Assignment 1

COMP 250 Fall 2020

posted:	Monday, Sept. 21, 2020
due:	Wednesday, Oct. 7, 2020 at 23:59

Learning Objectives

This assignment is meant for you to practice what we have learned in class in the past few weeks. A lot of the design decision have been taken for you, but it is important for you to ask yourself why each choice has been made and whether there could be a better way of doing it. You'll soon realize that the classes you have to write are all closely related to one another. We hope that the assignment will help you appreciate the importance of class design. This is of course just a taste, you will learn much more about it in COMP 303. As mentioned in class, we suggest you take the time to draw out a class diagram. This should help you develop a clear picture of the relationship between all these classes.

General Instructions

- **Submission instructions**
  - Late assignments will be accepted up to 2 days late and will be penalized by 10 points per day. Note that submitting one minute late is the same as submitting 23 hours late. We will deduct 10 points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it wrong file submitted, wrong file format was submitted or any other reason. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
  - Don't worry if you realize that you made a mistake after you submitted : you can submit multiple times but only the latest submission will be kept. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and codePost may be overloaded during rush hours).
  - These are the files you should be submitting on codePost:
    - `Insect.java`
    - `Hornet.java`
    - `HoneyBee.java`
    - `BusyBee.java`
    - `StingyBee.java`
    - `TankyBee.java`

    - `Tile.java`
    - `SwarmOfHornets.java`

  Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks

- ADDED ON SEPT. 22nd Please note that the classes you submit should be part of the default package. Please remove any package declaration you have at the top.
- You will have to create all the above classes from scratch. The assignment shall be graded automatically. Requests to evaluate the assignment manually shall not be entertained, so please make sure that you follow the instruction closely or your code may fail to pass the automatic tests. Note that for this assignment, you are NOT allowed to import any other class (including for example `ArrayList` or `LinkedList`). **Any failure to comply with these rules will give you an automatic 0.**
- Whenever you submit your files to codepost, you will see the results of some exposed tests. These tests are a mini version of the tests we will be using to grade your work. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will test your code on a much more challenging set of examples. We highly encourage you to test your code thoroughly before submitting your final version.
- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Piazza.

Toward a Tower Defense Game

For this assignment you will write several classes as a first step toward building a very simple Tower Defense game. Tower Defense games are strategy games where the goal is to defend the player's territories by obstructing the advance of the enemies troops. Note that you do not need to be familiar with Tower Defense games in order to successfully complete the assignment. Make sure to follow the instructions below very closely. Note that in addition to the required methods, you are free to add as many other **private** methods as you want. You can also add `public toString()` methods in each class to help with the debugging process. No other additional non-`private` method is allowed.

Let's start by creating a skeleton for some of the classes you will need.

- Write a class called `Tile`. You can think of a tile as a square on the board on which the game will be played. We will come back to this class later. For the moment you can leave it empty while you work on creating classes that represents characters in the game.
- Write an `abstract` class `Insect` which has the following `private` fields:
  - A `Tile` representing the position of the insect in the game.
  - An `int` representing the health points (hp) of the insect.

  The class must also have the following `public` methods:
  - A constructor that takes as input a `Tile` indicating the position of the insect, an `int` indicating its hp. The constructor uses its inputs to initialize the corresponding fields. Do not make a copy of the object of type `Tile`.
  - A `final getPosition()` method to retrieve the position of *this* insect. Do not make a copy of the object of type `Tile`.
  - A `final getHealth()` method to retrieve the health points of *this* insect.
  - A `setPosition()` method that takes a `Tile` as input and updates the value stored in the appropriate field. Do not make a copy of the object of type `Tile`.
- All of the following must be subclasses of the `Insect` class:
  - Write a class `Hornet` derived from the `Insect` class. The `Hornet` class has the following `private` field:
    - An `int` indicating the attack damage of the hornet.

    The `Hornet` class has also the following `public` method:
    - A constructor that takes as input a `Tile` which represents the position of the hornet, an `int` indicating its hp, and an `int` indicating its attack damage. The constructor uses the inputs to create a `Hornet` with the above characteristic.
  - Write an `abstract` class `HoneyBee` derived from the `Insect` class. The `HoneyBee` class has the following `private` field:

    * An `int` indicating how much this bee costs in food. Whenever a player wants to place a bee in the game, they will need to have a certain amount of food available. Larvae need to eat to become strong bees!

  The `HoneyBee` class has also the following `public` methods:

    * A constructor that takes as input a `Tile` which representing the position of the bee, an `int` indicating its hp, an `int` indicating its cost in food respectively. The constructor uses the inputs to create a `HoneyBee` with the above characteristic.
    * A `getCost()` method which returns how much this bee costs in food.

- Write a class `BusyBee` derived from the `HoneyBee` class. The `BusyBee` class is used to represent bees that collect pollen. This class has no fields, but it has the following `public` method:

    * A constructor that takes as input a `Tile` which represents the position of the bee in the game. The constructor uses the input to create a bee given with the given position, health equal to 5, and food cost equal to 2.

- Write a class `StingyBee` derived from the `HoneyBee` class. The `StingyBee` class is used to represents bees that sting their enemies. The class has the following `private` field:

    * An `int` indicating the attack damage of the bee.

  The `StingyBee` class has also the following `public` method:

    * A constructor that takes as input a `Tile` which represents the position of the bee, and an `int` indicating its attack damage. The constructor uses the inputs to create a bee given the position, the attack damage, health equal to 10, and food cost equal to 1.

- Write a class `TankyBee` derived from the `HoneyBee` class. The `TankyBee` class is used to represents bees that are slow to sting, but are very sturdy. These bees are great to stall the advance of a swarm of hornets. The class has the following `private` field:

    * An `int` indicating the attack damage of the bee.
    * An `int` indicating the armor of the bee.

  The `TankyBee` class has also the following `public` method:

    * A constructor that takes as input a `Tile` which represents the position of the bee, an `int` indicating its attack damage, and an `int` indicating its armor. The constructor uses the inputs to create a bee given the position, the attack damage, the armor, health equal to 30, and food cost equal to 3.

You can now leave these classes as they are, we will come back to them later. Let's now focus on the two other classes:

- Write a class `SwarmOfHornets`. This class represents a group of hornets together. Despite what the name 'swarm' might let you think, these hornets group together forming a very neat line. The purpose of this class is to implement your own queue of hornets. Note that you

will not be tested on the efficiency of your code, but we highly encourage you to think about how you can make your code more efficient when implementing a data structure that does not have a fixed size of elements and for which the most common operations are removing the first element of the queue and adding an element to the end of the queue. Note that even though the remove method you will be writing for this class serves a general purpose, when using this data type in this assignment only hornets in first position will end up being removed from a swarm. We want to stress once again that you are not allowed to import any class for this assignment! The class `SwarmOfHornets` must have the following `private` fields:

  - An array of `Hornet`s which will be used to store the hornets that are part of the swarm.
  - An `int` indicating the size of the swarm, i.e. how many hornets are part of the swarm.

  The class must also have the following `public` methods:

  - A constructor that takes no inputs and creates an empty swarm. To do so, the fields should be initialized to reflect the fact that at the moment there are **no hornets** in the swarm.
  - A `sizeOfSwarm()` method that takes no inputs and returns the number of hornets that are part of *this* swarm.
  - A `getHornets()` method which takes no inputs and returns an array containing all the hornets that are part of *this* swarm. The hornets should appear in the order in which they have joined the swarm. This array must contain as many elements as the number of hornets in the swarm, and it should not contain any `null` elements.
  - A `getFirstHornet()` method which takes no inputs and returns the first hornet that joined the swarm. If there's no hornet in this swarm, then the method returns `null`. Note that this method should not remove the hornet from the swarm.
  - An `addHornet()` method which takes as input a `Hornet` and does not return any value. The method adds the hornet at the end of the queue of hornets in *this* swarm. Make sure to handle the case in which there might not be enough space for this hornet to join the swarm. In such case, you need to make sure to create additional space. No hornet should be rejected from the swarm. If need be, this is a great place to create your own private method to help you with the implementation. Warning: make sure that no hornet is removed from the swarm as a result of adding the new one.
  - A `removeHornet()` method which takes as input a `Hornet` and returns a `boolean`. The method removes the **first occurrence** (remember the hornets should appear in the swarm based on the order in which they joined it) of the specified element from *this* swarm. If no such hornet exists, then the method returns `false`, otherwise, after removing it, the method returns `true`. Note that this method should **compare hornets using directly their reference**.

- Go back to the class `Tile` which you have created before. To this class add the following `private` fields:
  - An `int` indicating the food present on the tile. Bee collect food to feed their larvea. This

is how strong bees are raised. If a player wants to add a new bee to the game, they will need to have enough food to do so.

- A `boolean` indicating whether or not a bee hive has been built on the tile.
- A `boolean` indicating whether or not a hornet nest has been built on the tile.
- A `boolean` indicating whether or not this tile is part of the path that leads from the hornet nest to the bee hive.
- A `Tile` containing a reference to the next tile on the path from the hornet nest toward the bee hive. It contains `null` if the tile is not on the path or at the end of it.
- A `Tile` containing a reference to the next tile on the path from the bee hive toward the hornet nest. It contains `null` if the tile is not on the path or at the end of it.
- A `HoneyBee` indicating the bee positioned on the tile.
- A `SwarmOfHornets` containing all the hornets positioned on the tile.

The class must also have the following `public` methods:

- A constructor that takes no inputs and creates a new tile. A new tile does not have a bee hive, a hornet nest, nor is on the path that leads from one to the other. On a new tile there's no food, no bee, and no hornets. Initialize the fields to represent this.
- A second constructor that takes all the inputs needed to initialize the fields from this class. The method takes the inputs in the same order as the fields are listed above. Please do not make copies of any of the objects received as input.
- A `isHive()` method which returns whether or not a bee hive has been built on this tile.
- A `isNest()` method which returns whether or not a hornet nest has been build on this tile.
- A `buildHive()` method which builds a bee hive on this tile. This is a `void` method that simply updates the field indicating whether or not there's a bee hive on the tile.
- A `buildNest()` method which builds a hornet nest on this tile. This is a `void` method that simply updates the field indicating whether or not there's a hornet nest on the tile.
- A `isOnThePath()` method which returns whether or not this tile is part of the path that leads from the hornet nest to the bee hive.
- A `towardTheHive()` method which returns the next tile on the path from the hornet nest to the bee hive. If this tile is not on the path, then the method returns `null`.
- A `towardTheNest()` method which returns the next tile on the path from the bee hive to the hornet nest. If this tile is not on the path, then the method returns `null`.
- A `createPath()` method which takes two `Tile`s as input representing the next tile on the path toward the hive, and the next tile on the path toward the nest respectively. The method updates the respective fields, making this tile become a tile that is part of the path that leads from the hornet nest to the bee hive.

- A `collectFood()` method with takes no inputs and returns an `int` representing the food stored on the tile. The tile is then left with no food.
- A `storeFood()` method which takes an `int` as input representing the amount of food received and it adds it to the food stored on the tile. This method does not return anything.
- A `getBee()` method which returns the bee positioned on this tile. You should not make a copy of this object.
- A `getHornet()` method which returns the hornet which first joined the swarm on this tile. You should not make a copy of this object.
- A `getNumOfHornets()` method which returns the number of hornets positioned on this tile.
- An `addInsect()` method which takes as input an `Insect` and adds it to the tile. Note that a bee can be added to the tile only if there's no other bee positioned on this tile. More over, no bee can be positioned on the hornet nest! On the other hand, there's no limit to the number of hornets that can be positioned on a tile. But hornets can only be positioned either where there's the hornet nest, or where there's the bee hive, or on a tile that is on the path from the nest to the hive. The method returns `true` if the insect was successfully added to the tile, `false` otherwise. Note that adding an insect on a tile, not only changes the properties of the tile, but also the properties of the insect. That is, the insect should now result as positioned on this tile. Once again, please do not make a copy of the input object.
- A `removeInsect()` method which takes as input an `Insect` and removes it from the tile. The method should also return a `boolean` indicating whether or not the operation was successful. Note that removing an insect from a tile, not only changes the properties of the tile, but also the properties of the insect. That is, the insect should now result as not positioned on a tile. You can indicate this by updating the position to be `null`.

We are now ready to go back to the classes we created at the beginning.

- In the class `Insect` go back to the constructor and make sure that when an insect with a specified position is created, such insect is also added to the corresponding tile. Note that it is not always possible to do that (for example there cannot be more than one bee on the same tile). If it is not possible to add the insect to the specified tile, then the constructor should *throw* an `IllegalArgumentException`.

  To the `Insect` class add the following `public` methods:

  - A `takeDamage()` method which takes as input an `int` indicating the damage received by the insect. The method applies the damage to the insect by modifying its health. To do so, subtract the damage from the insect's health points. Note that if the insect is a bee positioned on the bee hive, then the damage should be reduced by 10% before being applied (the damage should be rounded toward 0). Moreover, if after the damage is applied the insect ends up having a non-positive health (0 or below), then the insect

has been killed. In such a case, it should be removed from the game. To do that, remove it from the tile. This method does not return any value.

  - An `abstract` method `takeAction()` which takes no inputs and returns a boolean. This method should be abstract (thus, not implemented) because the action to take depends on the type of the insect.

  - Override the `equals()` method. It takes as input an `Object` and returns `true` if it matches `this` in type, position and health. Otherwise, the method returns `false`.

- In the `Hornet` class do the following:

  - Implement the `takeAction()` method. If there's a bee positioned on the same tile as this hornet, the hornet will sting it inflicting on the bee an amount of damage equal to this hornet's attack damage. If on the other hand there's no bee on the same tile as this hornet, then the hornet will try to move toward the bee hive. Unless of course the hornet is already on the bee hive, in which case there's nothing for this hornet to do (the hornets have won the game!). In the case in which the hornet tries to move toward the hive, it will do so by moving to the next tile on the path. Note that this means that the tiles and the position of this hornet must all be updated accordingly. The method returns true whether the hornet either stings a bee or moves to the next tile. Note that you can assume that the hornet will never be positioned on a tile that is not the bee hive, the hornet nest, or on the path from the nest to the hive.

  - Override the `equals()` method. The method returns `true` if the `Object` received as input matches `this` in type, position, health and attack damage. Otherwise the method returns `false`. Note that you do not want to rewrite code that you have already written in the superclass. How can you access methods from the superclass that have been overridden?

- In the `HoneyBee` class do the following:

  - As for the `Hornet`, override the `equals()` method. The method returns `true` if the `Object` received as input matches `this` in type, position, health, and food cost.

- In the `BusyBee` class do the following

  - Override the `takeAction()` method. This type of bees collect pollen, so excuting an action results into 2 food being added to the tile where the bee is positioned. The method returns `true`.

- In the `StingyBee` class do the following:

  - Implement the `takeAction()` method. A sting-y bee that is positioned on the path that leads from the nest to the hive or on the bee hive itself will attempt to sting an hornet. If on the other hand, it is not positioned on the path that leads from the nest to the hive nor on the bee hive itself, then it won't do anything and the method will return `false`. Note that if the bee tries to sting an hornet, it is not just simply stinging a random one! It will start to look for the first non-empty swarm that it can find on the path that leads from the bee's tile to the hornet nest. When it finds it, it will sting the first hornet in

the swarm by inflicting it an amount of damage equal to this bee's attack damage. Note that the bees cannot sting a hornet that is positioned on its nest! If there's no hornet to be stung, then the bee won't do anything and the method return `false`. If the bee stings a hornet, then the method returns `true`. Note that no matter what the bee does, the bee will end its action on the same tile on which it was at the beginning of it. So, this method does not modify the tile on which the bee is positioned on.

  - As for the `HoneyBee`, override the `equals()` method. The method returns `true` if the `Object` received as input matches `this` in type, position, health, food cost, and attack damage.

- In the `TankyBee` class do the following
  - Implement the `takeAction()` method. Tanky bees can only sting hornets that are positioned on the same tile as them. The bee will look to sting the first hornet in the swarm (if not empty) by inflicting them an amount of damage equal to the bee's attack damage. If no hornet is there, then the tanky bee doesn't do anything. The method returns `true` if the bee stings a hornet, `false` otherwise.
  - Override the `takeDamage()` method. The method receives an `int` as input indicating the damage the bee should receive. Since this bee is tanky, before applying the damage to the bee (in the same way as `takeDamage()` from `Insect` does), the damage should be multiplied by a multiplier which depends on this bee's armor. The multiplier is a double equal to $100/(100 + armor)$. The damage to be applied to the bee should be then round toward 0. Once again, you do not want to rewrite code that you already have written in one of the superclasses. How can you access methods from the superclass that have been overridden?
  - As for all the other bee classes, override the `equals()` method. The method returns `true` if the `Object` received as input matches `this` in type, position, health, food cost, attack damage, and armor.