Homework 7

Homework 7: Inheritance in Java

In this assignment, we will practice working with and implementing inheritance hierarchies and interfaces.

Details

You are welcome (and encouraged) to buddy or pair program for all parts of this assignment! The CS 60 Fall 2025 Syllabus can remind you of the differences. If you have not tried buddy or pair programming, give it a try! Feel free to use the "Search for Teammates" channel on Piazza to identify potential programming buddies, or ask the teaching team to help you find a partner during lab hours.

We are really excited to help people on Piazza - so please ask questions early and often! Some more tips:

- Please use the `#hw7` tag on Piazza.
- Your code will be **graded anonymously**. Please **do not include your name** anywhere in your submission.
- If you choose to **pair program**, remember to **add your partner to each Gradescope submission**.

— Your profs & your awesome CS60 Grutoring team!

Bonus!

This homework assignment is one of the best to review if/when you are practicing for **job interviews** in software. Interviewers *love* to ask questions about linked lists. Fortunately, this is not a job interview, so make sure to get help and work together (e.g. pair programming) as you are learning this stuff!

Starter Code

Before starting, **download the starter code** for this homework. Visit the Homework Setup guide for how to get the starter code setup in Racket or VS Code.

Navigation

This guide uses Document Tabs for navigation. You can find these tabs in the collapsible left panel and linked below.

Problem Instructions

- Problem 1: Inheritance
- Problem 2: Refactor

Other Details

- Grading Rubric

1 Problem 1: Inheritance

CS 60 HW 7

Problem #7.1: Inheritance and Interfaces

The **learning goal** of this problem is: practice extending a class and implementing an interface.

Fun Facts: The guinea pigs and clarinets in the part of the assignment are inspired by Prof. Wu, who originally wrote this assignment and who has three guinea pigs and plays the clarinet.

Prerequisites

Java inheritance and interfaces

Starter Files

The starter files you will reference for this problem are:

- `GuineaPig.java`
- `Instrument.java`

These files are located in the `src/main/java` within the `com.gradescope.hw7` package.

Tasks

The tasks for this problem are:

- [ ] Submit `GuineaPig.java`, `Clarinet.java`, and `Squeakable.java` in Gradescope HW #7.1 GuineaPig, Clarinet, and Squeakable (3 files).

- [ ] Submit `GuineaPigTests.java` and `ClarinetTests.java` in Gradescope HW #7.1 GuineaPigTests and ClarinetTests (2 files).

Make sure you are submitting the files in the correct assignment!

Instructions

Part A: Make `GuineaPig` comparable

Update the `GuineaPig` class to implement the `Comparable` interface. Read the steps below for details.

Step 1: Update class declaration

Update the class declaration of `GuineaPig` to implement the `Comparable` interface. Specifically, change the original declaration from:

```java
public class GuineaPig {
```

...to the new declaration:

```java
public class GuineaPig implements Comparable<GuineaPig> {
```

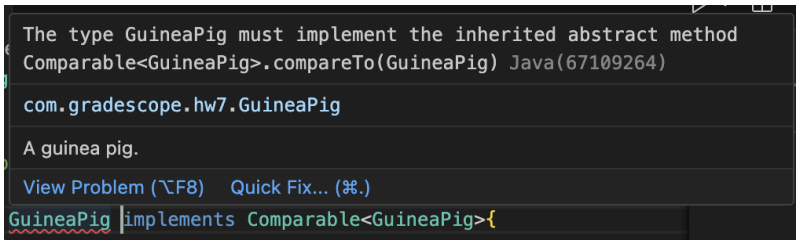
As you do this, think back to using `< >` in our `List` class. What does the use of `< >` in `Comparable<GuineaPig>` tell us about the `Comparable` interface?

Step 2: Add unimplemented method

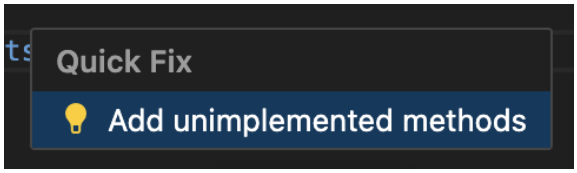
The change you just made to class `GuineaPig` (adding `implements Comparable<GuineaPig>`) is only a *promise* to implement the methods required by the `Comparable` interface. Your code will not compile until you *do* implement those methods (thus fulfilling that promise)!

Add the required method `compareTo`. It will take in an argument of type `GuineaPig`.

Pro Tip: Let VSCode help you! Look for the red squiggly that appears under `GuineaPig`. Mouse over it and see the following message, telling you that you must implement `compareTo`.



Then, click on Quick Fix to get the following menu. Click on “Add unimplemented methods”:



Now, you should see a stub (note: this is not a completed function) for `compareTo`! Thanks VSCode! Whether you do it yourself or let VSCode help, do not forget to add a doc comment! **Your doc comment should clarify how two guinea pigs are compared.** This said, you will probably have to read the logic below to properly specify how the comparison works...

Step 3: Write `compareTo` logic

Now, fill in the stub! The `compareTo` method should return:

- the value 0 if the argument guinea pig is "equal to" this guinea pig
- a value less than 0 if this guinea pig is "smaller than" the argument guinea pig
- a value greater than 0 if this guinea pig is "larger than" the argument guinea pig

For our purposes, we care only about the sign of the returned value (e.g. less than 0, exactly 0, or greater than 0). We do not care about its magnitude.

How do we determine smaller / equal / larger?

- If two pigs have different ages, the pig with the younger age is "smaller".
- If two pigs have the same age, the pig with the name that is alphabetically earlier is "smaller".
- If two pigs have the same age and name, they are "equal".

Hint: You can (and should) use the `String` method `compareTo(String anotherString)` to help out. Be careful to note that this method implements lexicographic order (where upper-case characters are "less than" lower-case characters). The `Integer` class also has a `static compare` method.

Step 4: Create `GuineaPig` test cases

Create the file `GuineaPigTests.java` in which you will write JUnit test cases. Your tests should:

- Test the constructor(s) and all public methods in `GuineaPig.java`
    - Be sure your tests for `compareTo` are thorough, i.e., they test each possible case
- Test the static type `Comparable<GuineaPig>`.

- Test that you can have references of static type `Comparable<GuineaPig>` that refer to `GuineaPig` objects.

- Test that we can compare such objects (static type `Comparable`) using `compareTo`. Note that the argument for `compareTo` must have static type `GuineaPig`.

  (Concept Check: What part of the code makes it so that `compareTo` must take in an object of static type `GuineaPig`?). Here, you need not test every path in `compareTo`, just that `compareTo` can successfully be called.

Reference the CS 60 HW5 - OOP in Java Resources tab for instructions on creating a new class file in VS Code.

Part B: Create `Clarinet` instrument

Create a new `Clarinet` class to inherit from the provided `Instrument` class. Read the steps below for details.

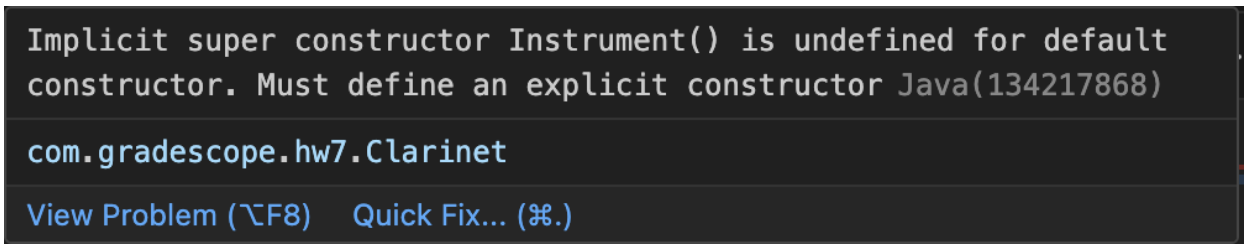
Step 1: Create the class

Create the file `Clarinet.java` and write the class stub (remember that for a class stub, you will declare the class with something like: `public class …` but the class will have nothing in it yet).

Reference the CS 60 HW5 - OOP in Java Resources tab for instructions on creating a new class file in VS Code.

Step 2: Add a constructor

Notice the error in the `Clarinet` class. When you mouse over the red squiggly, you'll see this error:



VSCode is telling us to define a constructor.

Create a zero-parameter constructor for `Clarinet`. It should use the superclass constructor and pass in `"Bb"` as the key. (That is, a new clarinet object has the key B-flat). Similar to Part A, you can click “Quick Fix” to create the stub, but you’ll still need to edit the constructor to fit the description.

Step 3: Create `Clarinet` test cases

Create the file `ClarinetTests.java` in which you will write JUnit test cases. Your tests should:

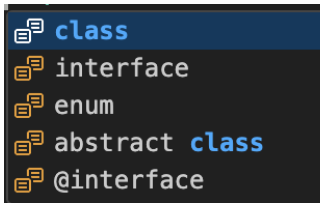
- Test all public methods in `Clarinet.java`, including inherited methods from `Instrument`
- Test the static type `Instrument` with dynamic type `Clarinet`. That is, test that you can have references of static type `Instrument` that refer to `Clarinet` objects.

Part C: Create `Squeakable` interface

Create a new `Squeakable` interface. Read the steps below for details.

Step 1: Create file and declaration

To create a new interface, the instructions are similar to creating a class. In VSCode, click "New File" and enter the name, in this case `Squeakable.java`, and then in the file, choose “interface” from the drop-down option list:



Alternatively, you can simply write the declaration the way you would with a class but replace `public class` with `public interface` instead.

The interface should require this method:

```java
public String squeak();
```

Step 2: Add documentation

Be sure to add doc comments to the interface and to the method. The `squeak()` method returns a string representation of the squeak.

Documenting an interface is essentially the same as documenting a class: We want to have a Javadoc comment for every method in the interface. Additionally, we want to have a top-level comment that describes the interface. As with classes, this Javadoc comment goes above the interface declaration, like so:

```java
/**
 * High-level description goes here
 *
 */
public interface Squeakable {
  ...
```

For a real-word example, you can check out the `Comparable` interface. But that is a gold-standard level of documentation; you do not have to be as detailed.

Part D: Implement `Squeakable` interface

Update the class declaration of `GuineaPig` and `Clarinet` to implement the `Squeakable` interface.

Step 1: Implement interface

If you have questions about how to implement an interface (or multiple interfaces), you might refer back to the lecture slides or do a quick web search on "Java interface example". Of course, you are also encouraged to post to Piazza or ask course staff, but we want you to get used to looking for information on the internet (and to practice identifying *quality* sources on the internet).

Then, let VSCode help make method stubs again, and fill in the body of the methods `squeak` as follows:

- When a guinea pig object squeaks, it returns:
  `"Squeak! (Feed me!)"`

- When a clarinet object squeaks, it returns:
  `"Squeak! (Need more practice...)"`

Be sure to add JavaDoc for the overridden methods that detail the message.

Step 2: Write `Squeakable` test cases

Add test cases to `GuineaPigTests.java` and `ClarinetTests.java`. Your tests should:

- Test the added `Squeakable` functionality with objects of static type `GuineaPig` and `Clarinet`.
- Test that you can have references of static type `Squeakable` that refer to `GuineaPig` and `Clarinet` objects.
- Test that references of static type `Squeakable` can be squeaked.

Part E: Submit files on Gradescope

There are two questions on Gradescope for this problem. The first asks for the `GuineaPig`, `Clarinet`, and `Squeakable` files; the other asks for the `GuineaPigTests` and `ClarinetTests` test files.

Double check you are not missing any files and everything compiles properly on Gradescope!

2 Problem 2: Refactor

CS 60 HW 7

Problem #7.2: Refactor Repetitive Code Using Inheritance

The **learning goal** of this problem is: practice writing code that uses inheritance.

Prerequisites

Java inheritance

Starter Files

The starter files you will reference for this problem are:

- `Table.java`
- `SnackTable.java`
- `HochTable.java`
- `SuperFriendlyHochTable.java`
- `TableTests.java`

These files are located in the `src/main/java` within the `com.gradescope.hw7` package.

Tasks

The tasks for this problem are:

- [ ] Submit `Table.java`, `SnackTable.java`, `HochTable.java`, and `SuperFriendlyHochTable.java` in Gradescope HW #7.2 Tables (4 files).

Make sure you are submitting the files in the correct assignment!

Instructions

Part A: Introduction

The starter code for this problem showcases BAD DESIGN! That is because the classes are closely related and contain duplicated code (both instance variables and methods) and do not take advantage of code reuse via inheritance, which could be used to streamline their functionality.

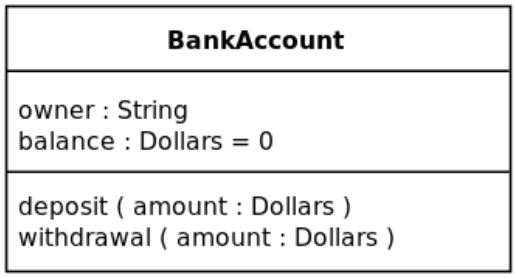
Your goal is to **refactor** this code to use inheritance. What is refactoring?

- Restructures existing code without changing its behavior
- Improves design, structure, and/or implementation while preserving functionality

How do we ensure that we are preserving functionality? Through an extensive test suite of course! Luckily, we have provided such a suite in `TableTests.java`.

- Before diving into the implementation of each class, we recommend that you take a look at the tests to try and understand the required functionality of each class.

Next, read through the starter files, and draw out a diagram showing the structure of each class. If you want to do it the way the pros do it, you can check out Unified Modeling Language (UML):



For example, the class diagram above shows a `BankAccount` class (top) with two fields (middle) and two methods (bottom). Notice how we specify types and parameter names. Though it is not shown in the example, you can even specify visibility (`private` / `protected` / `public`).

You will see many "bad code smells" (legit software term) as you progress in computer science. One is duplicated code—yuck! In this part of the assignment, your task is to remove the duplicated code/variables by moving them to one (or more) superclasses.

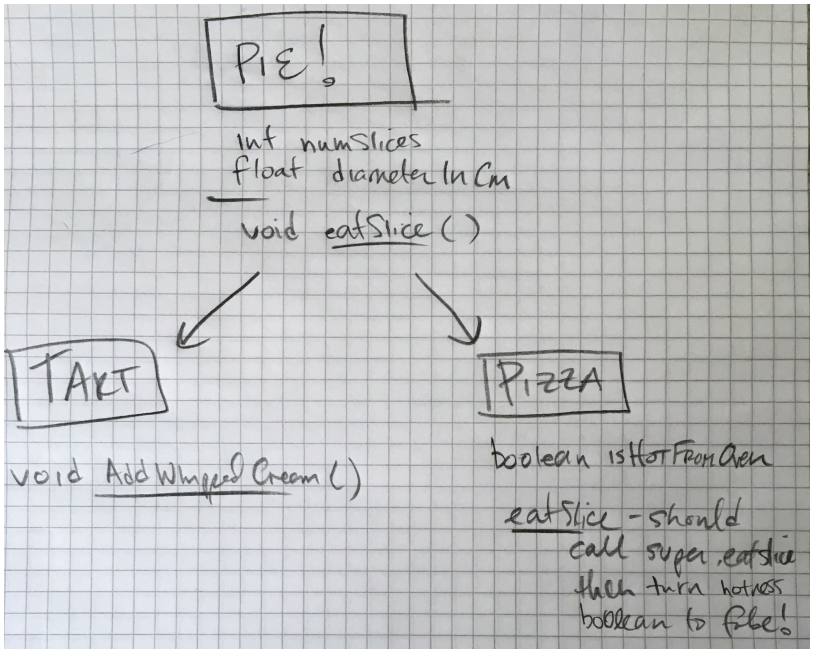
Things to think about as you carefully read through the starter files:

- Where will I put `extends`?
- Which instance variables are unnecessary/duplicated?
- Which methods are unnecessary/duplicated?

Part B: Plan Changes

Before changing any code, draw out a new diagram showing the different classes and their relationships *after refactoring*. This will help you set up your super/subclass relationships and move (or delete) the right code.

For example, if classes `Tart` and `Pizza` both inherit from superclass `Pie`, you might draw something similar to what you see below.[1] (Again, if you want to be fancy, UML has special arrows for showing relationships between objects. You are welcome but not required to use them.)



We recommend being explicit about whether fields and methods are `private`, `protected`, or `public`. Our general rule so far is that classes and methods are `public` and fields are `private` (though there are exceptions, e.g. `private` inner classes and helper functions). We have mostly been ignoring `protected` so far, but inheritance provides a great opportunity to use this access modifier! Check out Java's handy table (source).

Remember: **Do not change the tests! You should only need to make small modifications to the starter code!** The hardest part of this problem is thinking about how best to set up hierarchical relationships so that your code is as "condensed" as possible. If you are not sure, do ask for help before moving on.

---

[1] **Image description:** The diagram describes the classes, their fields and methods, and their inheritance relationships. The diagram has one box for each of the three classes: `Pie`, `Tart`, and `Pizza`. Inside each box is the name of the class. Arrows indicate that `Tart` and `Pizza` inherit from `Pie`. Associated with the `Pie` class are two fields (`numSlices` of type `int` and `diameterinCm` of type `float`) and one method (`eatSlice`, which takes no arguments and has return type `void`). Associated with the `Tart` class is one method (`addWhippedCream`, which takes no arguments and has return type `void`). Associated with the `Pizza` class is one field (`isHotFromOven` of type `boolean`) and one method (`eatSlice`, which should call the super `eatSlice`, then turn the `hotness` `boolean` to `false`).

Note: You do not need to turn in your diagram. But we are confident that drawing out the relationships between the classes will save you time in the next part!

Part C: Refactor Using Inheritance

Execute the plan you devised in Part B to rewrite the classes `Table`, `SnackTable`, `HochTable`, and `SuperFriendlyHochTable` using inheritance. The classes should behave exactly as they did in the starter version.

Hint: Refactor *one thing*, then test using `TableTests.java`. That way, you can see when something breaks and fix it (or revert it) before moving on!

Unfortunately, there is not an automated way to make sure that you have (1) removed all duplicate instance variables and (2) removed all duplicate code by calling parent methods. So you will have to work carefully and double-check these yourself!

- Companies use code reviews to do this double-checking. Finding and automatically removing code smells is also an active area of research!

- If you have been clicking around VSCode, you might notice that VSCode has a "Refactor" option when right-clicking within a file. We suggest **not** using it for this assignment — using it without a deeper understanding of what it is doing will do more harm than good.

Hint: Commonly, you might want a child class's method to (a) do what the parent's version of the method did, then (b) do something else specific to the child class. Remember that you can always explicitly call the parent's version of the method using the keyword `super`. To make that more concrete, here is an example using `Pie` and `Pizza` from the diagram above:

Superclass (parent class):

```java
class Pie {
    int numSlices = 8;

    public void eatSlice() {
        this.numSlices--;
    }
}
```

Subclass (child class):
(calls the parent version, then does something more)

```java
class Pizza extends Pie {
    boolean isHotFromOven;

    public void eatSlice() {
        super.eatSlice();
        this.isHotFromOven = false;
    }
}
```

Part D: Submit files on Gradescope

Submit all of the table files on Gradescope (without the tests). This includes: `Table.java`, `SnackTable.java`, `HochTable.java`, and `SuperFriendlyHochTable.java`. Do not include `TableTests.java` in your submission!

Double check you are not missing any files and everything compiles properly on Gradescope!

💯 Grading Rubric

CS 60 HW7

Assignment Rubric

We want to be as transparent as possible in our grading system. We'll include a grading rubric for each homework assignment:

#	Name	Autograder	Functionality	Style	Testing	Total
1	GuineaPig, Clarinet, Squeakable	21	0	4	0	25
1	GuineaPigTest & ClarinetTest	0	0	3	17	20
2	Tables	10	20	5	0	35
		39.00%	25.00%	15.00%	21.00%	80

In addition to the list from previous assignments, we will check for the following elements of **bad style**:

- Class Design
  - Mark methods with `@Override` when (1) a class method overrides a superclass method, (2) a class method implements an interface method, and (3) an interface method respecifies a superinterface method.

- Tests
  - Put `@Test` on its own line before all test functions. Make sure to use `import org.junit.jupiter.api.Test;` as the import (which uses JUnit 5).

  - It is generally better practice to test one functionality per test method! This potentially means multiple tests for a single function. This approach isolates exactly what functionality is failing without having to retest multiple times (and helps prevent cases in which we fix one bug only to discover another).