TP C#4 : Simple Monopoly

Submission instructions

At the end of this tutorial, your git repository must respect the following architecture:

```
tp4-firstname.lastname/
|-- README
|-- .gitignore
|-- Monopoly/
    |-- Monopoly.sln
    |-- Monopoly/
        |-- BeginCell.cs
        |-- Cell.cs
        |-- Company.cs
        |-- Game.cs
        |-- GameLayout/
            |-- game
        |-- Jail.cs
        |-- Luck.cs
        |-- Player.cs
        |-- Property.cs
        |-- Program.cs
        |-- Serializer.cs
        |-- Special.cs
        |-- Station.cs
        |-- Street.cs
        |-- Tax.cs
        |-- Monopoly.csproj
        |-- Tout sauf bin/ et obj/
```

Remember to check the following before submission:

- Replace *firstname.lastname* with your login.
- Submitting `README` is mandatory.
- No `bin` or `obj` in your project.
- Respect the given prototype for every function..
- Remove all tests from your code.
- **The code must compile !**

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).

Contents

1 Introduction **3**
1.1 Objectifs . . . 3

2 Courses **3**
2.1 Object oriented programming . . . 3
2.1.1 Why ? . . . 3
2.1.2 Classes and objects . . . 3
2.1.3 `null` and `this` . . . 4
2.1.4 Attribute and method accession . . . 4
2.1.5 Inheritance . . . 5
2.1.6 Abstract classes . . . 8
2.1.7 Interface . . . 9
2.1.8 Accessibility . . . 10
2.2 Lists . . . 11

3 Exercices **12**
3.1 Property.cs . . . 12
3.1.1 Attributes . . . 12
3.1.2 getter/setter . . . 12
3.1.3 Constructor . . . 13
3.2 Street.cs . . . 13
3.2.1 Attribute . . . 13
3.2.2 Constructor . . . 13
3.3 Company.cs . . . 13
3.3.1 Constructor . . . 13
3.4 Station.cs . . . 14
3.4.1 Constructeur . . . 14
3.5 Player . . . 14
3.5.1 Attributes . . . 14
3.5.2 Getters . . . 14
3.5.3 Constructeur . . . 14
3.5.4 Methods . . . 15
3.6 Special.cs . . . 16
3.6.1 Méthodes . . . 16
3.7 Tax.cs . . . 16
3.7.1 Attributes . . . 16
3.7.2 Getter . . . 16
3.7.3 Constructor . . . 17
3.7.4 Méthodes . . . 17
3.8 Luck.cs . . . 17
3.8.1 Constructor . . . 17
3.8.2 Méthodes . . . 17
3.9 Game . . . 18
3.9.1 Attributes . . . 18
3.9.2 Constructor . . . 18
3.9.3 Getters and Setters . . . 18
3.9.4 Methods . . . 18

1 Introduction

1.1 Objectifs

The objective of this tutorial is to teach you the basics of object-oriented programming (OOP) in C#. OOP is a programming paradigm, i.e. a way to represent a problem to make it simpler. The functional programming that you have seen in Caml is another paradigm.

2 Courses

2.1 Object oriented programming

2.1.1 Why ?

Before you learn how to create and use objects, you need to understand why they are useful. To do so, we will start with an example.

Let's imagine than we want to create a racing game in which each car has a speed, an acceleration and a name. If we wanted two cars, we could represent them as follows:

```
1 string name1 = "averynicecar";
2 string name2 = "aslowercar";
3 int speed1 = 0;
4 int speed2 = 0;
5 int acceleration1 = 100;
6 int acceleration2 = 50;
```

As you may see, this is not clean, and it would quickly become unworkable if we chose to add new cars or new properties. Fortunately, thanks to OOP, we don't have to write things like that.

2.1.2 Classes and objects

Classes are a way to represent new data types, and objects are **instances** of classes. To make it easier, if we take the car example, we could see classes as a building plan and objects as the cars themselves.

The goal of classes, therefore, is to define the different properties of objects. There are two kinds of properties : **attributes** and **methods**.

We can define **attributes** as variables declared inside a class. They can be:

- **static attributes**, which means that they are linked to a class and will have the same value inside every instance of this one.
- **instance attributes**, which means that they are owned by an object and will have a different value inside every instance of a class.

As for **methods**, they are functions defined within the scope of the class to give a behaviour to objects. Like attributes, they can be:

- **static methods**, which means that they can be called without instantiating an object.

- **instance methods**, which means that they can only be called using an object, and they directly depend on its attributes.

A **constructor** is a special method which is called whenever an object is created. It is particularly useful when we want to initialize attributes. Its name is the name of the class and it has no return type.

Here is an example of a class:

```
public class Car
{
    // Attributes
    public string name;
    public int speed;
    public int acceleration;

    // Constructor
    public Car(string name, int speed, int acceleration)
    {
        // The 'this' keyword is used to make a distinction
        // between an attribute and a constructor parameter
        // which shares the same name
        this.name = name;
        this.speed = speed;
        this.acceleration = acceleration;
    }

    // Methods
    public void Accelerate()
    {
        speed += acceleration;
    }

    public static void MakeSomeNoise()
    {
        Console.WriteLine("VROUM VROUM");
    }
}
```

2.1.3 `null` and `this`

It is possible to give the value of the absence of an object giving the `null` keyword to a variable or an attribute.

It is also possible, inside a method, to reference the current instance of an object (meaning the instance which is calling the current method) using the `this` keyword.

2.1.4 Attribute and method accession

The syntax to create an object is the following :

```
Car fastCar = new Car("AlbinMobile", 0, 50);
```

The constructor is called with the specified parameters. Once the object is created, you can access its properties and modify them as follows:

```
string carName = fastCar.name;
// carName is now equal to "AlbinMobile"
fastCar.acceleration += 5;
// fastCar.acceleration is now equal to 55
fastCar.Accelerate();
// fastCar.speed is now equal to 55
```

As mentioned before, static methods can be called without instantiating an object:

```
Car.MakeSomeNoise();
// This line will print "VROUM VROUM" on the standard output
```

2.1.5 Inheritance

Inheritance is a concept of object oriented programming which allows the user to gather communal properties of different classes.

Let's imagine that we wished to add bikes to our favorite racing game, defined by the following class:

```
public class Bike
{
    // Attributes
    public string name;
    public int speed;
    public int acceleration;

    // Constructor
    public Bike(string name, int speed, int acceleration)
    {
        this.name = name;
        this.speed = speed;
        this.acceleration = acceleration;
    }

    // Methods
    public void Accelerate()
    {
        speed += acceleration;
    }

    public static void MakeSomeNoise()
    {
        Console.WriteLine("BRR BRR");
    }
}
```

You may have noticed that this class is similar to the *Car* class, and there are a lot of lines to copy just to change the display of *MakeSomeNoise*. Thanks to inheritance, we don't have to do that.

It is possible to create *Vehicle* and to say "**a car is a vehicle**". Doing that, the *Car* class will inherit the properties of *Vehicle*, without copying them manually. Here is an example of inheritance:

```
1  public class Vehicle
2  {
3      // Attributes
4      public string name;
5      public int speed;
6      public int acceleration;
7
8      // Methods
9      public void Accelerate()
10     {
11         speed += acceleration;
12     }
13 }
14
15 public class Car: Vehicle
16 {
17     // The attributes are already defined in Vehicle.
18     // We don't need to write them again.
19
20     // Constructor
21     public Car(string name, int speed, int acceleration)
22     {
23         this.name = name;
24         this.speed = speed;
25         this.acceleration;
26     }
27
28     // Methods
29     public static void MakeSomeNoise()
30     {
31         Console.WriteLine("VROUM VROUM");
32     }
33 }
```

```
1  public class Bike: Vehicle
2  {
3      // Constructor
4      public Bike(string name, int speed, int acceleration)
5      {
6          this.name = name;
7          this.speed = speed;
8          this.acceleration;
9      }
10
11     // Methods
12     public static void MakeSomeNoise()
13     {
14         Console.WriteLine("BRR BRR");
15     }
16 }
```

The inherited class (here *Vehicle*) is called **superclass**, and the derivative is called **subclasses**.

This code is already cleaner than the one we wrote at the beginning. Nevertheless, we could still do better. In fact, using this implementation, it is possible to instantiate *Vehicle* objects, which is meaningless. We want to be able to create either cars or bikes. This is why we will introduce **abstract classes**.

2.1.6 Abstract classes

An abstact class is a class which cannot be instantiated. It can only be used to be a superclass. It can be defined using the keyword **abstract**. Inside the scope of an abstract class, a method can be declared without being implemented. It will be implemented inside the scope of the subclasses using the keyword **override**.

We can once again rewrite our code as follows:

```
1  public abstract class Vehicle
2  {
3      // Attributes
4      public string name;
5      public int speed;
6      public int acceleration;
7
8      // Methods
9      public void Accelerate()
10     {
11         speed += acceleration;
12     }
13
14     public static abstract void MakeSomeNoise();
15 }
16
17 public class Car: Vehicle
18 {
19     // Constructor
20     public Car(string name, int speed, int acceleration)
21     {
22         this.name = name;
23         this.speed = speed;
24         this.acceleration;
25     }
26
27     // Methods
28     public static override void MakeSomeNoise()
29     {
30         Console.WriteLine("VROUM VROUM");
31     }
32 }
```

```
1  public class Bike: Vehicle
2  {
3      // Constructor
4      public Bike(string name, int speed, int acceleration)
5      {
6          this.name = name;
7          this.speed = speed;
8          this.acceleration;
9      }
10
11     // Methods
12     public static override void MakeSomeNoise()
13     {
14         Console.WriteLine("BRR BRR");
15     }
16 }
```

2.1.7 Interface

An interface is a pool of attributes and methods, without implementation. It behaves like a superclass that has noting implemented.

It is important to know that a subclass can only inherit from **one** superclass whereas it can inherit from several interfaces Here is an example:

```
1  interface Vehicle
2  {
3      public int speed;
4      public void Accelerate();
5  }
6  interface Wheel
7  {
8      public int numberofWheels;
9      public void ChangeWheels();
10 }
11 public class Car: Vehicle, Wheel
12 {
13     public override void Accelerate()
14     {
15         speed += 5;
16     }
17
18     public override void ChangeWheels()
19     {
20         Console.WriteLine("Changing wheels...");
21     }
22 }
```

2.1.8 Accessibility

While reading this course, you may have noticed the keyword **public** in front of attributes and methods. This keyword is used to specify the **accessibility** of the property.

More precisely, there exist three keywords to handle accessibility :

- **public** : public properties can be both accessed from the inside and from the outside of the class.
- **private** : private properties can only be accessed and modified from the inside of the class.
- **protected** : protected properties can only be accessed and modified from the inside of the class, and from the inside of its subclasses.

If you want to access private or protected attributes from the outside of the class, it will be necessary to create methods called **getters**. It is useful when we don't want to allow the user to modify the attributes as he wishes. For instance, in our car example, we don't want the speed to be negative. Defining the speed attribute as protected, therefore, could be a good idea.

Important

The course of this practical is very dense and can be scary at first, but it is really important. You are advised to reread it with a clear head until you feel that you have understood everything. Do not hesitate to ask questions.

Note that in C# there is a particular syntax to create getters/setters in a faster way. In fact, to declare a **getter** you can proceed as follows.

```
1  public int Speed => this.speed
2
3  /*
4   * Usage:
5   */
6
7   Car car = new Car("paroumobile", 0, 50);
8   Console.WriteLine(car.Speed);
```

For the setter, we can also use a simplified syntax:

```
1  public int Speed
2  {
3      set { this.speed = value; }
4  }
5
6   /*
7    * Usage:
8    */
9
10  Car car = new Car("paroumobile", 0, 50);
11  car.Speed = 42;
```

It is also possible to combine both declarations:

```
public int Speed
{
    get => this.speed;
    set { this.speed = value; }
}
```

2.2 Lists

There is a **List** class in C#. Unlike arrays, lists can hold a variable number of elements. They, therefore, are very useful when we frequently have to insert and remove elements. To create a list, you just have to call its constructor:

```
List<int> list = new List<int>();
```

Observation

This line will create an integer list. In fact, the type of the elements from the list is defined inside the brackets. Therefore, the type of a character list would be **List<char>**.

Here is a non-exhaustive enumeration of list attributes and methods:

```
List<double> list = new List<double>();

// Add method:
// Insert the given element at the end of the list
list.Add(2.5);
list.Add(3);
list.Add(-1);
// List content : [2.5, 3, -1]

// Count attribute:
// Number of element contained in the list
Console.WriteLine("{0}", list.Count);
// Prints 3

// You can get an element of the list the same
// way as for an array:
double f = list[1];
// f is now equal to 3
```

Documentation

There are a lot of other methods which will be useful in the future. We **strongly** advise you to read the documentation : `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=netcore-3.1`

3 Exercices

In this tutorial you will implement a simplified version of Monopoly! To do this we will use OOP. Let's make an inventory of all the elements needed to play a game of of Monopoly.

- An abstract class `Cell` to designate the squares of the game board.
- An abstract class `Special` to designate all the special cells (taxes, exchange cards, etc)
- A `Jail` class to designate the jail square.
- A `Tax` class to designate the special squares containing a tax.
- A `Luck` class which designates the special squares which allow one to have a luck card.
- A `Property` class which designates the boxes which it is possible to possess.
- A `Street` class which designates the squares that you can own which are streets.
- A `Station` class which designates the squares that you can own which are stations.
- A `Company` class which designates the boxes which can be owned which are companies.
- A `Player` class containing all the player's information.
- A `Game` class to collect the information on the board

3.1 Property.cs

Let's start by implementing the abstract class that designates the cells that a player can own. It inherits directly from the `Cell` class. This class is relatively simple and will only serve to group the different attributes common to the three possible types of properties.

3.1.1 Attributes

Declare the following attributes.

```
1 private Player owner; // The owner of the cell
2 protected int price; // The price to buy the cell
3 protected int rentCost; // The price a Player has to pay
4 protected string name; // The name of the cell
```

3.1.2 getter/setter

Declare a `getter` and a `setter` on `owner`

```
1 public Player Owner
```

Declare a `getter` on `price`

```
1 public int Price
```

Declare a `getter` on `rentCost`

```
1 public int RentCost
```

Declare a `getter` on `name`

```
1  public string Name
```

Important

Respect the names given in the subject for the attributes and for the getter/setter.

3.1.3 Constructor

Implement the class constructor.

```
1  public Property(string name, int price, int position, int rentCost)
```

Aide

By default, a property has no owner.

3.2 Street.cs

We are going to implement the `Street` class which inherits directly from the `Property` class and which adds only the attributes which are proper to it.

3.2.1 Attribute

Only one attribute distinguishes a street from a normal property and that is its color. Declare the following attribute:

```
1  private Color color; // The color indicating the group of the street.
```

3.2.2 Constructor

Implement the class constructor.

```
1  public Street(string name, int price, int position,
2              int rentCost, Color color)
```

3.3 Company.cs

In the same way, we will implement the class `Company`, which designates a company and which inherits directly from `Property`.

3.3.1 Constructor

Implement the class constructor.

```
1  public Company(string name, int price,
2              int position)
```

Indication

No additional attribute is necessary. The constructor simply calls the constructor of the parent class and gives it the value `100` for the `rentCost`.

3.4 Station.cs

To finish with the properties, we must implement the `Station` class which inherits directly from `Property`.

3.4.1 Constructeur

Implement the class constructor.

```
1  public Station(string name, int price,
2                 int position)
```

Indication

Once again, no additional attributes are needed. The constructor simply calls the constructor of the parent class and gives it the value `150` for `rentCost`.

3.5 Player

In order to represent a player, we will implement the class **Player**.

3.5.1 Attributes

Declare the following attributes.

```
1  private List<Property> possessions; //A List of all player's properties
2  private int balance; //Player's current money
3  private int position; //Player's position on the board
4  private string name; //Player's name
5  public bool jailed; //Indicates whether the player is in jail or not
```

3.5.2 Getters

Declare a `getter` on `possessions`

```
1  public List<Property> possessions
```

Declare a`getter` on `balance`

```
1  public int Balance
```

Declare a `getter` on `position`

```
1  public int Position
```

Declare a`getter` on `name`

```
1  public string Name
```

3.5.3 Constructeur

Implement the class constructor of `Player`:

```
1  public Player(string name, int initBalance, int initialPosition);
```

The list of `Property` must be initialized empty by the constructor.

3.5.4 Methods

Implement the **SendToJail** method which sends the player `player` to jail. In practice, it only changes the value of the player's `jailed` attribute.

```
1  public void SendToJail()
```

Implement the **ReceiveMoney** method that allows a player to receive money.

```
1  public void ReceiveMoney(int amount)
```

Implement the **RetrieveMoney** method which allows one to retrieve money from the player.

```
1  public bool RetrieveMoney(int amount)
```

Implement the **Buy** method that returns `true` if the player can buy the `Property p`, `false` otherwise. We will assume that the `Property` in question does not belong to any other player.

```
1  public bool Buy(Property p)
```

Tip

A player can buy a `Property` as long as they have enough money. Remember to update the list `possessions`!

Implement the **Sell** method which sells the `Property p` of a player. The method returns `true` if the sale could be done, `false` otherwise. The player receives the sale price of `p` and `p` is left without an owner.

```
1  public bool Sell(Property p)
```

Implement the **TransferTo** method which transfers an `amount` from the player to another player. It returns `true` if the transfer took place `false` otherwise.

```
1  public bool TransferTo(Player p, int amount)
```

Tip

A player can only transfer money if he has enough money!

Implement the **SellTo** method which allows one to sell Property p to another player. It returns `true` if the sale took place, `false` otherwise.

```
1  public bool SellTo(Property p, Player player)
```

Finally, we want to be able to move the player on the board. Implement the **Move** method which takes care of this by taking into account `boardSize`, the size of the board and `vector`, the number of squares the player has to move.

```
1  public void Move(int vector, int boardSize)
```

Indication

Once a player reaches the last square, he returns to the beginning.

In the class `Player` there is a method which is given to you[1]. The method `ToString` allows you to display in the terminal all the information relative to a player

Example :

```
1  Player player = new Player("Respo No Fun", 55, 0);
2  Console.WriteLine(player.ToString());
```

Writes on standard output:

```
player: 'Respo No Fun'
balance: 55£
position: 0
possessions:
```

Remarque

This function will be very useful to test the behavior of your code.

3.6 Special.cs

Let's continue by implementing the `Special` class which will add a specific behavior to the special boxes. These are the tax boxes and the chance cells. `Special` inherits directly from `Cell`.

3.6.1 Méthodes

We don't need to add any extra attributes or even a constructor. We will only need a method to modify the budget of a player so that the special effect is applied.

Implement the **ModifyBudget** method that changes the budget of `player`. The `amount` parameter is, in absolute value, the amount of the transaction.

The sign of `amount` determines whether the player wins or loses money. If `amount` is negative, **ModifyBudget** will remove money, but if the parameter is positive, the method will add money.

The method returns `true` if the operation is successful `false`, otherwise.

```
1  protected bool ModifyBudget(Player player, int amount)
```

3.7 Tax.cs

We now need to implement the special cells, and we will start with the `Tax` class which inherits directly from `Special`.

3.7.1 Attributes

As you might expect, we'll start with the attributes. Here, you will add only one, `amount`, which represents the value of the tax.

```
1  private int amount
```

3.7.2 Getter

Declare a `getter` on `amount`

```
1  public int Amount
```

[1] Because we are nice

3.7.3 Constructor

Implement the class constructor. Don't forget to initialize the attribute `position` of the superclass.

```
1  public Tax(int amount, int position)
```

3.7.4 Méthodes

Implement the **TaxPlayer** method which withdraws from the player `player` the amount given by `amount`. This method returns `true` if the operation is successful, `false` otherwise.

```
1  public bool TaxPlayer(Player player)
```

3.8 Luck.cs

It is now time to implement the `Luck` class which inherits directly from the `Special` class and applies the effects of a luck card.

3.8.1 Constructor

You don't need any additional attributes, so you have to directly implement the constructor. It only initializes the attribute the class inherits from.

```
1  public Luck(int position)
```

3.8.2 Méthodes

Implement the **MoneyEffect** method which retrieves an amount thanks to the random number `rand`[2] and which adds or withdraws it to the player `player`. This method returns `true` if the operation is successful, `false` otherwise.

```
1  public bool MoneyEffect(Player player, double rand)
```

Aide

You **must** use the function **GetAmount** which generates the amount value from `rand`.

Implement the **GetEffect** method which generates a random number and applies the associated effect to the player `player`.

In the case where the random number obtained is lower in absolute value[3] than 2, you must call the **MoneyEffect** method with the random number previously obtained as parameter. If it is greater or equal in absolute value than 2, you must call the **SendToJail** method.

This method returns `true` if the operation is successful, `false` otherwise.

```
1  public bool GetEffect(Player player)
```

Aide

You **must** use the method **GetRandomValue** to generate the random number.

[2] `rand` can be negative, in this case the function **GetAmount** will return a negative number.
[3] Math.Abs, `https://docs.microsoft.com/fr-fr/dotnet/api/system.math.abs?view=net-5.0`

3.9 Game

In order to start a game and play it, you will implement the class **Game**. In this class you will handle everything that will happen on the board on each round. This is why you're going to implement methods that will handle the players movement, the purchase of properties for the players and everything that happens in the game.

3.9.1 Attributes

We're going to declare the following attributes
Once again every attribute has to be declared as **private** !

```
private List<Cell> board;      // The board that will
                               //contain the different cells
private int boardSize;         // The board size
private List<Player> players; // The list of players
private int playersNumber;     // The number of players in the game
```

3.9.2 Constructor

Implement the constructor of the class **Game**.

```
public Game(int boardSize)
```

The **Cell** and **Player** lists have to be initialized as empty lists by the constructor.

3.9.3 Getters and Setters

It is then necessary to create the following **Getters and Setters**:
Declare a `getter` on the attribute `players`

```
public List<Player> Players
```

Declare a `getter` on the attribute `boardSize`

```
public int BoardSize;
```

Declare a `getter` on the attribute `players`

```
public int PlayersNumber
```

Declare a `getter` on the attribute `board`

```
public List<Cell> Board
```

3.9.4 Methods

First of all, we need to know if a player has won the game. To do that, you will implement the **PlayerWon** method. If the player has won, the method **PlayerWon** has to display:

```
Congratulations Sherlock, you won !
```

Of course, "Sherlock" has to be changed by the name of the winner. The method has to return **true** if the player won and **false** if the player lost.

```
1  public bool PlayerWon()
```

If a player lost the game, you have to remove him from the list of players and you have to decrement the number of players. You also have to remove the **Properties** from the player and set the owner of the property to null. Implement the method **PlayerLost** which has this behavior. Moreover, this method has to display:

```
You don't have enough money, you lost !
```

```
1  public void PlayerLost(Player player)
```

In order to play you will need some dice ! Implement the method **RollDice** that returns a random integer between 1 and 12 in order to simulate a dice throw.

```
1  public int RollDice()
```

The game that you're going to implement will be in direct interaction with the Console. This is why, to take some decisions, you will have to require an input from the user (when he is on a property). Implement the method **GetInput** that asks the **player** if he wants to buy the **property** on which he is standing. The method has to return 0 or 1 and has to display this message:

```
No one owns the street you can buy it for 1000£, you have 2000£ left.
Press 1 if you want to buy it, 0 otherwise.
```

Help

In this method, you don't need to check if the player has enough money, this will be done later.

```
1  public int GetInput(Player player, Property property)
```

Important

Your program should ask the user for an input until the value is equal to 1 or 0.

You will now have to implement the method **BuyProperty**. This method will be called given the **property** on which the player stands and the result of the method **GetInput**. This means that you must not call the method **GetInput** in **BuyProperty**.

In this method we consider that the property has no owner, therefore, the player has the possibilty to buy it with condition that he has enough money. If **input** equals 1, the user decided to buy the property. Use the method **Buy**. If the player has enough money, the program has to display:

```
You bought Baskerville Hall, your balance is now: 1000£
```

Else it has to display:

```
Insufficient funds
```

Reminder

The method **Buy** returns a value which allows the user to know if the purchase has happened or not.

```
1  public void BuyProperty(Property property, Player player, int input)
```

You will now have to implement the method **OnRegular**. The **cells** on which the **player** stands are given in parameters of this function. In this method, you can consider that you're on a cell of type **Property**, which is why you can explicitely cast the **cell** to the type **Property**.

Then you have to display the property on which the player stands:

```
You're now on the cell Baskerville Hall
```

It is in this method that you will handle everything once the players arrive on a cell. You will reuse the previous methods. Here is what you will have to do:

1. If a cell has no owner, get the decision of the player on whether he wants to buy the property using the method **GetInput**. Then use **BuyProperty** to handle the possible purchase of the property.
2. Else it means that the cell has an owner. If the owner is not **player**, the player has to play the owner. And display:

   ```
   This cell is owned by Sherlock and the rent cost is 200£
   ```

   Using the method **TransfertTo**, if the player has enough money, display:

   ```
   You paid 200£, your balance is now 1000£
   ```

   If the player doesn't have enough money, he lost, call the method **PlayerLost**.
3. In the case where the player is the owner of the cell, simply display.

   ```
   You own this property !
   ```

The method has to return **false** if the player lost and **true** if he still is in the game.

```
1  public bool OnRegular(Cell cell, Player player)
```

You will now have to implement the method **OnTax** that is called if the player happens to be on a Tax cell. This method is rather simple, it has to inform the player that he is currently on a Tax cell and tell him how much he has to pay:

```
You are now on a tax cell: you have to pay: 1000£
```

If the player has enough money withdraw the money and display:

```
You paid 1000£, your balance is now 500£
```

Of course, the price of the tax is the one of the attributes of the object **tax** given as parameter of the function. If the player does not have enough money, he lost and you have to call the method **PlayerLost**.

> Help
>
> Use the method **TaxPlayer** implemented in the class Tax !

The method has to return **false** if the player lost and **true** if he's still in the game.

```
1  public bool OnTax(Tax tax, Player player)
```

You will now have to implement the method **PlayRound**, this method has to handle the players' movements and, therefore, call the corresponding methods.

Before anything, display what value which was rolled by the player:

```
Sherlock it is your turn to play please press any key to roll the dice
```

This is probably the hardest method to implement, but do not panic, we are going to guide you. First of all, if the player **player** is in prison, he has the right to attempt to do a double with his dice. In order to do that, call the static method **AttemptDiceDouble** of the **Jail** class and return **true** because the player finished his round and he's still in the game. Move the player, if he goes past by the begin cell (if he went further that the list size) you have to give him 200£. Then, get the new cell where the player is currently. Your objective is now to know on which type of cell the user is the latter can be on a cell of type:

1. Street
2. Station
3. Company
4. Tax
5. Luck
6. Jail

> Help
>
> In C# it is possible to do a switch case on the type of the element.

According to the type of the cell, you will be able to call the corresponding method that handles the type of cell.

> Help
>
> What a surprise; you implemented the methods **OnRegular**, **OnTax**, **GetEffect** and **SendToJail** just before this one.

In the same way as for the previous method. This one has to return **false** if the player lost and **true** if he is still in the game. Remember that the previous functions also have a return value. Use them in order to determine the value of **PlayRound**

```
1  public bool PlayRound(Player player)
```

You will now implement the principal method that will put every previous method in relation: **Play**. Until a player wins, the game has to continue. At each round, you have to display:

```
Sherlock this is your turn to play please press any key to roll the dice
```

As per usual, Sherlock has to be replaced by the current players name. The display of this line has to be followed by a call to **Console.Readline**. Then call the method **PlayRound** and get the next player while being careful not to skip a player if the current player has lost and is now removed from the list !

```
1  public void Play()
```

Now to test your code, you just have to create players and use the method **AddPlayer** from game provided to you. Be careful not to modify this method. Finally, call the **Play** method and the game may start !

Clue

A clue is hidden somewhere...

There is nothing more deceptive than an obvious fact