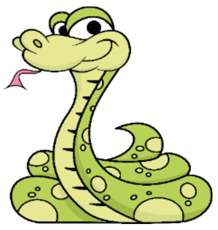
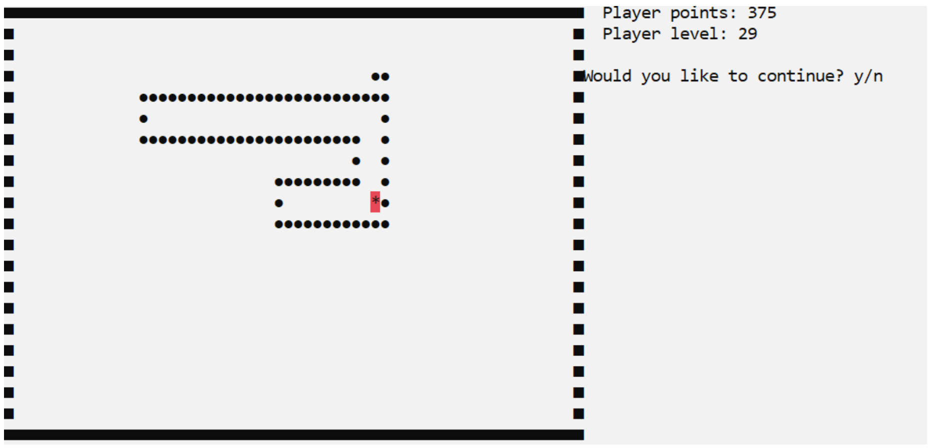
Snake

```
Player points: 375
Player level: 29

Would you like to continue? y/n
```

Overview

In this workshop, you need to build a snake game on your own, following the basic principles of OOP.

Setup

You are provided with a **skeleton**, which contains the following items:

- **StartUp** class – your program entry point

- **Core** folder – the main program functionality

- **Enums** folder – data about directions for the snake

- **GameObjects** folder – holding the main objects for our game

- **Utilities** folder – already written (contains information about your **ConsoleWindow**)

Task 1: Structure

Game objects

In the **GameObjects** folder create the following classes:

Point

The **Point** class contains information about the **2D space** where all the objects exist. It has two properties, which indicate the **horizontal and vertical position** of the object. You can give them the names **LeftX** and **TopY** and both are **integers**.

```
public int LeftX { get; set; }

public int TopY { get; set; }
```

We are almost done with this class, but lastly, we need to create **two methods** for **drawing** on the **console**, accepting a different number of arguments. The first **Draw** method accepts a **symbol** and calls the **Console.SetCursorPosition** method with the current **point coordinates** and writes the current **symbol** on the console. The second **Draw** method accepts **LeftX**, **TopY** and **symbol** as arguments.

Your code should look like this:

```
public void Draw(char symbol)
{
    Console.SetCursorPosition(this.LeftX, this.TopY);
    Console.Write(symbol);
}

public void Draw(int leftX, int topY, char symbol)
{
    Console.SetCursorPosition(leftX, topY);
    Console.Write(symbol);
}
```

Constructor

A **Point** accepts the following values upon initialization:

int leftX, int topY

Wall

The **Wall** class inherits **Point** and its parent constructor. It must have methods (which are inaccessible from outside the class) for setting the **horizontal** and **vertical lines**. The **SetHorizontalLine** method accepts an argument of type **int** for the **topY** position. In its body, it iterates from zero to the current **LeftX** position.

The **Draw** method is called inside of the loop with the following parameters:

int leftX, int topY, char wallSymbol

The **wallSymbol** is a **constant value** for the Wall class and uses Unicode code point **\u25A0** (represents a square symbol for the right and left borders of the wall). It looks like this: ■

```
private const char wallSymbol = '\u25A0';

private void SetHorizontalLine(int topY)
{
    for (int leftX = 0; leftX < this.LeftX; leftX++)
    {
        this.Draw(leftX, topY, wallSymbol);
    }
}
```

The **SetVerticalLine** method has the same logic, but accepts **leftX** as parameter and iterates through **topY**

```
private void SetVerticalLine(int leftX)
{
    for (int topY = 0; topY < this.TopY; topY++)
    {
        this.Draw(leftX, topY, wallSymbol);
    }
}
```

Constructor

Calls **InitializeWallBorders** method, where the **horizontal** and **vertical** lines for each **four** sides of the wall are set:

```
private void InitializeWallBordes()
{
    SetHorizontalLine(0);
    SetHorizontalLine(this.TopY);

    SetVerticalLine(0);
    SetVerticalLine(this.LeftX - 1);
}
```

Food

Food is an **abstract class** which inherits **Point** and implements all the logic for the types of food our game will have. We must have a property for the points given by each type of food.

```
public int FoodPoints { get; private set; }
```

Declare a **SetRandomPosition** method which will set the position of the current food at a random place inside the boundaries of our wall. It accepts a **Queue** of type **Point** as its only argument, which contains all the snake parts at this moment. Our method should generate a random position for the **LeftX** and **TopY** points.

We must also check if the position of our random points is **equal to** the snake's position and if it is, we set **new** random positions. If we want our type of food to be colorful, we can set the **Console BackgroundColor** property to choose a color for our food. Finally, we call the **Draw** method with the correct food symbol for the type of food (we will see the different symbols when we implement the types of food)

```
private char foodSymbol;

public void SetRandomPosition(Queue<Point> snakeElements)
{
    this.LeftX = random.Next(2, wall.LeftX - 2);
    this.TopY = random.Next(2, wall.TopY - 2);

    bool isPointOfSnake = snakeElements
        .Any(x => x.LeftX == this.LeftX && x.TopY == this.TopY);

    while (isPointOfSnake)
    {
        this.LeftX = random.Next(2, wall.LeftX - 2);
        this.TopY = random.Next(2, wall.TopY - 2);

        isPointOfSnake = snakeElements
            .Any(x => x.LeftX == this.LeftX && x.TopY == this.TopY);
    }

    Console.BackgroundColor = ConsoleColor.Red;
    this.Draw(foodSymbol);
    Console.BackgroundColor = ConsoleColor.White;
}
```

We must know if the food is in front of the snake's head and if it, is the snake **eats** it (implement later). Create a **IsFoodPoint** bool method which accepts the current **snake** position. Compare the snake position with the current food position and return the result of the comparison.

```
public bool IsFoodPoint(Point snake)
{
    return snake.TopY == this.TopY &&
           snake.LeftX == this.LeftX;
}
```

Constructor

Your constructor should look like this:

```
private Wall wall;
private Random random;
private char foodSymbol;

protected Food(Wall wall, char foodSymbol, int points)
    : base(wall.LeftX, wall.TopY)
{
    this.wall = wall;
    this.FoodPoints = points;
    this.foodSymbol = foodSymbol;
    this.random = new Random();
}
```

Child Classes

There are several concrete types of **food**:

- **FoodAsterisk**
- **FoodDollar**
- **FoodHash**

FoodAsterisk implements **Food** and its constructor. It has two constant values. One for the food symbol ("*") and one for the points it gives to the snake - **1**

```
private const char foodSymbol = '*';
private const int points = 1;

public FoodAsterisk(Wall wall)
    : base(wall, foodSymbol, points)
{
}
```

Implement the same logic for the other types of **Food**. The **FoodDollar** has "**$**" symbol and gives **2** points. The **FoodHash** has "**#**" symbol and gives **3** points.

Snake

The **Snake** class is a class, which is used to draw, design, control movement and define Snake behavior. This class has the most complex logic. Here we are going to implement methods for the movement of the snake and the types of food she eats. Let's create our snake. Create a field, which holds all point positions of the snake parts. Declare a **CreateSnake** method which iterates from **1** to **6** (this is our snake's starting length) and each time, adds a new Point in its queue of snake parts.

```
private void CreateSnake()
{
    for (int topY = 1; topY <= 6; topY++)
    {
        this.snakeElements.Enqueue(new Point(2, topY));
    }
}
```

At this time, we don't have any food which our snake can eat. Create a **GetFoods** method which will initialize all the types of food we already have.

```
private void GetFoods()
{
    this.food[0] = new FoodHash(this.wall);
    this.food[1] = new FoodDollar(this.wall);
    this.food[2] = new FoodAsterisk(this.wall);
}
```

For now, our snake can't move. Let's change that. Create a **bool** method **IsMoving**, which takes a Point as parameter, representing the current **direction**. To move the snake, we must first find its head. This happens when we take the **last element** of our queue of snake parts. We have to check which the next position of the snake is. That's why we are going to implement the **GetNextPoint** method, which accepts a **point direction** and the snake's head's position. All this method does is sum the snake's head's **leftX** point and **leftX** direction and write the result of the operation in our **Snake nextX** field. The same logic is inside our Snake's **nextTopY** field.

```
private void GetNextPoint(Point direction, Point snakeHead)
{
    this.nextLeftX = snakeHead.LeftX + direction.LeftX;
    this.nextTopY = snakeHead.TopY + direction.TopY;
}
```

After that, in our **IsMoving** method, we have to check if our snake has bitten herself. If she has, our **IsMoving** method must return **false**. Now we need to check if the snake has stepped on one of the walls' borders and if it has, the snake can't move anymore. To do this we have to implement a new method in the **Wall** class **IsPointOfWall**. This method must accept a **Point**, which contains the coordinates of the snake and check if they are on the border of the wall:

```
public bool IsPointOfWall(Point snake)
{
    return snake.TopY == 0 || snake.LeftX == 0 ||
        snake.LeftX == this.LeftX  -1 || snake.TopY == this.TopY;
}
```

For now, our **IsMoving** method should look like this:

```
public bool IsMoving(Point direction)
{
    Point currentSnakeHead = this.snakeElements.Last();

    GetNextPoint(direction, currentSnakeHead);

    bool isPointOfSnake = this.snakeElements
                .Any(x => x.LeftX == nextLeftX && x.TopY == nextTopY);

    if (isPointOfSnake)
    {
        return false;
    }

    Point snakeNewHead = new Point(this.nextLeftX, this.nextTopY);

    if (this.wall.IsPointOfWall(snakeNewHead))
    {
        return false;
    }
```

We have to make our snake move without leaving dots behind on the console. To do that, we must create a new point for the next position of the head and add it to our snake parts. After that, we should draw the snake symbol.

```
private const char snakeSymbol = '\u25CF';

this.snakeElements.Enqueue(snakeNewHead);
snakeNewHead.Draw(snakeSymbol);
```

We are almost done with this class, but before that, we need to check for food at the position of the new head. If there is food, the snake must eat it.

```
if (food[foodIndex].IsFoodPoint(snakeNewHead))
{
    this.Eat(direction, currentSnakeHead);
}
```

Create a new method with the name **Eat**, which accepts two points – one for the **direction** and one for the **current snake head**. Because we have different points for the different foods, we have to check how much score the current food **gives**. After that, we have to increase the length of the snake with the count of the score. Finally, we have to create a **new food** at a new random position.

```
private void Eat(Point direction, Point currentSnakeHead)
{
    int length = food[foodIndex].FoodPoints;

    for (int i = 0; i < length; i++)
    {
        this.snakeElements.Enqueue(new Point(this.nextLeftX, this.nextTopY));
        GetNextPoint(direction, currentSnakeHead);
    }

    this.foodIndex = this.RandomFoodNumber;
    this.food[foodIndex].SetRandomPosition(this.snakeElements);
}
```

As you can see on the image above, we have **RandomFoodNumber** property which we haven't implement . Think about how to get a new random position for the food each time this property is called.

```
private int RandomFoodNumber =>
                    new Random().Next(0, this.food.Length);
```

Finally, we have to take the position of the snake tail and we have to remove this point with drawing an empty space on the console. If we have managed to complete all the functionality in this method, we must return true.

```
Point snakeTail = this.snakeElements.Dequeue();
snakeTail.Draw(emptySpace);

return true;
```

Constructor

If you have managed to implement this class, your constructor should look like this:

```
public Snake(Wall wall)
{
    this.wall = wall;
    this.snakeElements = new Queue<Point>();
    this.food = new Food[3];
    this.foodIndex = RandomFoodNumber;
    this.GetFoods();
    this.CreateSnake();
}
```

Engine

Our engine class will be responsible for the user interface. This class will take care of all the buttons clicked by the user and visualize all the logic we have already written. It will have only one public method **Run()** which will do the main logic for the application.

First, we need to create a collection of points where we will collect our possible directions (up, down, left, right). To fill up our points collection we need to create a method which will initialize four points with different **leftX** and **topY** positions. For the down direction we need to create point with **leftX** equal to **1** and **topY** equal to **0**. For the up we need **-1** for **leftX** and **0** for **topY**. Think about the other two directions 😊

```
private void CreateDirections()
{
    this.pointsOfDirection[0] = new Point(1, 0);
    this.pointsOfDirection[1] = new Point(-1, 0);
    this.pointsOfDirection[2] = new Point(0, 1);
    this.pointsOfDirection[3] = new Point(0, -1);
}
```

Now let’s implement a **GetNextDirection** method which will be responsible for the buttons the user clicks. To do that first we need to know which part of the keyboard the user clicks, and this happens by calling the class Console and its **ReadKey** method. Next, we need to know if the button is **Right**/**Left**/**Up**/**Down** arrow and call the correct type of **enum** in our **Direction** enumeration.

```
private void GetNextDirection()
{
    ConsoleKeyInfo userInput = Console.ReadKey();

    if (userInput.Key == ConsoleKey.LeftArrow)
    {
        if (direction != Direction.Right)
        {
            direction = Direction.Left;
        }
    }
    else if (userInput.Key == ConsoleKey.RightArrow)
    {
        if (direction != Direction.Left)
        {
            direction = Direction.Right;
        }
    }
    else if (userInput.Key == ConsoleKey.UpArrow)...
    else if (userInput.Key == ConsoleKey.DownArrow)...
}
```

Use the same logic for the other arrows. Finally, in this method set the cursor to invisible.

```
Console.CursorVisible = false;
```

Let’s check is our snake moving. To do that we need to have private field of type Snake which will represent our console snake and we can call the **IsMoving** method with the correct direction.

```
bool isMoving = snake.IsMoving(this.pointsOfDirection[(int)direction]);
```

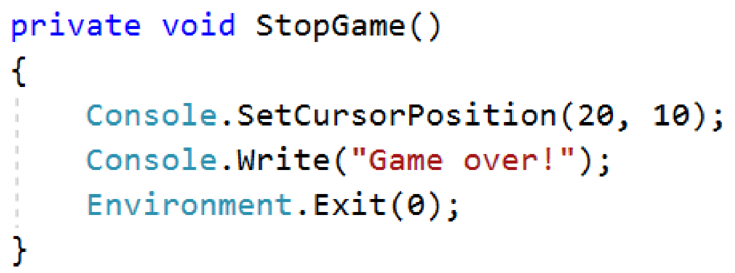
If our snake is not moving, she might be dead, and we must ask the user for restart. Create method **AskUserForRestart**.

```
if (!isMoving)
{
    AskUserForRestart();
}
```

This method will print a message to the console "Would you like to continue? y/n". If the user press y on the keyboard we must clear the console and call our **StartUp** Main method again.

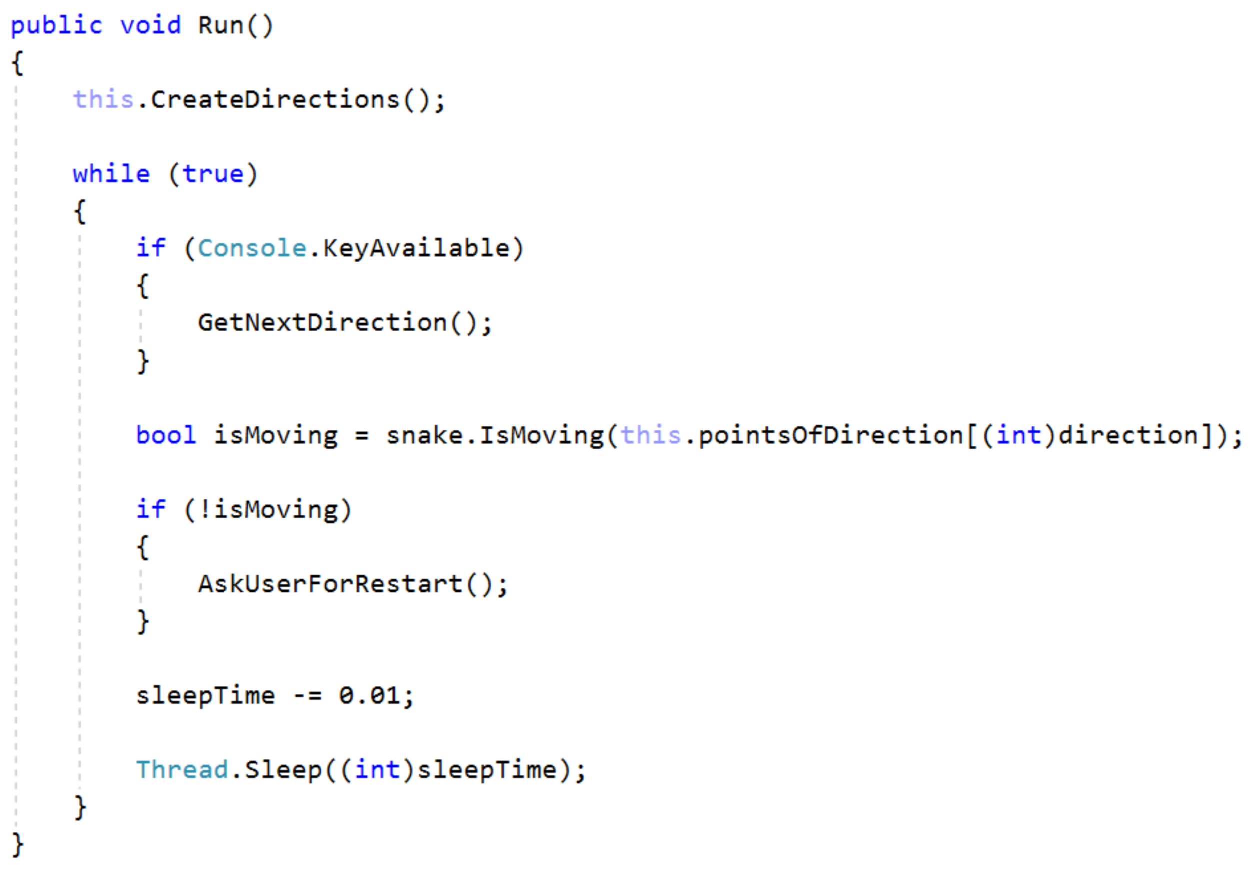
```
private void AskUserForRestart()
{
    int leftX = this.wall.LeftX + 1;
    int topY = 3;

    Console.SetCursorPosition(leftX, topY);
    Console.Write("Would you like to continue? y/n");

    string input = Console.ReadLine();

    if (input == "y")
    {
        Console.Clear();
        StartUp.Main();
    }
    else
    {
        StopGame();
    }
}
```

If he presses n create a **StopGame** method which writes **GameOver** and exits from the console.



```
private void StopGame()
{
    Console.SetCursorPosition(20, 10);
    Console.Write("Game over!");
    Environment.Exit(0);
}
```

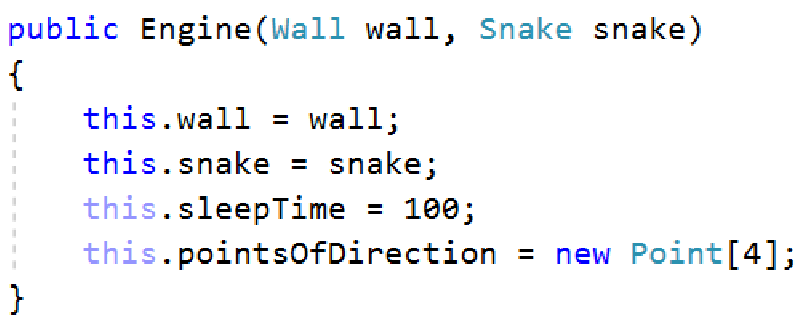
Your **Run()** method should already look like this:



```
public void Run()
{
    this.CreateDirections();

    while (true)
    {
        if (Console.KeyAvailable)
        {
            GetNextDirection();
        }

        bool isMoving = snake.IsMoving(this.pointsOfDirection[(int)direction]);

        if (!isMoving)
        {
            AskUserForRestart();
        }

        sleepTime -= 0.01;

        Thread.Sleep((int)sleepTime);
    }
}
```

Constructor

Your engine constructor will take Wall and Snake as parameters. It will also initialize our for points and set a default sleep time of one hundred milliseconds.



```
public Engine(Wall wall, Snake snake)
{
    this.wall = wall;
    this.snake = snake;
    this.sleepTime = 100;
    this.pointsOfDirection = new Point[4];
}
```

StartUp

Our **StartUp** class should only initialize our **wall**, **snake** and call the **Engine Run** method!

```
ConsoleWindow.CustomizeConsole();

Wall wall = new Wall(60, 20);
Snake snake = new Snake(wall);

Engine engine = new Engine(wall, snake);
engine.Run();
```

As you can see on the image of the game on the right side you have game statistic. You can think of a way to show on the console these game stats.

Start the game and have a nice play 😊!