DUE DATE: MARCH 13TH, 2020 AT 11:59PM

Instructions:

- Read the **`SubmissionProcedures.PDF`** and the **`Standards.java`** carefully BEFORE you start.
- Let me know about any typos, clarifications or questions you may have right away. Don't wait.
- Read the whole assignment before you start.
- Start right away. The sooner you get stuck, the sooner you can get help to get un-stuck.

Assignment overview

Learning outcomes:

- Learning to extend your previous assignments
- An understanding of object hierarchies and use subclasses to extend other classes.
- An understanding of how to draw simple graphics using StdDraw.
- Familiarity with StdDraw to create basic animation driven by user input.

Additional Notes:

Read **Standards.java** and **SubmissionGuidelines.PDF** Carefully & before you start.

Several Classes will be provided as a starting point. You should have a look at **Vector3.java**, **InputObject.java** and **DrawingTest.java** as you will need them for the assignment and I will reference them in class examples. You can either continue from your own solution to A2 (I will give a couple of bonus marks for this if you can adapt your current code to work) or start from the handout.

As before, you should be tracking your project in .git and using all the best practices we have been discussing so far.

Please provide detailed commenting and commit your code often!!!

Note: Invest in a flash drive or otherwise back up your work. There have been several cases of hard drives crashing lately so learn from your unfortunate classmates. It is a very mysterious circumstance and you cannot be too careful. I also sometime email the files to myself or transfer them using a service like wetransfer which will keep them around for a short time.

You also may want to make backups of your working files as you go if you are unsure about git. You can take the whole folder (**`.git`** file included) and put it in a **`.zip`** file if you are worried about breaking things.

Please come speak with me if you have any questions about any of these topics.

Git Tips:

- Do not commit any file that should not be there. It is harder to remove files that have been added then to not add them in the first place. The commit is meant to be a reflective phase where you review your work. You CANNOT UNDO a commit and anything that gets put in will be in the record forever. This is a feature not a bug.

- Each git commit should include the Phase and a clear and complete summary of what code was added.

- If you find a bug, include that bug fix as its own commit. You should include the specific error (eg. Null reference) and how you fixed it. This will be a useful reference when you encounter the same error again.

- You may need to enable hidden files in your Windows/Mac file explorer in order to see the **`.git`** and **`.gitignore`** files.

Tips:

Start early. Read the assignment fully before beginning. We will go through it in the first class after it has been handed out. That is your chance to ask questions

Seriously, start early. I know you are busy but it is much easier if you get stuck early and can get help troubleshooting.

Notes on Academic Misconduct:

Your code should be done entirely by you. Any suspicion of Academic Misconduct or plagiarism will be fully investigated and can lead to serious academic penalties including 0 on the assignment for a first office, or an F in the course. Third offences could lead to expulsion from ICM. If you have any questions as to what qualifies as Academic Misconduct, please discuss it with me.

Phase 0: Introduction

Before you start.

I am providing a framework of the methods here with the understanding that you will create a **`main`** method and a **`TestClass`** for testing all of your methods as you progress.

Phase 1: Thing Object

Subclasses and polymorphism: Base Object

We will focus on extending the **`Thing`** Object first. You will add several variables to the **`Thing`** class you made in A2 and we will later subclass it into several variations. Add the following variables and methods (if you don't have them already).

```
String name
String description
```

In kilograms
```
double weight
```

In your local currency back home
`double value` (or **`int`** if that makes more sense)

Also add a **`double totalWeight()`** method. In the **`Thing`** object this just returns the **`weight`**, however we will override it later.

Phase 2: Draw Things

StdDraw and parent class methods to override later.

Add the following variables and methods which will draw the **Thing** to the GUI using **StdDraw**. This will work very much like the **Person** class did but we will update and modify it later.

Variables

A coordinate position in the "world map". We will use values between **[0..1],** inclusive for each of the **x** and **y** values. If you don't want to use the Vector2 class, you can use 2 doubles but you will have to do the math yourself for distance, normalization and other calculations.
`Vector2 position`

The file to draw, such as a `Person0.png.` You can create your own png files using any art program (such as MS Paint)
`String drawFile`

A flag for whether this `Thing` should be drawn or not.
`boolean isDrawn`

Methods

Draws the art at the appropriate position using the file reference provided. Should check the `isDrawn` flag and only draw if it is `true`.
`void draw()`

Optional Upgrades

a) Optional – You might want to have additional scale variables or other control variables as needed.
b) Optional + Ambitious: - You also can rescale the entire window number system in `StdDraw` so that you aren't constrained to its default scale.
c) Optional + Very Ambitious: If you want to make your art sizes more intuitive, you can record the size in real world scale (such as meters) and translate it to a local scale with a multiplier variable (ie 1m = .01 screen size units or whatever your conversion factor is).

Phase 3a: Thing - Subclasses and polymorphism

Add a **`void update()`** method to **`Thing.`** This method does nothing for now, however we will override it later. You should keep an **`ArrayList`** of all **`Things`** created in your **`main`** method and call each **`Things update()`** method once per frame from your main **`update`** loop. This will be used later in order to create animation over time.

Called each frame
`void update()`

GameController

Create a new **`GameController`** class that includes a **`main`** method. This will be your main starting point for execution. In it, you should create an **`ArrayList allThings`** that contains all Thing objects that you have created. They will be stored as an **`Object`** in the **`ArrayList`** by default so you will need to cast it back to a **`Thing`** in order to access the methods you need. I have provided pseudocode below that shows what your **`main`** method should contain.

```
// Create some different types of Things for testing
// Add them all to the allThings ArrayList

// update loop logic in main
while(true){
	// iterate through each Thing in the scene, calling update() on each Thing
	// iterate through each Thing in the scene, calling draw() on each Thing
	// wait for specified time. 1/frameTime is your frame rate.
	StdDraw.show (frameTime);
}
```

Phase 3b: A Thing with a quantity.

Create a **`Thing`** subclass called **`QuantityThing`** which is used for a **`Thing`** with an **`int`** counter such as a bag of apples or a wallet holding money.

Variables:

Unit of measurement ("Apples", "Dollars", etc)
`String measurement`

The weight of one unit of the stored item (eg the weight of a single apple).
`double weightPerUnit`

The quantity of what is stored (for example, the number of apples or dollars).
`int units`

The maximum capacity. Anything added beyond this value should be refused.
`int maxCapacity`

`Methods:`

This returns the total **`weight`** of this **`Thing`** plus all the items it contains (ie the container itself as well as all items in the container). This can be calculated by adding the base **`weight`** (its own **`weight`**) to the number of **`units`** * **`weightPerUnit`**.

`double totalWeight()`

The number of possible spots left
`int remainingCapacity()`

Create methods to **`add`** and **`remove int`** amounts from the **`QuantityThing`**. These should first check the **`units`** available and the **`maxCapacity`** and reject the transaction if there is not enough space (adding) or not enough to take (removing).

Phase 4: ArrayLists

A Thing to hold other Things

Next, you will create a **ContainerThing** class. It will also be a subclass of **Thing** and is a **Thing** that can store other **Things**. Examples of real life objects that this could represent include a chest of drawers, a suitcase or an inventory.

Use an **ArrayList** to store any kind of **Thing** as an **Object**. The **ArrayList** itself should be storing **Objects** and you will need to convert them back to **Thing** when you return them (such as from the **remove()** method).

> A method that returns the **weight** of the **Thing** itself plus all **Things** inside itself. Should be calculated dynamically as needed.
>
> **double totalWeight()**
>
> The maximum combined weight capacity of all **Things** stored in itself. Can't add a new **Thing** if it causes the current weight to climb above this value.
>
> **double maxWeightCapacity**

The next step will be to add methods to add and remove **Thing Objects** from the **ContainerThing**. You should use the built in **ArrayList add** and **remove** methods inside your own **addThing** and **removeThing** methods, while adding additional verification (eg. by **weight**).

Methods:

> Add a given **Thing** (the parameter) to the **ArrayList** if there is enough space remaining (the **newThing weight** plus current (base **Thing**) **weight** is less than the **maxWeightCapacity**). Use the **ArrayList** version of the **add** method to store the **Thing** as required. Return **true** if it was added successfully (including checking the **weight** requirements) and **false** otherwise.
>
> **boolean addThing(Thing newThing)**
>
> Returns the **Thing** that was removed or **null** if it could not be found. This should use the **ArrayList remove** method (which will in turn use the **Object equals** method to check if it exists).
>
> **Thing removeThing(Thing toRemove)**

A **Thing** in the **ContainThing ArrayList** should not have their art drawn (since it is in a container). Set the **isDrawn boolean** variable to **false** for any **Thing** that is added to the **ArrayList** and to **true** when it is again **removed**. Remember, you will first have to cast the type back from **Object** to **Thing** to access this variable.

Phase 5: Movement

Finally, something interesting

`AnimatedThing`

The **`AnimatedThing`** will update its position variable by a given displacement each frame. It will do this by storing a **`destination`** coordinate and updating its current **`position`** to move towards that destination by a small amount each time the **`update`** method is called.

You might need more variables then noted (for example it can be useful to keep a **`boolean`** value of whether it is currently moving).

> The point this **`AnimatedThing`** is moving towards.
> **`Vector2 destination`** (or again, 2 doubles if you prefer)
>
> Distance to move per frame OR you can store it as distance per second and convert it to frame by dividing 1/time per frame (with the frame time set by the **`show(int milliseconds)`** call to **`StdDraw`**).
> **`double movementSpeed`**
>
> Set a new position for this object to move towards.
> **`boolean setDestination(Vector2 position)`** // or 2x doubles

The easiest way to handle the movement speed is to normalize the directional vector then multiply it by the speed (or fraction of the speed per frame). The Vector2 class can be helpful or you are free to figure out an alternative way if you wish (and like vector math). You can get a vector pointing from the position to the destination by subtracting the position from the destination. The Vector2 class can help with normalization.

It should also detect when it is close to its destination and stop (or find a new destination to move towards).

For now **`AnimatedThing`** can just move back and forth. It does not have to be complicated.

Phase 5b. The Player Controls

Its Alive!

Make a **`PlayerThing`** that extends **`AnimatedThing`**

Take input from the **`InputObject`** and translate it into an animated movement. Start on one axis (ie X) and get that working first. You will probably want to create some additional methods to make working with it a little easier (rather then using the **`boolean array`** provided in the example). You may even want to restructure how the whole class works. I personally like to use a **`Vector2`** for input as it makes the translation to movement much easier. There is an example of animation in the demo code from **A2** (the first **DrawingTest.java** demo I gave you). You should be able to move around the map, controlling your player with the arrow keys on the keyboard.

Phase 6: Bonus (and Optional)

So you want more work, hmmm?

Optional feature: **Collision Detection**

Implement collision detection between the **`PlayerThing`** and all the **`Things`** currently drawn. There are several methods to do this and the **`Vector2`** library should provide some shortcuts with the math.

Hand in instructions:

Place all your **`.java`** files, the **`.git`** repo and the **`.gitignore`** files in a single .zip file with the name LASTNAME_FIRSTNAME_A2.zip (NOT RAR or other archive formats) in the hand-in folder on Moodle with your **`readme.txt, honesty declaration and test_output.txt`**

DO NOT:
- DO NOT hand in **`.class`** files
- DO NOT hand in Standards.java or other unused or unneeded files.
- DO NOT hand in **`bluej`** files or other various editor files **`(.vs, .proj, etc).`**
- DO NOT ignore the hand in instructions.

Congratulations, you are done. Be proud of yourself and relax.

Hand In Additional Notes:

- It is especially important that your classes and methods are very accurate. Make sure your class names are **`Uppercase`** and your **`methodNames`** are lowercase and don't contain typos. **If your program doesn't compile for the marker you will lose significant marks**, even if it works on your machine.

- Also be sure you remove any package headers added by IntelliJ or Eclipse etc as these will break compilation.

- See the **`SubmissionGuidelines.pdf`** document for full instructions. See the **`Standards.java`** document for formatting guidelines. Don't hand in the **`Standards.java`** file.

- Include a **`README.txt`** file when you hand in the assignment to tell the marker which phases were completed, so that only the appropriate test can be run. This can also include messages about errors you are encountering.

- For example, if you say that you completed Phases 1-3, then the marker will compile your files to check for completion of those phases. If it fails to compile and run, you will lose all of the marks for the test runs (at least half of the assignment). The marker will not try to run anything else, and will not edit your files in any way.

- Submit a completed Honesty Declaration with EACH assignment. Your assignment will not be marked without it. I will provide an easy text based version you can use so you don't have to mess around signing PDFs.

Good luck! Don't Panic!