10/01/2021

Design Pattern and Software Development Process Project

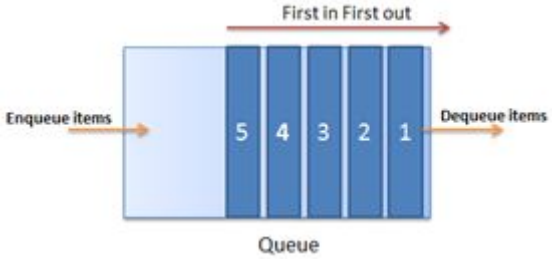
MAIER Flora
LASCAR Sarah
JERUSALMI Kevin

Exercise 1 – CustomQueue – Generics

1. Introduction

This exercise requires the implementation of a Queue which is used to represent a first-in, first out (FIFO) collection of objects. It is used when you need first-in, first-out access of items.

Two difficulties have been added: the queue must be generic, and a queue must be usable with a for each.



2. Design Hypothesis

To meet the requirements of the exercise, we are going to implement the Custom Queue in the same way we did in TD1 (with linked lists). To be precise, we must create a class for each node. This generic class defines a node by data of type T, as well as by the next node.

```
class Node<T> : IEquatable<Node<T>>
{
    //Attributes
    10 références
    public T data {get; set;}
    13 références
    public Node<T> next {get; set;}

    //Constructor
    6 références
    public Node(T val)
    {
        data = val;
        next = null;
    }
```

In addition, the implementation of the IEquatable interface allows the comparison of two nodes.

```
public override int GetHashCode() //The hashcode is usefull to have a comparison between 2 nodes
{
    return data.GetHashCode();
}
0 références
public bool Equals(Node<T> other) // in order to compare 2 nodes
{
    return this.GetHashCode() == other.GetHashCode();
}
```

Next, we create the generic CustomQueue class. This class defines a Queue by a head, a tail, and a size, initialized to 1 on creation and then modified according to the use of the Queue.

```
3 références
class CustomQueue<T> : IEnumerable
{
    //Attributes
    14 références
    public Node<T> head { get; set; }
    4 références
    public Node<T> tail { get; set; }
    3 références
    public int size { get; set; }

    //Constructor
    1 référence
    public CustomQueue(Node<T> node)
    {
        head = tail = node;
        size = 1;
    }
```

Here are the functions implemented allowing the use of the customQueue:

- The Enqueue function allows you to add a Node to the Queue
- The Dequeue function allows you to exit the Node by the tail
- The printQueue function displays the CustomQueue
- The getNode function allows you to find a Node by index
- The setNode function is used to modify a Node by index

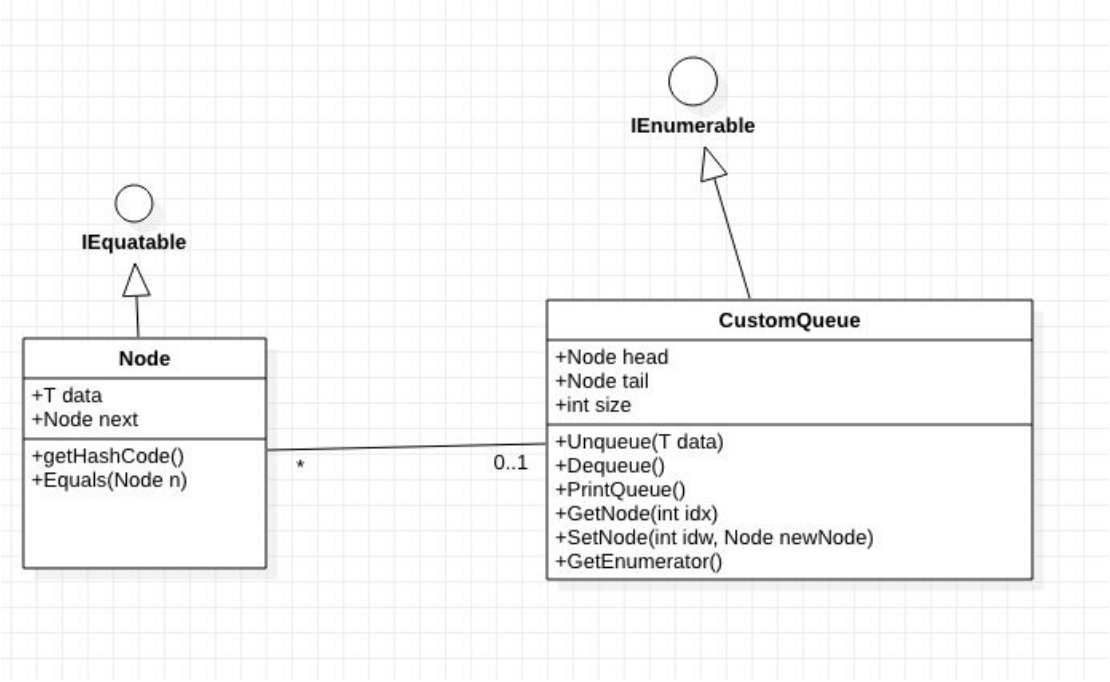
Finally, in order to make it possible to use for each on the CustomQueue, we need to implement the IEnumerable interface.

For this we have used the following functions:

```
//IEnumerable
2 références
public IEnumerator<T> GetEnumerator()
{
    Node<T> courant = head;
    while (courant != null)
    {
        yield return courant.data;
        courant = courant.next;
    }
}

0 références
IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
```

3. UML diagrams

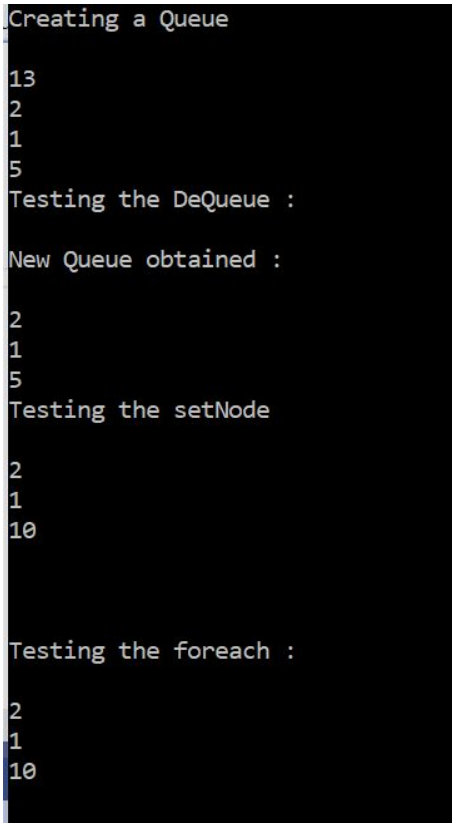


4. Test cases

Here are the cases we tested on our CustomQueue.

First, we created a CustomQueue that we printed with the printQueue function. Then, we tried the Dequeue function, which removes the first element from the list (the 13th). After a second print, we observe that the 1 "is no longer part of the customQueue. We then tried the setnode by changing the 5 to 10, and it worked fine.

Finally, we tried the IEnumerable interface with the forEach, which displayed our CustomQueue.



Exercise 2 – MapReduce – Design patterns, Threads & IPC

1. Introduction

This exercise requires the use of the MapReducing principle on simple data such as a character string.

In order to understand the stakes of this problem, we must first understand what MapReducing is. MapReduce is a concept invented by Google and used by companies like Facebook and Amazon. The goal is to process very large data while being as efficient as possible.

MapReduce consists of two functions map () and reduce ().

The map step splits a problem into several sub-problems, while the reduce step brings together the results.

Here is a diagram explaining the MapReduce programming model:

2. Design Hypothesis

First of all, we need to process the first step: the **splitting part**.

To do this, we start by programming the **textToBlocks** function, which aims to transform our string into several enumerable substrings.This function will be used in parallel in the mapWords function, which will take care of the mapping.

This function takes as parameter (INPUT) the string to study. Then, it cuts its string into several parts according to the following rules:

- Each substring contains a maximum of 10 characters
- The character string is splitted with a space as a separator

The yield allows to return the substrings one by one.Thus, we are able to make the text blocks iterable. We can go to the **mapping** step.

To do this, we implement the **blocksToWords function.** Thanks to Parallele.ForEach we are able to launch the mapping process as soon as a block is finished, that allows several blocks to be mapped at the same time.

The **blocksToWords** function will separate the words in each block by removing spaces and punctuation marks. We retrieve each word from each block and add them to wordCollection. The interest of wordCollection which is a Blocking collection object is that the words of each block can be added simultaneously.

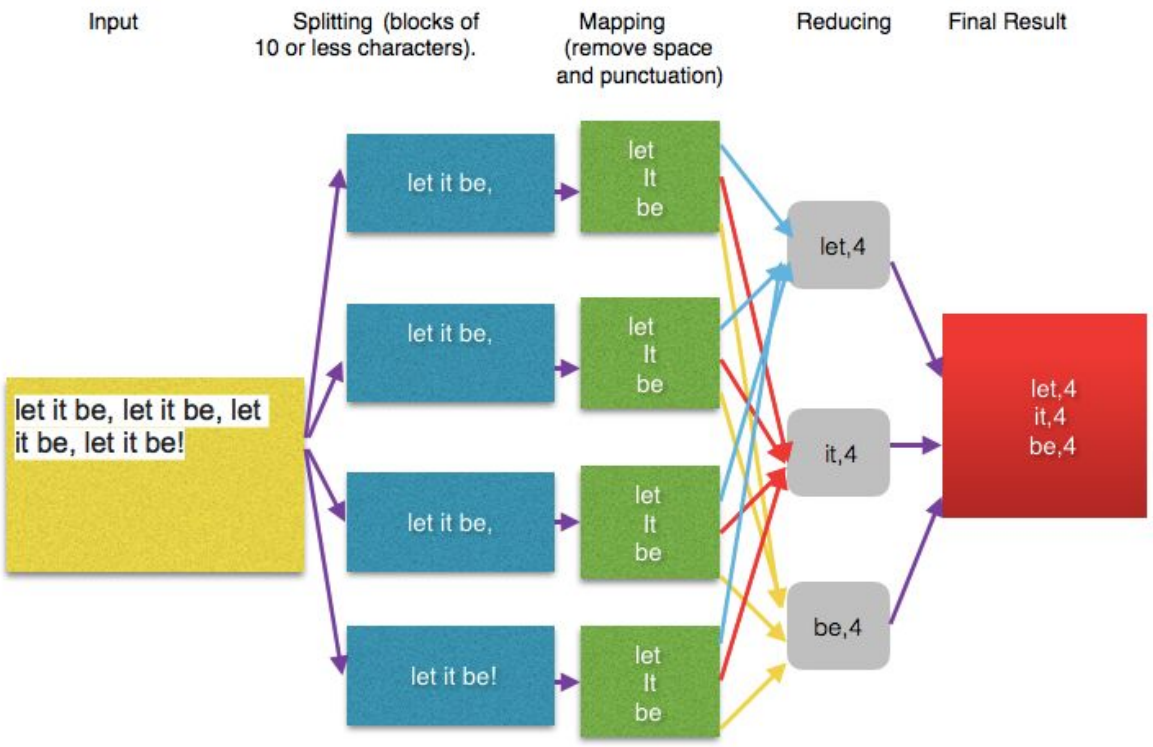
Once the processus is over, wordCollection is complete and we cannot add any element.

Now, we can go to the **reducing** step with the **countingWords** method. This uses a ConcurrentDictionnary<string,int> **wordDico** in order to add a word as a key, with a value of 1 or more if the word is already in wordDico. In this case, the value increments by 1. As we use Parallel.ForEach method this dictionary allows us to use different threads at the same time, hence we can add several words simultaneously.

The **final result** is given thanks to the **mapReduce** function.

3. Diagram

We display the diagram of the MapReduce process linked to our code.



4. Test cases

We entered the following input data :
Let it be, let it be, let it be, let it be! There will be an answer, let it be!

If we count by hand the number of words, we are waiting for this result :

- let : 5
- it : 5
- be : 6
- there : 1
- will : 1
- an : 1
- answer : 1

We notice that our algorithm has to be careful about counting the words, even with a capital letter, and taking off the punctuation marks.

The algorithm has to show different steps we made : First, it has to show the splitting and mapping step, and then the counting (reducing) one, before it shows the result.

Here is what in obtained by running it :

```
Splitting the file text in blocks of 10 or less characters
BLOCK : let it be,
WORD : let
WORD : it
WORD : be
BLOCK : let it be,
WORD : let
WORD : it
WORD : be
BLOCK : let it be,
WORD : let
WORD : it
WORD : be
BLOCK : let it be!
BLOCK : there will
WORD : let
WORD : it
WORD : be
WORD : there
WORD : will
BLOCK : be an
BLOCK : answer,
WORD : be
WORD : an
WORD : answer
BLOCK : let it
BLOCK : be!
WORD : let
WORD : it
WORD : be
```

First, the program executed the split from text to blocks, and from blocks to words. As we see, the program did these 2 actions well in parallel because some words had been splitted before all the blocks had been created. By splitting blocks into words, we notice that the punctuation signs have been deleted and the capital letters turned into regular ones.

```
Reducing method, we count the iterations of words and we store it in wordDico :
COUNTING : let
COUNTING : it
COUNTING : be
STORING : be
COUNTING : it
STORING : let
STORING : it
COUNTING : it
STORING : it
COUNTING : be
STORING : be
COUNTING : let
STORING : let
COUNTING : it
STORING : it
STORING : it
COUNTING : there
STORING : there
COUNTING : be
STORING : be
COUNTING : be
COUNTING : let
STORING : let
COUNTING : an
STORING : an
COUNTING : let
STORING : let
COUNTING : it
COUNTING : be
STORING : be
COUNTING : will
STORING : will
STORING : it
STORING : be
COUNTING : answer
STORING : answer
COUNTING : let
STORING : let
COUNTING : be
STORING : be
```

Following the result shown by the program, we see that the second step is executed as well. The goal was to count each word of the text obtained before and to store the number of iterations in a ConcurrentDictionnary. We can again notice here that the program did the counting action and the storing one well in parallel, since on the screen some words are counted even before others finished to be stored.

Then we finally obtained what we wanted :

```
Final results :

will : 1
it : 5
an : 1
there : 1
let : 5
answer : 1
be : 6
```

Exercise 3 - A Monopoly game (8 points) - Design patterns

1. Introduction

To code the Monopoly rules given by the statement, the difficulty lies in changing the behavior of the players.

Indeed, the players move on the board according to the result of the dice, but in certain situations, like 3 consecutive doubles or when they fall on space 30, they change state, i.e. they go to prison.
In this situation they behave differently, they are stuck in square 10, and they will act differently with regard to the roll of the dice : they can only be released if they obtain a double or if three turns have been passed.

2. Design Hypothesis

In order to be able to deal with this problem, we use the “State Pattern”: it is used when it is desired to be able to change the behavior of an object when its state changes, without changing its instance.
Thus, the player can have several different states: normal, jail1 (1st prison turn), jail2 (2nd turn) or jail3 (3rd turn) in which his behavior when facing the dice will change.

Our state pattern offers two main classes:

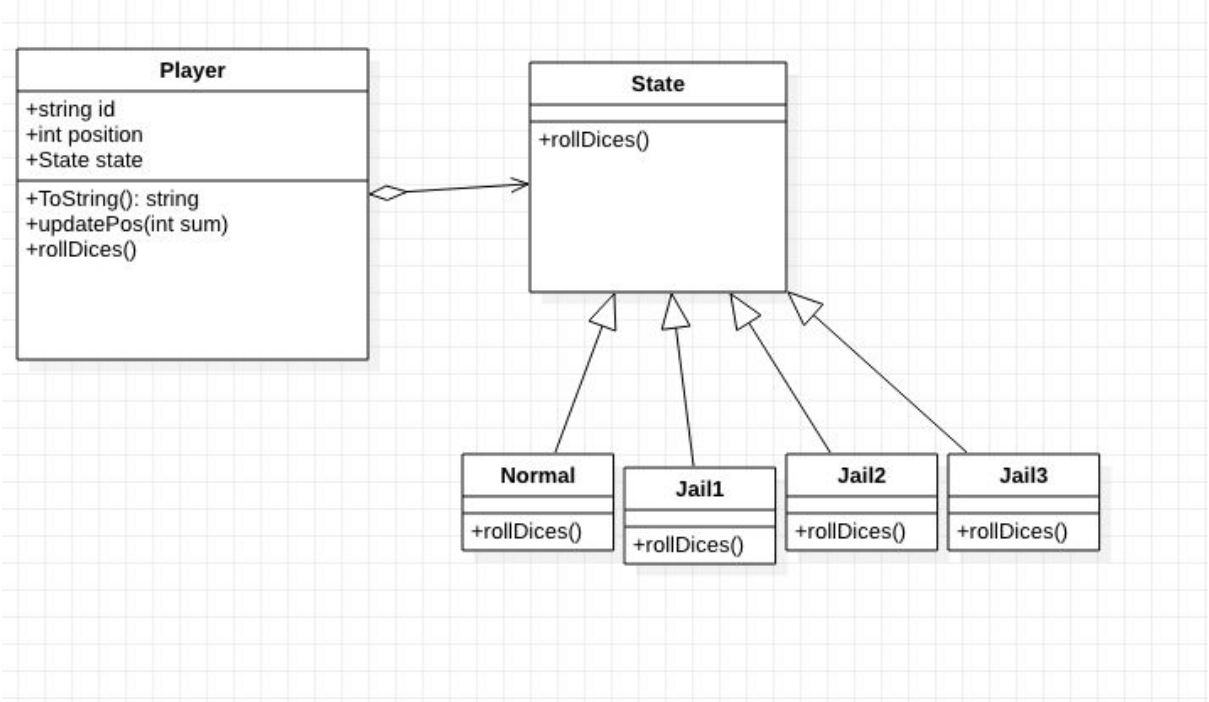
- The State interface, which is implemented by all the states that the player can have.
- The Player class, which implements the State interface but also implements a state as an attribute to interact with the different state and update the state of the player.

The State interface allows, through its specializations, to dynamically create and manage the behaviors that the players will need. The Player class allows the players to update the behavior to be implemented and so that the Player has a different behaviour when facing the dice.

The Player class keeps a link to the State interface. Each behavior is a specialization of State. When the State interface instance changes, the player’s behavior changes. This pattern allows the Player to change behavior dynamically without changing his instance. For example, when a Player goes to jail, his behavior changes when facing dice, but he does not change his instance.

3. UML diagrams

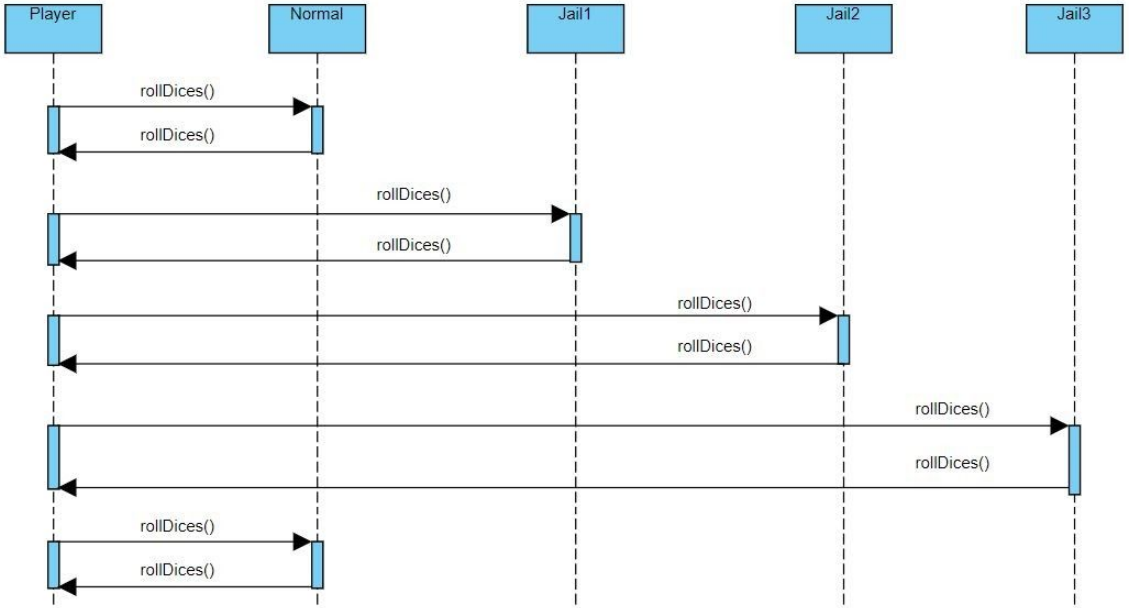
a. Class diagram of the solution



The player class implements the interface State and also need it as an attribute.
The different state classes (Normal, Jail1…)implement the interface State. They are all the state that the player can have.

b. Sequence diagrams

Here is the sequence diagram corresponding to the case where the player stayed 3 turns in prison.
The first time he played, the player went on the Go-To-Jail case or did 3 doubles in a row, so he was in the Normal state and went to the Jail1 state due to the rollDices() function. In our chosen case the player will never do a double in prison, so everytime he plays, the information given by the dice makes him go to the next level of prison, until in Jail3 state. In this last state, whatever the result of the dice, the player is going back to the Normal one and will move on regarding the dice score.



4. Test cases

The cases we want to test only depends on the dice results through the game. To test all the cases we would like to see, we just have to change the dice results.
To easily see that the player is indeed in the state wanted, we added “NORMAL STATE”, “JAIL1 STATE”, “JAIL2 STATE” and “JAIL3 STATE” before messages we added to the rollDices() method in each state class. At the beginning of the fame, each player is at the position 1 and in the normal state.
At the end of each turn, we check the position of the player that just played.

- **Case 1 : Normal → Normal**

Situation : If the player is on a normal case, and the dice shows a regular result, the player stays in a normal state and only his position on the board is actualised. If the player does one double, his position is actualised and he can play again

```
-------------------------MONOPOLY ----------------------
Let's play ! The players are : Sarah, Flora, and Kevin

-------------------------TURN 1 ----------------------

Sarah's turn !
NORMAL STATE : We roll the dices ! We are for now just in a normal case.
2 2
Double dices ! Replay
5 4
The player Sarah is at the position 13

Flora's turn !
NORMAL STATE : We roll the dices ! We are for now just in a normal case.
4 1
The player Flora is at the position 5
```

Conclusion : Sarah got a double, and well replayed, his position actualized all the dice results and not just the last one.
Flora had a regular result, so she just moved on the board.

- **Case 2 : Normal → Normal**

Situation : If the player is in a Normal state and reaches Case 10, he simply visits the prison and stays in the Normal state.
How ? To check it we on purpose setted the dice at 4 and 6.

```
-------------------------TURN 1 ----------------------

Sarah's turn !
NORMAL STATE : We roll the dices ! We are for now just in a normal case.
6 4
You simply visit the jail !
The player Sarah is at the position 10
```

```
-------------------------TURN 2 ----------------------

Sarah's turn !
NORMAL STATE : We roll the dices ! We are for now just in a normal case.
6 4
The player Sarah is at the position 20
```

Conclusion : As we see, the player simply visits the prison and then stays in the Normal state for the next turn.

- **Case 3 : Normal → Jail1**

Situation : If the player is on a normal case , and then moves to case 30, he has to go to jail, on case 10 and goes in the Jail1 state.
How ? Keeping the same game as the last test case, Sarah is going to move case 30 at the turn 3, because for now she is at case 20.

```
-------------------------TURN 3 ----------------------

Sarah's turn !
NORMAL STATE : We roll the dices ! We are for now just in a normal case.
6 4
You are on the Go-To-Jail case !
You are in jail ! :(
The player Sarah is in jail
```

```
-------------------------TURN 4 ----------------------

Sarah's turn !
JAIL1 STATE : You are in jail ! Hope you are going to do a double !
6 4
The player Sarah is in jail
```

Conclusion : Sarah went to prison and at Turn 4 is well in the Jail1 state.

- **Case 4 : Normal → Jail1 → Jail2 → Jail3 → Normal**

Situation : If the player is in Jail1 state and did not do a double, he stays in prison and goes to Jail2 state. Next turn, if in Jail2 state he still did not do a double, he goes in Jail3 state. Finally, whatever the dice's result, he goes out of prison since he stayed 3 turns in it.
How ? We are keeping the game as before, with the dice always showing 4 and 6 so the player never does a double.

```
-------------------------TURN 5 ----------------------

Sarah's turn !
JAIL2 STATE : You are still in jail ! Hope you are going to do a double !
6 4
The player Sarah is in jail
```

```
--------------------------TURN 6 -----------------------

Sarah's turn !
JAIL3 STATE : You stayed 3 times in jail and now you are free !
6 4
The player Sarah is at the position 20
```

```
-------------------------TURN 7 ----------------------

Sarah's turn !
NORMAL STATE : We roll the dices ! We are for now just in a normal case.
6 4
```

Conclusion : Sarah never did doubles in prison and also went through all the jail states, until Jail3 where she went back to normal and moved on the board starting from case 10, that's why she went Turn 6 at case 20. We noticed that at Turn 7 she is indeed well in the Normal state.

- ## Case 5 : Normal → Jail1

Situation : If the player did 3 doubles in a row, the player goes to Jail, in case 10 and moves to the Jail1 state.
How ? We setted the dice at double 6 everytime the player is rolling them, so he will obviously do 3 doubles in a row.

```
-------------------------TURN 1 ----------------------

Sarah's turn !
NORMAL STATE : We roll the dices ! We are for now just in a normal case.
6 6
Double dices ! Replay
6 6
Double dices ! Replay
6 6
The player Sarah is in jail
```

```
-------------------------TURN 2 ----------------------

Sarah's turn !
JAIL1 STATE : You are in jail ! Hope you are going to do a double !
6 6
You did a double in jail ! You are free !
The player Sarah is at the position 22
```

Conclusion : Sarah is well in Jail after 3 doubles in a row.

- **Case 6 : Jail1 → Normal**

Situation : If the player is in Jail1 state, if he does a double he finally goes out of prison and moves on the board starting from the prison case, case 10.
How ? We already saw it on the last example, Sarah was in jail1 state and did a double.

```
-------------------------TURN 3 ----------------------

Sarah's turn !
NORMAL STATE : We roll the dices ! We are for now just in a normal case.
6 6
```

Conclusion : After Sarah did a double in Jail1, she is well in Normal state.

- **Case 7 : Jail2 → Normal**

Situation : Same as case 6, but for Jail2 state.
How ? We setted the dice for a random result between 1 and 2, in order to have a high possibility of doubles but still the possibility to stay in Jail and go in Jail2, before going out by a double.

```
-------------------------TURN 6 ----------------------

Sarah's turn !
JAIL2 STATE : You are still in jail ! Hope you are going to do a double !
2 2
You did a double in jail ! You are free !
The player Sarah is at the position 14
```

```
-------------------------TURN 7 ----------------------

Sarah's turn !
NORMAL STATE : We roll the dices ! We are for now just in a normal case.
2 1
The player Sarah is at the position 17
```

Conclusion : Sarah went out of Jail2 state thanks to a double at Turn 6, and is in a Normal state at Turn 7.