Object-Oriented Programming – Practical Exam

Problem 1 – HTML Rendering Engine

A software company has designed a very simple **HTML rendering engine**. It supports only two types of elements: HTML elements and HTML tables. **HTML elements** have name and text content and may optionally have a list of child nested HTML elements. **HTML tables** are special kind of HTML elements that have fixed size (rows and columns) and hold rows * columns cells which are HTML elements.

Design the Class Hierarchy

Your **first task** is to **design an object-oriented class hierarchy** to model the HTML rendering engine **using the best practices for object-oriented design (OOD) and object-oriented programming (OOP)**. Avoid duplicated code through abstraction, inheritance, and polymorphism and encapsulate correctly all fields.

You are given few C# **interfaces** that you should **obligatory** implement and use as a basis of your code:

```
namespace HTMLRenderer
{
  public interface IElement
  {
    string Name { get; }
    string TextContent { get; set; }
    IEnumerable<IElement> ChildElements { get; }
    void AddElement(IElement element);
    void Render(StringBuilder output);
    string ToString();
  }

  public interface ITable : IElement
  {
    int Rows { get; }
    int Cols { get; }
    IElement this[int row, int col] { get; set; }
  }

  public interface IElementFactory
  {
    IElement CreateElement(string name);
    IElement CreateElement(string name, string content);
    ITable CreateTable(int rows, int cols);
  }

  public class HTMLElementFactory : IElementFactory
  {
    // TODO: implement the IElementFactory interface
  }
}
```

All your HTML elements should implement **`IElement`**. All your HTML table elements should implement **`ITable`**. HTML tables have no text content and their name is "**`table`**". HTML tables cannot have child elements.

Your **`IElement.Render(StringBuilder output)`** implementation should render the element as follows:

```
<(name)>(text_content)(child_content)</(name)>
```

If ***(text_content)*** is missing, it is not rendered. The ***(child_content)*** is rendered by rendering all child elements in the order of their addition. If the element has no child elements, the ***(child_content)*** is empty.

Your **`ITable.Render(StringBuilder output)`** implementation should render the table as follows:

```
<table><tr><td>(cell_0_0)</td><td>(cell_0_1)</td>…</tr><tr><td>(cell_1_0)</td><td>(cell_1_1
)</td>…</tr>…</table>
```

For each row its content is rendered enclosed between the **`<tr>`** and **`</tr>`** tags. For each column inside a row its element content is rendered enclosed between the **`<td>`** and **`</td>`** tags.

The **`IElement.ToString()`** method renders the element into a string and returns it as result.

The **`Name`** and **`TextContent`** properties are optional and can be **`null`** (**`null`** values are not rendered). When a **`TextContent`** is rendered, the HTML special characters **`<`**, **`>`** and **`&`** are escaped as **`&lt;`**, **`&gt;`**, **`&amp;`** respectively. When elements are rendered, no spacing or new lines is put between them.

Write a Program to Execute C# Statements Using Your Classes

Your **second task** is to write a program that **executes a standard C# code block** (sequence of C# statements) using your classes and interfaces and prints the results on the console. A sample C# code block is given below:

```
IElementFactory htmlFactory = new HTMLElementFactory();
IElement html = htmlFactory.CreateElement("html");
IElement h1 = htmlFactory.CreateElement("h1", "Welcome!");
html.AddElement(h1);
Console.WriteLine(html);
```

The statements are guaranteed to be **valid C# sequences of statements** that will compile correctly if you implement correctly the interfaces and classes described above. The statements will be no more than **500**. Each statement will be less than **100 characters** long. The statements end with an empty line. The code block will not throw any exceptions at runtime and will not get into an endless loop.

Additional Notes

To simplify your work you are given a C# code block execution engine that compiles and executes a sequence of C# statements read from the console using the classes and interfaces in your project (see the file **`HTMLRenderer-Skeleton.rar`**). Please put all your code directly in the namespace "**`HTMLRenderer`**".

You are only **allowed to write classes**. You are **not allowed to modify the existing interfaces**.

Sample Input

```
IElementFactory htmlFactory = new HTMLElementFactory();
IElement html = htmlFactory.CreateElement("html");
IElement h1 = htmlFactory.CreateElement("h1", "Welcome!");
html.AddElement(h1);
Console.WriteLine(html);
ITable table = htmlFactory.CreateTable(3, 2);
table[0, 0] = htmlFactory.CreateElement("b", "First Name");
table[0, 1] = htmlFactory.CreateElement("b", "Last Name");
table[1, 0] = htmlFactory.CreateElement(null, "Svetlin");
table[1, 1] = htmlFactory.CreateElement(null, "Nakov");
table[2, 0] = htmlFactory.CreateElement(null, "George");
table[2, 1] = htmlFactory.CreateElement(null, "Georgiev");
html.AddElement(table);
```

```
IElement br = htmlFactory.CreateElement("br", null);
html.AddElement(br);
IElement div = htmlFactory.CreateElement("div",
  "(c) Nakov & Joro @ <Telerik Software Academy>");
html.AddElement(div);
Console.WriteLine(html);
(empty line)
```

Sample Output

```
<html><h1>Welcome!</h1></html>
<html><h1>Welcome!</h1><table><tr><td><b>First Name</b></td><td><b>Last
Name</b></td></tr><tr><td>Svetlin</td><td>Nakov</td></tr><tr><td>George</td><td>Georgiev</t
d></tr></table><br></br><div>(c) Nakov &amp; Joro @ &lt;Telerik Software
Academy&gt;</div></html>
```

Problem 2 – Ecosystem Simulation API

You are given an ecosystem simulation API that gives the base for creating organisms, moving organisms and handling encounters between organisms. The API includes an engine, which controls the simulation world and executes commands on the objects within it. You are also given a C# file, which has a Main method and uses the API for processing commands from the input.

There are some simple rules the simulation API supports:

- Objects can be created anywhere
- There are 3 major types of objects – carnivores (meat-eaters), herbivores (plant-eaters) and plants
  - carnivores can eat herbivores and other carnivores
  - herbivores can eat plants
- Objects can go anywhere if they support the "go" command
  - Only one object can move at a time
  - When one object moves, all objects (including the moving one) get "updated" with the time required for the moving object to travel (the time is determined by the Manhattan distance between the start and end positions).
  - When the moved object gets to a position where other objects are located, it "encounters" all of them – it tries to eat them, according to the rules above
- Objects can sleep
- Some objects change their size when eating or sleeping

The following lines will help you better understand the API.

Important Classes and Interfaces

These are the API's interfaces:

- **`IOrganism`** – provides base properties and methods, supported by all organisms – **`IsAlive`**, **`Location`**, **`Size`**
- **`IHerbivore`** – represents a plant-eater and provides a method **`EatPlant(Plant)`**, which is called when the object encounters other objects. Eating a plant could reduce its size or kill it. Note: The proper

way to implement the **`EatPlant`** method is to call the passed plant's **`GetEatenQuantity(int)`**. EatPlant must also **return the eaten quantity**.

- **`ICarnivore`** – represents a meat-eater and provides a method **`TryEatAnimal(Animal)`**, which is called when the object encounters other objects. The carnivore could fail in eating the animal, but if it succeeds, the animal dies. Note: The proper way to implement the **`TryEatAnimal`** method is to call the passed animal's **`GetMeatFromKillQuantity()`**. The TryEatAnimal method must **return the quantity of meat eaten**.

These are the API's classes and structures:

- **`Point`** – **`struct`**, represents a **two-dimensional point with integer coordinates** and provides a **`Parse(string)`** and overloaded **`ToString()`** method, along with a GetManhattanDistance(Point, Point) static method
- **`Organism`** – **`abstract`** implementation of the **`IOrganism`** interface. Has an overloaded **`ToString()`** method
- **`Animal`** – **`abstract`** base for **all animals** (organisms which can move), which can execute "go" and "sleep" commands. Provides the respective **`GoTo(Point)`** and **`Sleep(int)`** commands. Also provides an implementation of the **`GetMeatFromKillQuantity()`** method, which kills the animal and returns its meat value. Also has a **`State`** property, indicating whether the animal is sleeping or awake. Has a **Name** and an overloaded **`ToString()`** method
- **`Plant`** – **`abstract`** base for organisms that never change their position. Has a **`GetEatenQuantity(int)`** method, which takes a "bite size" and reduces the size of the plant by that bite size (if the plant is bigger), or kills the plant and sets its size to 0, if the bite is larger than the plant's size. The method returns the change in size of the plant (i.e. the "eaten" quantity). Has an overloaded **`ToString()`** method
- **`Tree`** and **`Bush`** – plants with a predefined size
- **`Deer`** – animal class, implementation of the **`IHerbivore`** interface. Does not change in any way when eating.
- **`Engine`** – handles commands and executes them on the simulation world, keeps several lists of the objects in it, provides methods for adding objects and removes dead objects after each "go" command

Commands

There are two types of commands the Engine supports:

- Organism birth command – creates an object in the world
    - Format: **birth <object-type-name> <object-parameters…>**
    - Example: **birth tree (0,0)** – creates a tree at the position (0, 0)
    - Example: **birth deer Rudolf (0,0)** – creates a deer with the name Rudolf at position (0,0)
- Object instance command – orders an object to execute a command. If the object executes the command, a string is printed.
    - Format: **<command-name> <object-name> <command-parameters>**
    - Commands:
        - **`sleep <object-name> <time>`**
        - **`go <object-name> <position>`**
    - Example: **sleep Rudolf 10** – puts the animal with the name Rudolf to sleep for 10 time units

- Example: **go Rudolf (5,5)** – sends the animal with the name Rudolf to position (5,5), updates all objects with the elapsed time and checks for possible encounters with other organisms

Here is a list of all commands the Engine supports currently:

- **birth tree <position>** - creates a tree at the specified coordinates
- **birth bush <position>** - creates a bush at the specified coordinates
- **birth deer <name> <coordinates>** - creates a deer with the specified name at the specified coordinates
- **sleep <name> <time> -** puts the animal with the specified name to sleep for the specified time
- **go <name> coordinates** – moves the animal with the specified name to the specified coordinates, updates all objects with the elapsed time (calculated by Manhattan distance from the object position to the object destination) and checks for encounters

The Engine handles all listed commands and **you shouldn't write your own parsing code** for these commands.

Study the aforementioned classes to get a better understanding of how they work and how the `Engine` class handles the listed commands.

Tasks

You are tasked with extending the API by implementing several commands and object types. You are **not allowed to edit any existing class from the original code of the API**. You are **not allowed to edit the Main method**. You **are allowed** to edit the `GetEngineInstance()` method.

- Implement a command to create a `Wolf`. The `Wolf` should be **an animal** and should have a `Size` of 4. The `Wolf` should be able to **eat any animal smaller than or as big as him**, or **any animal which is sleeping.**
  - Format: **birth wolf <name> <position>** - creates a `Wolf` at the specified position with the specified name
  - Example: **birth wolf Joro (1,1)** – creates a Wolf with the name Joro, at coordinates (1, 1)
- Implement a command to create a `Lion`. The `Lion` should be **an animal** and should have a `Size` of 6. The `Lion` should be able to **eat any animal, which is at most twice larger than him (inclusive).** Also, the **`Lion` should grow by 1 with each animal it eats.**
  - Format: **birth lion <name> <position>** - creates a `Lion` at the specified position with the specified name
- Implement a command to create `Grass`. The `Grass` should be **a plant** and should have a `Size` of 2. The `Grass` should **grow by 1 at each time unit** (i.e. by as much as the timeElapsed parameter is in the Update method), if it `IsAlive`**.**
  - Format: **birth grass <position>** - creates grass at the specified position
  - Example: **birth grass (1,2)** – creates grass at coordinates (1,2)
- Implement a command to create a `Boar`. The `Boar` should be **an animal** and should have a `Size` of 4. The `Boar` should be able to **eat any animal, which is smaller than him or as big as him.** The **`Boar` should be able to eat from any plant.** The `Boar` always has a **bite size** of 2**.** When eating from a plant, the `Boar` increases its size by 1.
  - Format: **birth boar <name> <position>** - creates a boar at the specified position, with the specified name
  - Example: **birth boar Gruhcho (7,-3)** – creates a boar at the coordinates (7,-3), with the name Gruhcho

- Implement a command to create a **`Zombie`**. The **`Zombie`** should be an animal and should **not be able to eat**. When a **carnivore** (of the so-far existing) **tries to eat** a **`Zombie`**, it should **always succeed** and **receive 10 meat** from the **`Zombie`**. However, the **`Zombie`** should never die.
    - Format: birth zombie <name> <position> - creates a zombie with the specified name at the specified position
    - Example: birth zombie Joro (0,0) – creates a zombie named Joro at the coordinates (0,0)

Input and Output Data

You should not concern yourself with handling input and output data – the engine does it for you. You should only consider how to implement the required creation commands. See the existing Engine code for hints. Also:

- The names in the commands will always consist of upper and lowercase English letters only. The numbers in the commands will always be 32-bit signed integers (System.Int32).
- There will never be two plants at the same position, nor two organisms with the same names.

Sample Input	Sample Output

`birth bush (0,0)`
`birth wolf WhiteTooth (1,1)`
`go WhiteTooth (0,0)`
`go WhiteTooth (1,1)`
`birth boar Daddy (0,0)`
`go Daddy (0,0)`
`go Daddy (1,1)`
`birth lion Simba (-1,-1)`
`go Simba (1,1)`
`sleep Simba 10`
`birth wolf Mizernik (0,0)`
`go Mizernik (1,1)`
`birth zombie Joro (10,10)`
`go Mizernik (10,10)`
`go Mizernik (10,10)`
`end`
	`Boar Daddy ate 2 from Bush (0,0)`
	`Boar Daddy ate 4 from Wolf WhiteTooth`
	`Wolf WhiteTooth is dead ;(`
	`Lion Simba ate 5 from Boar Daddy`
	`Boar Daddy is dead ;(`
	`Wolf Mizernik ate 7 from Lion Simba`
	`Lion Simba is dead ;(`
	`Wolf Mizernik ate 10 from Zombie Joro`
	`Wolf Mizernik ate 10 from Zombie Joro`