TCSS 143A Winter 2019, Project 3

Given the problem statement below, complete the following:

- ✓ A class hierarchy that meets all requirements listed in this description. In order to receive credit, your code MUST (85 pts):
  - compile and run in jGrasp
  - be compatible with the instructor-provided classes
- ✓ Coding style and comments
  - Coding style (meaningful identifiers, indentation, etc.) and comments (15 pts)
    - A comment at the top of each file with your name
    - Inline comments explaining code logic for more complex methods

You are NOT allowed to help one another with this program or use somebody else's code – check the rules listed on the syllabus. However, you may seek help from CSS mentors or a TA or an instructor.

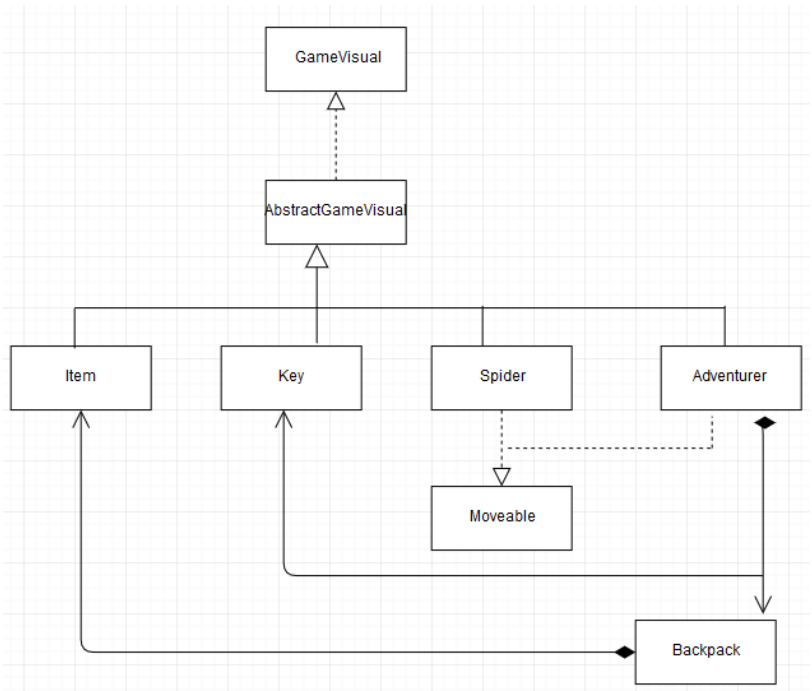
Problem statement

For this assignment you are to write classes that will be used to create various game objects. Overall, the program is to simulate a 2D world in which an adventurer is walking around, trying to collect artifacts. Unfortunately for the adventurer, the world is also occupied by giant spiders that destroy everything in their path. After the random initial placement of various game objects, adventurers and spiders move according to a set of type-specific instructions.

The front-end of the simulation has been written and provided with this assignment. Your job is to develop the back-end according to the set of instructions given below.

Class hierarchy

This is the simplified class diagram for the set of classes and interfaces you are to define:



Interfaces

Your program is to define the following two interfaces:

- Interface `GameVisual`: This interface defines the methods necessary for all game objects that are to be rendered on the screen. It consists of the following methods:

`getChar()`: Returns a character to display on the screen
`getX(), getY()`: Return the x or y position of the game object
`setX(int x), setY(int y)`: Set the x or y position of game object
`toString()`

- Interface `Movable`: This interface defines the `move()` method necessary for every movable game object to make a legal move in the world. This method takes no arguments and does not return anything. When implemented in a class, it will mutate the state of the object that implements it.

Classes

Your program is to define six classes:

- An abstract class `AbstractGameVisual` that implements the `GameVisual` interface. It needs to provide x and y position of a game object and define all methods other than `getChar()` method. In addition, it should contain a parameterless constructor.
- A concrete class `Key` that is an `AbstractGameVisual` type, with an additional name component denoted by the constant character 'K' – the same character for all objects of this type. Defines a parameterless constructor and `getChar()` method.
- A concrete class `Item` that is an `AbstractGameVisual` type, with 3 additional components: each item of this type has its individual weight, point value, and a char name component. Defines a parameterless constructor, all setters and getters, and `toString()` methods.
- A concrete class `Backpack` that is not related to other types by inheritance hierarchy and will be used to create objects that contain all artifacts collected by an adventurer (one backpack per adventurer). Each backpack object has a preset maximum weight it can carry. This maximum weight is assigned via a parameterized constructor. You can think of this class as a list of items. Defines `getWeight()` that returns the total weight of all items placed in the backpack, `getMaxSize()` that returns the maximum weight the backpack can carry, `addItem(Item someItem)` to add an item to a backpack, and `toString()` methods.
- A concrete class `Spider` that is an `AbstractGameVisual` type with an additional name component denoted by the constant character 'S' – the same character for all objects of this type. Defines a parameterless constructor and `getChar()` method.
  In addition, `Spider` is also a `Movable` type and implements the `Movable` interface. The `move` method in this class should move a spider object from its current location to a new location following the zig-zag pattern. However, spider objects do not move upon every call to `move`, but rather every 10th time the method is called. The way the animation is written right now is based on a timer, so every time a timer ticks, the `move` method is called but we don't want to move the spider that often. The 10th call to `move` changes the spider object's location two cells to the left. The next 10th call to `move` changes the spider object's location two cells up. Then, the pattern gets repeated. This is the picture of the pattern, where bold indicates the original location:

  S
  .
  S.**S**

  To support the move pattern, you may define additional properties in this class that help remember/determine the object's direction and the `move` call number.
- A concrete class `Adventurer` that is an `AbstractGameVisual` type, with an additional name component denoted by the constant character 'A' – the same character for all objects of this type. Each adventurer object also has his own backpack, which for the purposes of this simulations should be preset to 1000 units of weight in the parameterless constructor. Each adventurer object keeps on collecting `Item` objects into his backpack until his backpack capacity is reached. He also should collect

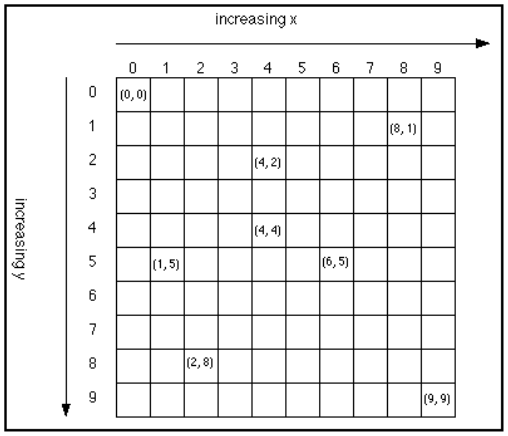
one `Key` object (but no more than one) because without the key, he won't be able to open any of the boxes he may collect. Defines `hasSpace(double newItemsWeight)` that informs the user whether the adventurer's backpack has enough space for the new item, `addItem(Item someItem)` to add an item to a backpack an adventurer is carrying, `addKey()` that changes the adventurer object to indicate he is in possession of a key, `hasKey()` that informs the user whether the adventurer has a key, `getChar()` and `toString()` methods.
In addition, `Adventurer` is also a `Movable` type and implements the `Movable` interface. The `move` method in this class should move an adventurer object from its current location to a new location following the square spiral pattern. The 1st call to `move` changes the adventurer object's location one cell to the left. The 2nd and 3rd call to `move` change the adventurer object's location one cell up. The 4th, 5th, 6th call to `move` change the adventurer object's location one cell to the right. The 7th, 8th, 9th, 10th call to `move` change the adventurer object's location one cell down. Then, the pattern gets repeated. This is the picture of the pattern, where bold indicates the original location:

AAAA
A … A
A**A .** A
. . . A
. . . A

To support the move pattern, you may define additional properties in this class that help remember the object's direction and the `move` call number.

Additional Information
In your move methods, you need to account for world boundaries, as if the world formed a sphere. This means that an object never disappears but rather when it reaches the right boundary, it reappears on the left and when it reaches the top boundary, it reappears at the bottom. The world in our simulation has dimensions 50 x 50.

Note that the Java geometry flips the y-axis so that the coordinate system is modeled as:



When the simulation runs, objects are consumed by spiders or collected by adventurers only if a movable object lands in the same cell. This logic is already implemented and this note is just an explanatory note so that you understand the output.

Your code should be organized in the following manner:

- All fields are declared before all constructors.
- All constructors are declared before all other methods.
- All static fields are declared before all non-static fields.
- All static methods are declared before all non-static methods.
- Among the same type of entity (constructors, instance methods, static methods, instance fields, static fields), the order for information hiding modifiers should be public, then private.

Extra Credit

You may earn extra credit of up to 10 points for this assignment. Extra credit is given at the discretion of the instructor and graded more strictly than the regular parts of the program.

- Create a complete UML diagram for the classes and interfaces you are defining for this project
- Javadocs for all classes, methods, and interfaces
- If you have some other idea for extra credit, please, run it by the instructor first.

Project Submission

On or before the due date, use the link posted in *Canvas* next to *project 3* to submit your `zipped` classes. The file should be named `project3.zip` and contain all the 8 files described above: `GameVisual.java, Movable.java, AbstractGameVisual.java, Key.java, Item.java, Backpack.java, Spider.java, Adventurer.java.` The best way to create this zipped version is to put all `.java` files in one folder and zip the folder. Do not include instructor files or `.class` files generated by the JVM.