Homework 02

Due: 2016-09-29T23:59:59

Introduction

Civilization (https://en.wikipedia.org/wiki/Civilization_(video_game)) is a turn-based strategy game centered around founding and building a civilization. More commonly referred to as Sid Meier's Civilization, the game has been around since 1991 and has since developed somewhat of a cult following. This semester we are going to be implementing our own version of this game for your homework assignments and by the end of the semester you will have a working game you can show off to your friends! If you are not familiar with Civilization, there is a free version (https://en.wikipedia.org/wiki/Freeciv) that you can play in order to get familiar with the game.

This assignment will get you ramped up on Classes and Data Abstraction. This project will:

- Test your ability to convert a written description into a Java program
- Test your ability to use access modifiers correctly
- Test your understanding of static vs. instance data
- Test your ability to create, use, and manipulate arrays

Problem Description

For the last assignment we had you implement a (what should have been) very straightforward and basic version of Civilization. Like many of you noted on Piazza that implementation was so basic it just barely followed the actual game design! Well keep in mind we are working on an iterative process of creating this big project, and the next step is to get some more features of the game set up. You will be creating classes to represent **Civilizations**, **Resources**, **Victories**, and **Different types of Terrain**.

Before we begin…

- Like all homework descriptions, it is very important you read and understand this entire document.
- It is important to note that you will not need to create a new repository for this homework. You will be working in the same git repository you set up for the first homework.
- This homework will be very focused on details. So make sure you read the description carefully and implement every detail described!
- We have provided some files for you as well as a new driver for your project. You will need to replace your `Civilization.java` file from homework 1 with the `CivilizationGame.java` one found in this folder (./hw2.zip). Unfortunately the first assignment wasn't quite complicated enough and we need to make sure everyone is caught up.
- Note that many of the instance variables will require getter and setter methods (since all of your instance data should be private) that we did not explicitly describe in the homework description. If you complete all of the classes correctly then `CivilizationGame.java` should compile and run no problem! If you receive errors about missing methods, you should add that functionality to your class.

Solution Description

Resources

You will be writing 2 resource classes: `Treasury.java` and `Game.java`, Two additional resource classes, `CoalMine.java` and `Fish.java`, have already been written for you. Take a look at their descriptions to get ideas of how to complete the other resources.

`Treasury.java`:

This is one of the most important classes in the game. Almost any action that your civilization takes will require using their Treasury.

- A Treasury object has an `int` to represent the amount of gold coins the Civilization has. Each civilization starts out with 200 coins.
- There are three actions that can be taken with a treasury:
  - `getCoins` - This method should return the number of coins contained in your Treasury Object
  - `spend` - This method takes in an `int` cost and checks to see if there are enough coins to pay the cost proposed. If there are enough coins, decrement the coins instance variable, and return true. If there are not, return false.
  - `earn` - This method takes in an `int` amount of coins earned and updates the instance variable appropriately. This method should not return anything.

`Game.java`

Wild game objects will provide food for your citizens!

- They have one piece of instance data: an `int` healthIncrease.
- Game should have two constructors. One should take in a proposed healthIncrease and assign it to the instance data. The second should assign healthIncrease to 20 automatically. Utilize constructor chaining.
- You should write a method that will allow other classes to get the value of healthIncrease called `getHealth`.

`Fish.java`

Fish will provide food for your citizens! This class is also already provided for you. You do not need to change anything about the provided file. We provide it's description below.

- They have one piece of instance data, an integer healthIncrease.
- Fish should have two constructors. One should take in a proposed healthIncrease and assign it to the instance data. The second should assign healthIncrease to 20 automatically. Utilize constructor chaining.
- You should write a method that will allow other classes to get the value of healthIncrease.

`CoalMine.java`

- Coal is required to cook Game and Fish and feed your citizens! This class is also already provided for you. You do not need to change anything about the provided file. We provide it's description below.
- `CoalMine` has two pieces of instance data:
  - an `int coal` that represents the amount of coal present.
  - an `int` BURN_COST. Burn cost should be the same for all instances of coal and the value should *never change*. It should be initialized to a value of 10.
    - Again. *The value of BURN_COST should never change*.
- The class has 4 methods:
  - `burn` - This method will check to see if there is enough coal to burn by comparing the amount of coal the mine has to the BURN_COST. If there is enough, the amount of coal will be decremented by BURN_COST and the method will return true. Otherwise, return false.
  - `increaseCoal` - This method should take in an amount of coal that was mined. It should then add the passed in parameter to the current coal value.
  - You should write two methods that allow for other classes to access the BURN_COST and amount of coal instance data.

Victories

In Civilization there are many different ways to achieve "victory" or win the game. We will focus on just 2:

`Technology.java`

You will be writing a technology class that represents how advanced the civilization is. There will be some boolean values that will be used to determine if the player has won the game. Throughout the game, players will perform actions that allow them to improve their technology. Once their technology has reached a certain state, they will have won the game.

- Philosophy
  - Philosophy is represented by an integer `understanding` which should be initialized as zero and a boolean `foundMeaningOfLife` which should be initialized to false.
  - You should have one method named `philosophize` that will increase understanding by 25 and one method `improveWriting` that will increase understanding by 10.
  - If understanding ever surpasses 200, foundMeaningOfLife should be set to true.
- Architecture
  - Architecture is represented by an integer `experienceLevel` and one boolean value `builtWonderOfTheWorld`. `builtWonderOfTheWorld` will be initialized as false, and `experienceLevel` should be initialized as zero.
  - You should have a method `increaseExperience` that will increase the `experienceLevel` by the value passed in. If the `experienceLevel` surpasses 200, `builtWonderofTheWorld` should become true.
- Finally you will want to create a method that returns whether or not there is a technology win called `hasTechnologyWin`. If `foundMeaningOfLife` and `builtWonderOfTheWorld` become true then victory has been achieved!

`Strategy.java`

The strategy class has to do with war strategy.

- This class should have a `strategyLevel` integer, a `BATTLE_INCREASE` integer, a `SIEGE_INCREASE` integer, and a boolean `conqueredTheWorld`.
  - `BATTLE_INCREASE` should be initialized to 10 and should hold the same value throughout all instantiations of Strategy.
  - `SIEGE_INCREASE` should be initialized to 40 and should hold the same value throughout all instantiations of Strategy.
  - Neither of these variables should be able to change after they have been instaniated.
- You should create two methods, named `battle` and `siege`. Each should increment the `strategyLevel` by the corresponding constant. If `strategyLevel` ever becomes greater than 180, `conqueredTheWorld` should be set to `true`.

If `conqueredTheWorld` gets set to true, then victory has been achieved!

Terrains

Your Civilization will have 3 Terrains: `Desert.java`, `Hills.java`, `River.java`. Terrains provide you with food and treasure.

`River.java`

- When a `River` object is created it must be given a name.
- Your River object should have a name and an array of `Fish`. The River will never have more than 5 fish, and it will initialize with 5 fish with random integer healthIncrease values between 0 inclusive and 5 exclusive.
- You should create a `getFish` method that will return a `Fish` from your array if there are any `Fish` available. Once one `Fish` object is returned from the array, you should never be able to access it again. If no `Fish` are available, `getFish` should return null.
- However, all hope is not lost! You will also be writing a `replenishFish` method. This method will randomly generate 5 Fish to fill your backing array. However, this method will only replenish the fish in the river if there are no more fish left! Return whether or not fish were replenished.
  - This method gets called in the main class in `CivilizationGame.java` so you do not need to call it yourself.

`Hills.java`

`Hills.java` has been provided for you. We provide its description below, but you do not need to make any changes to this class.

The Hills terrain allows your citizens to gain 3 resources: Game, Gold, and Coal.

- The Hills constructor: randomly hides gold coins and coals in different places in the Hills. It will also fill the Game array with new wild Game objects.
- excavate method: randomly looks up a spot in the two dimensional goldLocation array and returns the number of gold coins found there.
- mineCoal method: randomly looks up a spot in the two dimensional coalLocation array and returns the number of coals found there.
- hunt method: looks at the last Game object in the Game array and returns it. It also moves the numGame pointer to point at the next Game object available. If there are no Game objects available to return, hunt will return null.
- replenishGame method: randomly generates new Game to fill the Game array. The numGame value is also reset to the length of the Game array.

`Desert.java`

The primary function of a Desert is to find treasure of course! Some of `Desert.java` has been implemented for you. You will not need to add anything to the `lost` method.

- You will be writing a method `findTreasure` that will randomly generate and return an integer value between 0 exclusive and 500 inclusive that represents the amount of coins that were found.
- However, Deserts are also fairly dangerous, and our players may get lost. If the player does get lost, you should call the lost method (written for you) to give them another shot at escaping the Desert. If lost() returns true, call lost() again. If lost() returns false, send your player off to find treasure again. A player should get lost 10% of the time.
    - Hint: Use random number generation to simulate the 10% requirement.
    - One way is to generate a random integer between 0 inclusive and 10 exclusive indicating whether or not the player has gotten lost while searching for treasure. How would 10% be evaluated given a random number between 0 and 10?
    - Hint: You will be calling the `lost` method inside of `findTreasure`.

`Population.java`

This class represents the population of a certain civilization as a whole.

- Each population is made up of warriors who go to battle and civilians who go to work. Amount of warriors and civilians should be made up of whole numbers.
    - A population starts out with 50 warriors and 50 civilians
- A population also has a representation of the amount of happiness that starts at 200.
- Population has the ability to
    - `increaseHappiness` that takes in the integer amount to increase the happiness by.
    - `decreaseHappiness` that takes in the integer amount to decrease the happiness by. Note that happiness should never go below 0.
    - `canWork` which takes in the number of workers needed and checks to see if there are enough civilians to satisfy the required number of workers. If there are enough civilians, then the amount of civilians gets decreases by the input amount.
    - `canBattle` is provided for you and determines if you have enough warriors for a randomly generated battle.
    - `hunt` takes in a Hills instance as a parameter and returns the result of the hunt
    - `fish` takes in a River instance as a parameter and returns the result of the fishing trip
    - `canCook` takes in as parameters `Game` and the Civilization's `CoalMine`. If possible, burn coal 4 times, increase the number of warriors by 40, increase the number of civillians by 60 and return true. Otherwise return false.
    - `canCook` takes in as parameters `Fish` and the Civilization's `CoalMine`. If possible, burn coal 4 times, increase the number of warriors by 10, increase the number of civillians by 15, and return true. Otherwise return false.

Settlements

Below describes the classes involved in settling new cities.

`Building.java`

This class is also already provided for you. You do not need to change anything about the provided file. We provide it's description below.

A Building object is the most basic element of a civilization. A building should have an integer cost and an integer workersRequired. A Building should have a constructor that takes in cost and workersRequired parameters and assigns them to each corresponding piece of instance data.

`Settlement.java`

A Settlement object is composed of Building objects. Your Settlement object MUST have an array of Building objects, as well as a String name. `Settlement` will have the following methods:

- `addBuilding` takes in a Building as a parameter and adds it to the backing array. There is no limit to the number of buildings that can be in a settlement. I should be able to add as many buildings to a settlement as I want.
- a method `public boolean build(int allottedMoney, Population population, int cost, int workersRequired)` that builds a new Building object and stores it in your backing array. A new Building should only be built if the cost is smaller than the allottedMoney, and the Population has enough civillians to complete the job.
- `expandSettlement` which doubles the size of your backing array while retaining all of the buildings that have been built

Civilizations

You will be writing 3 civilization classes: `Egypt.java` , `RomanEmpire.java` , and `QinDynasty.java`

- Each Civilization has a `Population` , a `Treasury` , a `CoalMine` , and a `River` .
- Each Civilization also has `Technology` and `Strategy`
- Each Civilization has a way to `settle`
  - `settle` take sin a prospective settlement, adds it to an array of settlements and returns whether or not the settlement was established.
- Each Civilization can have a maximum of 10 `Settlement` s and should have a way of returning how many settlements have already been settled through a method called `getNumSettlements` .
- Remember to consider how the methods you wrote interact here, especially when you are creating buildings.

`Egypt.java`

- `Egypt` has a `Desert`
- Has a method `buildPyramid` that takes in a Settlement and builds a pyramid building in that settlement.
  - Pyramids cost 500 coins and a worker requirement of 100
  - Returns whether or not the pyramid was built
  - It will increase the civilization’s architecture experience by 10.
- Has a method `practiceHieroglyphics`
  - Practicing hieroglyphics will help Egyptian Technology by improving this civilization’s writing.
  - It will increase the Population’s happiness by 10.

`QinDynasty.java`

- `QinDynasty` has Hills
- Has a method `buildWall` that takes in a Settlement and builds a wall in that settlement
  - A wall costs 1000 coins and requires 100 civillians.
  - Returns whether or not the wall was built
  - It will increase the civilization’s architecture experience by 10.
- Has a method `buildHouse` that takes in a Settlement and builds a house in that settlement
  - A house costs 30 coins and requires 8 civillians.
  - Returns whether or not the house was built
  - It will increase the civilization’s architecture experience by 10.

- Has a method `establishLegalism`
  - Studying a philosophy of legalism will help improve a civilization's philosophy technology.
  - It will decrease the happiness of your Population by a value of 20. If the population's happiness would go below 0 after the completion of the action, the action should be aborted.

`RomanEmpire.java`

- `RomanEmpire` has `Hills`
- Has a method `buildAqueduct` that takes in a Settlement and builds an aqueduct in that settlement
  - An aqueduct costs 250 coins and requires 130 civillians.
  - Returns whether or not the aqueduct was built
  - It will increase the civilization's architecture experience by 10.
- Has a method `buildBathHouse` that takes in a Settlement and builds a bath house in that settlement
  - A Roman bath house costs 110 coins and requires 20 civillians.
  - Returns whether or not the bath house was built
  - It will increase the civilization's architecture experience by 10.
- Has a method `buildVilla` that takes in a Settlement and builds a villa in that settlement
  - A villa costs 80 coins and requires 15 civillians.
  - Returns whether or not the villa was built
  - It will increase the civilization's architecture experience by 5.
- Has a method `studyPhilosophy`
  - Studying Roman philosophy will decrease the happiness of your Population by a value of 10. If the population's happiness would go below 0 after the completion of the action, the action should be aborted.
  - It will help improve your civilization's Philosophy technology.

As you work…

Any time you get some code written and running you should push your changes up to the remote repository:

1. Type `git status` to see what changes are staged or unstaged
2. Type `git add *` to add all changes you have made to be staged for commit
3. Type `git commit -m "My message"` to commit your changes to your local repository.
4. Type `git pull origin master` to make sure you are up-to-date with the remote version of the repository.
5. Type `git push origin master` to push your changes up to the remote repository.

Tips, Tricks, and Reminders

- You are not allowed to use `Arrays.java` or `ArrayList.java`
- You are not allowed to use inheritence, polymorphism, or abstract classes. If you don't know what any of that is, ignore this message.
- Confused about how to interpret the description and turn it into a class? Use the provided files and their corresponding descriptions to help you get started!
- This has a lot of classes to it, but a lot of them are very similar to eachother. Remember to pay attention to details!
- Start early. Ask for help early. Note that on the day homeworks are due the TA lab traditionally gets very busy!

Checkstyle

Peruse the CS1331 style guide here (http://cs1331.gatech.edu/cs1331-style-guide.html). This also has the instructions for how to run checkstyle on your code. Here's a more step-by-step rundown:

- Go to the style guide (http://cs1331.gatech.edu/cs1331-style-guide.html)
- Right click on the link that says `checkstyle-6.2.2.jar` and select "save as" or "save link as" or whatever it is your browser suggests.
- Save the jar file in an easily accessible location.
- Copy the jar file into the same directory as the `.java` classes you want to check.
- Open the command line and `cd` into the directory containing your `.java` files and the checkstyle jar file.
- Type and run `java -jar checkstyle-6.2.2.jar MyJavaFile.java` replacing `MyJavaFile.java` with the name of your java file.

What gets printed out is the list of all style errors your code has. Do your best to fix all of them and start coding in the correct style. Each individual checkstyle error results in -1 points on your assignment.

Maximum number of points you can lose on Checkstyle this assignment: **20**

Turn-in Procedure

In order to submit your assignment you need to ensure that your working code is pushed to your remote repository by the due date! Follow the instructions outlined above to push your code to your remote repository!

You should have the following in your repository:

- `CivilizationGame.java`
- `CoalMine.java`
- `Treasury.java`
- `Game.java`
- `Fish.java`
- `Technology.java`
- `Strategy.java`
- `River.java`
- `Hills.java`
- `Desert.java`
- `Population.java`
- `Building.java`
- `Settlement.java`
- `Egypt.java`
- `QinDynasty.java`
- `RomanEmpire.java`

Verifying Your Submission

Please be sure that any code you push compiles and runs through the command line! Pull from your repository and make sure everything is working how you want it!