Practical C#5 : Space Invader

1 Guidelines for handing it

At the end of this tutorial, you have to submit an archive that respects the following architecture :

```
rendu-tp-login_x.zip
|-- login_x/
    |-- AUTHORS*
    |-- README
    |-- SpaceInvader.sln*
    |-- SpaceInvader*
        |-- Everything but bin/ and obj/
```

Of course, you must replace "`login_x`" by your own login. All the files marked with an asterisk are *MANDATORY*.

Don't forget to verify the following before submitting your work :

— The `AUTHORS` file must be in the usual format (a *, a space, your login and a line feed).
— No `bin` or `obj` directory in your project.
— **The code must compile !**

2 Introduction

2.1 Objectives

During this tutorial, you will introduced one of the most famous concepts of programming : Object-Oriented Programming (OOP). Furthermore, this way of programming is the keystone of languages like C# or C++.

In order to learn how to use objects in C#, you will study different structures such as enumerations, structs and of course : classes.
Classes represent the basis of OOP. Hence, we will study in detail its different components and functionalities :
— Attributes.
— Constructors.
— Methods.
— Inheritance.

We will then study one particular type of class : exceptions.

3 Course

3.1 Object

An object is a container, represented by data and a behaviour. Data are represented by its attributes (variables or constants contained inside the structure). Its behaviour is defined by its methods.

3.2 Classes

Classes are the most commonly used data representation in C#. They represent a type of data that you can instantiate. Which means that if we create a class 'Dog', it will be possible to create multiple objects of type Dog. The interface of a class is the grouping of all its public attributes and methods.

3.2.1 Declare a Class

A class can be defined using the keyword `class` followed by the name of the class. Like for every other variable it is possible to give different access levels to a class (`public`, `private` etc...), which will determine who can create an instance of this class.

```
public class MyClass
{
  public int myAttribute;
  public void myMethod()
  {
    Console.WriteLine(myAttribute);
  }
}
```

3.2.2 Instantiate an object

A class is not an object, it defines a type of object. Hence, if we want to use methods from a class, we have to create an instance of this class.

To do so we can use the keyword `new` :

```
MyClass myObject = new MyClass();
```

It is now possible to access attributes and methods of 'myObject' :

```
myObject.myAttribute;
myObject.myMethod();
```

3.2.3 Constructors

The line of code from the previous section is calling the constructor of 'MyClass' (`MyClass()`). This constructor will create an instance of our class. It is mandatory if we want to access the attributes of our object :

```
MyClass myObject;
myObject.myAttribute; // Error
```

It is possible to customize the object's creation by adding our own constructor to the class. To do so, we have to create a function with the name of our class, which does not have a return type. We can then add parameters to our constructor. Remark : It is possible to create multiple different constructors.

```
public class MyClass
{
  public int myAttribute;
  public MyClass(int value)
  {
    myAttribute = value;
  }
}
```

3.2.4 Object and References

There exists another way to create an object : assigning to it the value of another, already existant object of the same type. However, classes are referenced types, so the two objects would be linked to the same data.

```
int a = 0;
int b = a;
b++; // b == 1 && a == 0
MyClass object1 = new MyClass(1);
MyClass object2 = object1;
object2.myAttribute++;
// object1.myAttribute == 2 && object2.myAttribute == 2
```

In the same way, as you have seen in the previous tutorial, if we modify an object inside a function it will be modified outside of it.

3.3 Inheritance

Inheritance is a really important concept in OOP, and it will be really useful for you when coding your video game project. The big point about inheritance is to group attributes and methods shared by different classes in order to divide the code. It also enables the use of containers with objects of different type.

```
Vehicle[] vehicles = new Vehicle[2];
vehicles[0] = new Car();       // Car inherits from Vehicle
vehicles[1] = new Motorbike(); // Motorbike inherits from Vehicle
```

To create inheritance we use the following syntax :

```
public class Vehicle
{
  public string     brand;
  protected string  engine;
  private string    name;
}
public class Car : Vehicle
{
  // Interface of the class.
}
```

3.3.1 Access level

The different access rights are *public*, *private* and *protected*. They allow the coder to determine who will be able to access those values or functions. A private attribute will only be accessible by objects of its class, whereas a public one will be accessible anywhere. A class `Car` (child class), inheriting from a class `Vehicle` (parent class) will inherit from every attribute and method labelled with the *public* or *protected* access levels in the `Vehicle` class. Protected attributes and methods are only accessible by the parent class and its child classes.

```
public class Car : Vehicle
{
  public void shoot()
  {
    Console.WriteLine(engine);  // Possible
    Console.WriteLine(name);    // Impossible
  }
}
class Program
{
  public void play()
  {
    Car myCar = new Car();
    Console.WriteLine(myCar.brand);   // Possible
    Console.WriteLine(myCar.engine);  // Impossible
  }
}
```

However, it is possible to access private and protected attributes outside a class using accessors. Accessors are used like values. You can code them using the `get`, `set` and `value` keywords. An accessor is named like the values it is linked to, but with an uppercase on the first letter.

```
private int x;
public int X
{
  get { return x; }
  set { x = value; }
}
```

3.3.2 Class Hierarchy

It is possible to create multiple classes inheriting from a same parent or even a class inheriting from a child class. It is then possible to create a class hierarchy. For example, a class `Motorbike` inheriting from `Vehicle` or a class `ElectricalCar` inheriting from the class `Car`.

3.4 Structures

Structures are really similar to classes with some differences. Structures can be instantiated without the `new` keyword, they are value types (as opposed to referenced types : they are not modified after being called by a function). It is also impossible to use inheritance with structures. Hence, structures are most likely to be used when we only want to store variables. To define a structure in C# you have to use the keyword `struct` :

```
public struct Point
{
  int x;
  int y;
}
```

It is possible to create constructors in a `struct` but the new constructor will have to take parameters. A struct is usually initialized in this way :

```
Point pos;
pos.x = 10;
pos.y = 5;
```

3.5 Enumeration

An enumeration is a type containing a given number of constants. You can create an enumeration like this :

```
enum Color { Blue, Red, Yellow, Green = 5 };
// Cannot be declared in a function
```

It is then possible to create variables of type `Color`.

```
Color myColor = Color.Blue;
```

Remark : constants of an enum are related to a number.

```
Color myColor = (Color)1; // myColor == Color.Red
int n = (int)Color.Green; // n == 5
```

3.6 Exceptions

During the execution of your code, if you use an instruction that cannot be executed by your program, an exception is raised. For example :

```
int n = 1 / 0; // WTF
```

The DivideByZero exception will then be raised.

3.6.1 Try Catch

In order to avoid such errors, you can either code properly, or you can use the `try catch` statement : it is composed of two keywords `try` and `catch`. Inside the brackets related to the `try` you will write the code that might raise an error.

```
try
{
  int i = 0;
  // ...
  int n = 1 / i;
}
```

Then inside of the `catch` you write the code corresponding to the behaviour you want in case of an exception.

```
catch (Exception e) // Catches any kind of exception
{
  Console.WriteLine("You're dividing by 0!");
}
```

You can then use the keyword `finally`. The code inside of its brackets will be executed whatever happens inside of the `try`.

3.6.2 MyException

An exception being an object it is possible to create your own exception by creating a class inheriting from the class Exception.

```
public class MyException : Exception
{
  public MyException() : base() {};
  public MyException(string message) : base(message) {};
}
```

3.6.3 Throw

It is also possible to generate an exception with the keyword `throw` when you want to verify that a situation will never happen.

```
if (i == 0)
  throw new Exception("0, Again!!");
int n = 1 / i;
```

4 Exercise

Now let's practice with a little game. The goal of this exercise is to do a simplified 'Space Invaders', you'll be able to use all the notions you've just seen and you will understand why Object-Oriented Programming is so useful. As it's your first time with OOP we'll go step by step on this project : you will need at least 4 different files :

4.1 Program.cs

'Program.cs' will contain your main function where everything will be initialized and where you will handle your keyboard inputs.

Advice : the `ReadKey` function is blocking the execution, so you'll need a loop that will only stop in case of keyboard input (msdn : `isKeyAvailable`), you will then only need to get the key pressed with a `Readkey` and handle its effects. You can after that enter once again the previous loop, using that trick your game won't be stopped by your keyboard inputs.

Code the function `get_param` that takes the string defining the enemies' design and the one defining the player's design. This function should fill these parameters through a 'ReadLine' on the console then get the difficulty through another one. To get the latter, use a `try catch` to check that the input is an integer bigger than 0.

```
int get_param(ref string player_design, ref string enemy_design);
```

4.2 Character.cs

`Character.cs` will contain your `Character` class, it will be the core of your game, it contains all the methods shared by the enemies and the player.

You will need at least the following attributes (please care about giving them a proper visibility : `public, private, protected`) :

```
int nb_shots ;
string design ; // what the ship will look like
int last_shot_time;
// int containing the time since last shot
string refresh_str;
// string filled with spaces to clean the ship sprite when moving
int length; // size of the design
int life ;
int x; // advice : avoid values greater than 79 + length
int y; // 23 maximum on the default console window
List<Shot> shots; // list containing all the ship's active shots
int time; // int containing the game time
```

Here's the constructor :

```
Character(string design, int x, int y)
{
  // initialize all your attributes
}
```

You will need the following methods :

```
void print();
```

Prints your ship and erases its sprite when moving (recall : if you don't use `Console.Clear()` then nothing is cleared after being printed. We don't use it here to avoid flickering and to accelerate the game). Use `Console.setcursor(x,y)` to write at a specific place in the terminal.

```
void shoot(string design, int direction, int speed);
```

Creates a new object `shot` and adds it to the `shot` list of the `character`, the `speed` argument is here to increment the time between each shot. A substraction between `time` and `last_shot_time` could be useful.

```
string fill_refresh_str();
```

This little function returns a `string` filled with spaces of the ship's size, it will allow us to clean the ship's movements.

```
public void clear_shots();
```

Similar to the last one, it clears the ships's shots when destroyed.

The following function `delete_shot` is given to you to avoid some difficulties with collisions and to help you with the shots' removal

```
void delete_shot(Shot shot, List<Enemy> enemies)
{
    for (int i = 0; i < enemies.Count; ++i)
        if (shot.y == enemies[i].y && shot.x >= enemies[i].x
            && shot.x <= enemies[i].x + enemies[i].length)
        {
            shot.collision = true;
            enemies[i].life -= 10;
        }

    if ((shot.y < 1 && shot.direction == -1)
        || (shot.y > 22 && shot.direction == 1)
        || shot.collision)
    {
        Console.SetCursorPosition(shot.x, shot.y);
        Console.Write(" ");
        shots.Remove(shot);
    }
}

void delete_shot(Shot shot, Player player)
{
    if (shot.y == player.y && shot.x >= player.x
        && shot.x <= player.x + player.length)
    {
        shot.collision = true;
        player.life -= 10;
    }

    if ((shot.y < 1 && shot.direction == -1)
        || (shot.y > 22 && shot.direction == 1)
        || shot.collision)
    {
        Console.SetCursorPosition(shot.x, shot.y);
        Console.Write(" ");
        shots.Remove(shot);
    }
}
```

4.3 Player.cs

`Player.cs` contains your `Player` class, it will be the ship that you will control, it inherits from `Character` :

Here is its constructor :

```
Player(string design, int x, int y) : base(design, x, y)
```

`base` is a keyword used to pass arguments to the parent class, you can imagine `Character` instead of `base` if you don't get it. The content of this contructor is only to give a custom value to `life` for your ship. No need to define or assign anything else, everything is done by the `Character` class, it's the power of OOP.

The only required method is `update`, here's the prototype :

```
void update(int time, List<Enemy> characs);
```

This function updates the `time` attribute (please think about the keyword `this`), calls the `print` method to display the ship and updates every element of the `Shot` list, also call the function that checks the collisions for each element.

4.4 Enemy.cs

`Enemy.cs` will contain your `Enemy` class, same as `player`, it inherits from `Character` :
Here's the constructor :

```
Enemy(string design, int x, int y) : base(design, x, y)
```

Nothing to be done except giving a value to `life`
You will need two methods here :

```
void update(int time, List<Enemy> characs);
```

Same as `player` but you will also need to call `shoot()` because enemies will have to shoot automatically without any input. Finally you will also need to call `move` defined just below. Please consider using `time` to slow down `move` and `print` if you don't want your game to be too fast (`time` modulus a big value == 0 should do it)

```
void move();
```

Modifies `x` randomly and increments `y`
`Shot.cs` contains the `Shot` class and its attributes :

```
string design;
int x;
int y;
int direction; // 1 or -1 depending on whether the shot goes down or up
bool collision; // if the shot hit a ship
```

Here's the constructor :

```
Shot(string design, int x, int y, int direction)
```

Please do not forget to initialize all the attributes inside.
The class will have 2 methods :

```
void update(int time);
```

You will modify the `y` position of the shot and will call the last method `print`

```
void print();
```

Simple display of the shot, do not forget to erase its movement.

4.5 Bonus

For this part you're free to do whatever you want in a few lines, think about the power of OOP, for instance adding a boss that inherits from the `Enemy` class should take you no more than 20 lines, same for special shots that would inherit from the `Shot` class.

Here's a few ideas but please do not restrain yourself :

— Different enemies (design, shots, boss, etc ...)
— Different shots, auto guided shots etc.
— Different enemy waves with increasing difficulty
— Asteroïds to avoid
— ...

Using more classes with inheritance or exceptions will be greatly appreciated. Impress us !

The Bullshit is strong with this one...