (1 + 1 == 2)
    Console.WriteLine("Sounds good");
else
    Console.WriteLine("Doesn't look like anything to me");
```

Another consequence of this definition is that you can chain multiple `if` statements without having to explicitly nest them into each other:

```
if (1 + 1 == 2)
{
    // Do something important.
}
else
{
    if (2 + 2 == 3)
    {
        // Where did we go wrong?
    }
}
// Becomes
if (1 + 1 == 2)
{
    // Do something important.
}
else if (2 + 2 == 3)
{
    // Where did we go wrong?
}
```

Note: `cool == "cat"` is called a boolean expression, as in "an expression that evaluates to a boolean value" (in C# , either `true` or `false`). The `if` construct will **always** expect to be provided with a boolean value between its parentheses.

Beware! C# 's boolean operators are slightly different from those you have used in OCaml. As such, please do check out **the documentation** for an exhaustive list of all operators defined in C# .

3.3.2 Comparison operators

Here's a list of the most common comparison boolean operators you will have to use in C# :

Usage	Meaning
`a == b`	`a` is equal to `b`
`a != b`	`a` is different from `b`
`a < b`	`a` is smaller than `b`
`a > b`	`a` is greater than `b`
`a <= b`	`a` is smaller than or equal to `b`
`a >= b`	`a` is greater than or equal to `b`

3.3.3 Logical operators

You can combine multiple boolean expressions through the use of logical boolean operators. In the table below, `A` and `B` are boolean expressions.

Usage	Meaning
`A && B`	`A` AND `B`
`A || B`	`A` OR `B`
`!A`	NOT `A`

Just like in OCaml, those operators are **lazy**: they will be evaluated left to right and stop as soon as a final value can be inferred. For instance, in the following code

```
1  bool isCat = (IsAnimal(thing) && HasWhiskers(thing)) || Meows(thing);
```

When `IsAnimal(thing)` returns `false`, `HasWhiskers(thing)` is not evaluated since `false && anything`, by definition, will always evaluate to `false`. Similarly, when `IsAnimal(thing) && HasWhiskers(thing)` evaluates to `true`, `Meows(thing)` is not evaluated since `true || anything` always evaluates to `true`.

3.3.4 Ternary operator

There are times when we want to use conditional branching within expressions. Unfortunately, in C# , the `if/else` construct is a **statement** and not an **expression**. Instead, we use the **ternary operator**:

```
1  int sign = x < 0 ? -1 : 1;
```

Think of it as of a question: Is `x` inferior to `0`? If so, use `-1`, otherwise use `1`.

Beware! *Do not overuse the ternary operator. While it is possible to nest ternary expressions within each other, you should always favor code readability over conciseness. Remember that most of the time in programming is spent reading code and not writing it!*

3.3.5 The `switch` statement

Remember pattern matching? You're about to meet its far less competent cousin. The `switch` statement allows you to test an expression against a set of predefined values and execute a different set of instructions for each. The syntax is as follows:

```
switch (EXPRESSION0)
{
case EXPRESSION1:
    // Do something meaningful
    break;
case EXPRESSION2:
    // Do something for the greater good
    break;
default:
    // Do something else I guess
    break;
}
```

Here, `EXPRESSION{0,1,2}` can represent any expression, as long as the type of all expressions match. The `default` branch is optional, and will only be visited when no case matches. For instance, in the following code:

```
switch (40 + 2)
{
case 7 * 6:
    Console.WriteLine("What a great coincidence");
    break;
case 20 * 2:
    Console.WriteLine("Close enough");
    break;
default:
    Console.WriteLine("Who needs maths anyway");
    break;
}
```

Only the first branch will be visited and `What a great coincidence` will appear on the screen.

Beware! *Do not forget to add the* `break;` *statement at the end of your cases. Otherwise, the program will jump directly into the following case even if it doesn't match. As an exercise for the reader, what does the following code output for all values of* `count` *between* `0` *and* `6`*?*

```
switch (count)
{
case 0:
case 1:
case 2:
    Console.WriteLine("Too few kittens");
    break;
case 3:
case 4:
    Console.WriteLine("An acceptable amount of kittens");
    break;
default:
    Console.WriteLine("An unorthodox number of kittens");
    break;
}
```

3.4 A few more tools to complete this practical...

In order to complete this practical, you will need a few more functions than the ones we've mentioned previously. Namely:

- `Console.ReadLine()`: reads one line of input from the user and returns it as a `string`.
- `int.Parse(string)`: takes a `string` and converts it to an `int` value.
- `DateTime.Today`: a data structure that represents the date of the current day. Try to write `DateTime.Today.` in your editor, and don't forget the trailing period: the editor will automagically suggest the fields you might be interested in!

4 Exercises

The following few exercises will only require the C# notions taught in this document. As such, in many cases, recursion is your best bet.

Please ensure your functions handle all cases! As it turns out, negative numbers do exist...

4.1 Hello World

You'll forgive the lack of originality. In this exercise, you will have to print `Hello World!` in the console.

```
1 static void HelloWorld();
```

4.2 Say Hello

Saying Hello is GREAT! Know what's even better? Remembering someone's first name. Please ask for the user's first name, and then greet them appropriately.

Example in which the even-numbered lines are user input:

```
1 What's your name?
2 Alex
3 Well hello Alex!
```

```
1 static void SayHello();
```

4.3 Calculate someone's age

You have to ask the user's year of birth, and print on the console an approximation of their age.

Example, where the even-numbered lines are user input:

```
1 What's your year of birth?
2 1999
3 Looks like you're around 18!
```

```
1 static void CalcAge();
```

4.4 Absolute value

The `Absolute` function returns the absolute value of the integer `x`.

```
1 static int Absolute(int x);
```

4.5 Factorial

The `MyFact` function returns the factorial of the unsigned integer `n`.

```
1 static uint MyFact(uint n);
```

4.6 Power

The `MyPow` fucntion returns the number `x` raised to the power of `n`. **Beware!** You MUST handle negative values of `n`.

```
1  static double MyPow(double x, int n);
```

4.7 Fibonacci

The `MyFibo` returns the `n`th term of the fibonacci series, which is recursively defined as:

- $f_0 = 0$
- $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$

```
1  static uint MyFibo(uint n);
```

4.7.1 String manipulation

The `ChangeChar` function returns the string `s` with its `n`th character swapped with `c`. When `n` is greater or equal to the length of the string, return the string unchanged.
Example:

- `ChangeChar("hello", 'a', 1) = "hallo"`
- `ChangeChar("hello", 'n', 6) = "hello"`

```
1  static string ChangeChar(string s, char c, uint n);
```

4.7.2 GCD

The `MyGcd` function returns the Greatest Common Divisor of the two integers provided as parameters.

```
1  static uint MyGcd(uint a, uint b);
```

4.8 BONUS

4.8.1 Square Root

The `MySqrt` function returns the approximation of the square root of a positive number n over i iterations, using the Heron algorithm recursively defined as:

- $r_0 = \frac{1}{2}$
- $r_{i+1} = \frac{r_i + \frac{n}{r_i}}{2}$

```
1  static double MySqrt(double n, uint i)
```

4.8.2 String reverse

The `MyReverseString` function returns the string `s` reversed. Obviously, you are strictly forbidden from using `.Reverse()` or any fancy 21st century tools: the only authorised operators are described in subsubsection 3.2.2!
Example: `MyReverseString("hello") == "olleh"`

```
1 static string MyReverseString(string s);
```

4.8.3 Palindrome

The `MyIsPalindrome` function returns a boolean value indicating whether the string `s` is a palindrome. Again, you are forbidden from using any function or operator not defined in this document!
Example:

- `MyIsPalindrome("hello olleh") == true`
- `MyIsPalindrome("kayak") == true`
- `MyIsPalindrome("Kayak") == false`

```
1 static bool MyIsPalindrome(string s);
```

4.9 Calculate real age!

Now you have to ask the full birth date of the user and print their real age on the console.

Example (you know the drill):

```
1 What's your year of birth?
2 1999
3 What's your month of birth?
4 12
5 What's your day of birth?
6 11
7 Looks like you're exactly 17!
```

```
1 static void CalcRealAge();
```

These violent deadlines have violent ends.