COMP 1406
Fall 2022 - Tutorial #2

Objectives

- Learn how to create a hierarchy of classes
- Gain experience using inherited attributes and behaviour
- Work with abstract classes and interfaces

Getting Started:

Download the **Tutorial 2.zip** file from Brightspace. Extract the zip file contents and open the folder as a project in IntelliJ. Run the test code found in the GameTestProgram class. You should see the output shown below.

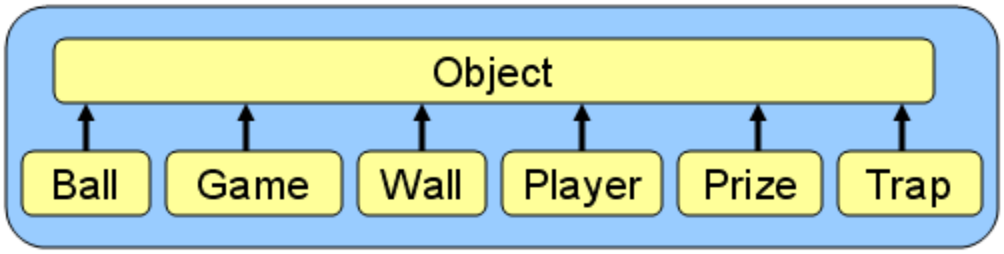
```
Here are the Game Objects:
Wall at (0,0) with width 10 and height 200
Wall at (10,0) with width 170 and height 10
Wall at (180,0) with width 10 and height 200
Wall at (10,190) with width 170 and height 10
Wall at (80,60) with width 100 and height 10
Wall at (10,90) with width 40 and height 10
Wall at (100,100) with width 10 and height 50
Prize at (165,25) with value 1000
Prize at (65,95) with value 500
Prize at (145,165) with value 750
Trap at (125,35)
Trap at (145,145)
Player Blue Person at (38,156) facing 90 degrees
Player Yellow Person at (55,37) facing 270 degrees
Player Green Person at (147,116) facing 0 degrees
Ball at (90,90) facing 0 degrees going 0 pixels per second
```

1) The project begins with files that are used to represent a ***Game*** of "ball tag" in which a ***Player*** (who is "*it*") chases other players around a ***Wall***ed environment with a ***Ball*** and tries to hit them with it. When a player gets hit with the ball, they become "*it*" and try to throw the ball at the other players. Each time a player hits another they gain points and when a player gets hit, they lose points.

Trap
Prize
Ball
Player

Assume that the environment has rectangular walls that prevent movement in certain directions. Also assume that there are ***Traps*** that a player cannot run into, otherwise they lose points. Lastly, assume that there are ***Prizes*** that the players can collect for bonus points while running around.

Some classes for this game have been developed and they form the class hierarchy shown below:



A. Examine the classes and look at the attributes (i.e., instance variables). You will notice that there is a lot of duplicate information. For example, **Ball**, **Wall**, **Player**, **Prize** and **Trap** all keep track of a **location**, and **Ball** and **Player** both have a **direction** and **speed**. This is not very nice.

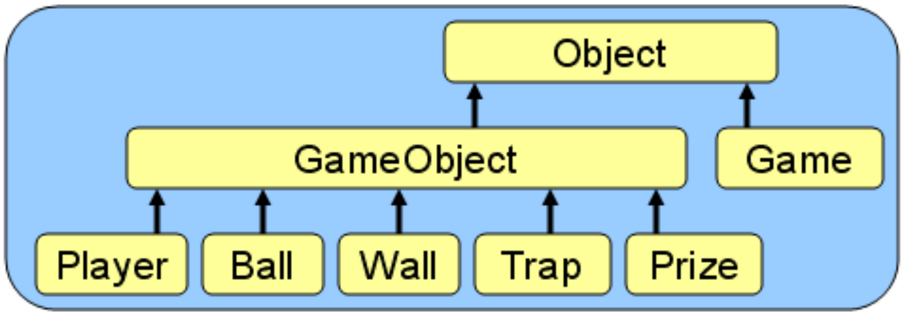
B. To eliminate the duplication, we will re-organize the classes by abstraction (i.e., by extracting the common attributes and behaviors). One way of doing this would be to see what all objects have in common. It seems that all objects except **Game** have a **location**. We will begin by getting that information out from the objects and storing it in a common class which they can all inherit from. We will start doing this in the next problem.

---

2) Create a class called **GameObject** that will store a single **private** attribute called **location** of type **Point2D**.

A. Create a constructor in **GameObject** that takes an initial **Point2D** location and sets it.
B. Remove the **location** variable from the **Ball**, **Wall**, **Player**, **Prize** and **Trap** classes. You will notice that none of these classes will compile now since the constructors, get/set methods and **toString()** methods are all using that **location** attribute that we just removed.

C. Adjust the **Ball**, **Wall**, **Player**, **Prize** and **Trap** classes (but not the **Game** class) so that they become subclasses of **GameObject** (i.e., simply append **extends GameObject** after the class name is defined). Here is the new hierarchy:



Now all the subclasses of **GameObject** inherit the **location** attribute that we added to the **GameObject** class. However, because we made that attribute **private**, if you browse around the subclasses (i.e., **Ball**, **Wall**, **Player**, **Prize** and **Trap**), you will notice that **location** is highlighted in red. That means that we cannot access it directly. We are still inheriting it though, so each of the **Ball**, **Wall**, **Player**, **Prize** and **Trap** classes actually do have a location. We just cannot access it. However, in general, it is usually safe to allow subclasses to access superclass attributes.

D. Go back to the **GameObject** class and change the location attribute to be **protected** instead of **private**. Subclasses may access and modify protected variables inherited from their parent class. Browse the subclasses (i.e., **Ball**, **Wall**, **Player**, **Prize** and **Trap**) and you will notice that **location** is now accessible, as it is no longer shown in red.

The subclasses still have an error though. If you try to compile the **Ball**, **Wall**, **Player**, **Prize** and **Trap** classes, you should notice an error in each of the subclasses:

```
Error:(7, 44) java: constructor GameObject in class GameObject cannot be
applied to given types;
  required: Point2D
  found: no arguments
  reason: actual and formal argument lists differ in length
```

This error is indicating that JAVA is trying to find a zero-parameter constructor in the **GameObject** class. It is doing this because in the constructors for **Ball**, **Wall**, **Player**, **Prize** and **Trap** classes, JAVA is trying to first do **super()**, which means it is trying to call **GameObject()**, which does not exist. As it turns out, whenever you write a constructor, there is automatically an implicit call to the super class's zero-parameter constructor. In our case, the **GameObject** class has a 1-parameter constructor, so things don't work.

E. To change this default behavior, replace the code `location = loc;` with `super(loc)` on the first line of your constructors in the **Ball**, **Wall**, **Player**, **Prize** and **Trap** classes. Now Java will use this new constructor call and not attempt to call the one that does not exist. The classes should now compile.

Basically, what we just did was move the initialization code for the **location** attribute into the **GameObject** class. We are "calling" this code from the other 5 constructors through **super(loc)** and passing it the initial **loc** that we want to use. If you do not understand what happened, ask a friend and/or a TA.

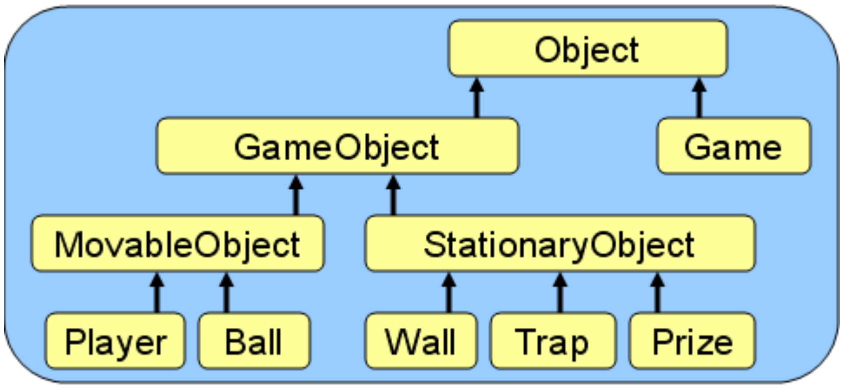
Now since we have moved the location attribute to **GameObject**, we should also move its *get/set methods* into that class as well.

F. Remove all of the **getLocation()** and **setLocation()** methods from the **Ball**, **Wall**, **Player**, **Prize** and **Trap** classes and place a single copy of them in the **GameObject** class.

G. Run the test program to make sure that it still works. At this point, 5 of our classes are now successfully inheriting the **location** attribute.

---

3) Notice also that both the **Player** and **Ball** classes have a **direction** and **speed** attribute. That means, they have something in common - they are both kinds of **GameObjects** that can move around. To avoid duplication, we will want to use inheritance again. We are going to abstract out even further resulting in more changes to the hierarchy.



A. Create two subclasses of **GameObject** called ***MovableObject*** and ***StationaryObject***. Then adjust the hierarchy so that **Player** and **Ball** extend (i.e., inherit from) ***MovableObject*** while **Wall**, **Trap** and **Prize** extend ***StationaryObject*** (none of the classes will compile yet though):

B. Remove the **direction** and **speed** attributes from **Player** and **Ball** and copy them into **MovableObject**. Make them **protected** instead of **private**. Remove the duplicated methods from **Player** and **Ball** and place them into the **MovableObject** class too. Don't worry that they don't compile yet, as things are still a bit messed up.

C. In **MovableObject**, create a single constructor that takes a direction **d**, speed **s** and location **loc** as its 3 parameters (in that order) and in the first line it should make the call **super(loc);** to its superclass' constructor. Make sure that your **MovableObject** class now compiles.

D. Adjust the first few lines of the **Player** and **Ball** constructors in order to get the code to compile by calling the new constructor that you made in **MovableObject** which now takes 3 parameters, not just the one. When making your changes, think about which initial direction, speed and location that you want to pass as parameters - are they fixed values or parameters? Your **Ball** constructor should have 2 lines remaining in it once you are done and your **Player** constructor should have 5 lines. Both **Ball** and **Player** should now compile successfully.

E. Add a constructor to the **StationaryObject** class which calls **super(loc)** to simply set the initial **location**. Although there are no shared attributes between **Wall**, **Trap** and **Prize**, we will still make them inherit from **StationaryObject** in case we need to add additional shared attributes or behaviors in the future. The **StationaryObject**, **Wall**, **Trap** and **Prize** classes should all compile. Make sure that they do before you continue.

---

4) Now we will complete our abstraction process by making some of the classes **Abstract**. In our program, we do not want to make any instances of **GameObject**, **MovableObject** or **StationaryObject**. Instead, we will force everyone to make instances of the more specific objects of **Ball**, **Wall**, **Player**, **Prize** and **Trap**. To do this, we will make **GameObject**, **MovableObject** and **StationaryObject** to all be **abstract** classes. But first, let's just test these classes before we move on.

A. Add the following lines of code in the **GameTestProgram** (just before the walls are added) which will create instances of **GameObject**, **MovableObject** and **StationaryObject**.

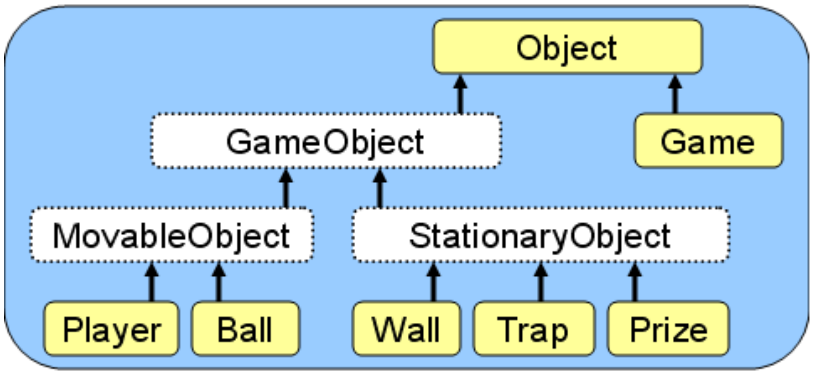
Run the code and then make sure that you notice the three new objects created:

```
// Simple Test
g.add(new GameObject(new Point2D(0,0)));
g.add(new MovableObject(0, 0, new Point2D(0,0)));
g.add(new StationaryObject(new Point2D(0,0)));
```

B. Add the keyword **abstract** before the word **class** at the top of the **GameObject**, **MovableObject** and **StationaryObject** class definitions.

Here is what we did

------>



The **GameTestProgram** will have three errors similar to this one shown here:

```
Error:(11, 15) java: GameObject is abstract; cannot be instantiated
```

We get the above error because we are no longer allowed to *instantiate* (i.e., create) any **GameObject** instances, because they are now **abstract**. Similar errors occur for **StationaryObject** and **MovableObject**.

C. Remove these 3 lines of test code from the test program and the code should compile and run as before. Remember, making a class **abstract** does not change the way it *works*, it just means that you can no longer *create instances* of that class directly.

---

5) Now we will make behavior for our objects to allow them to update themselves.

A. Create the following **abstract** *method* in the **GameObject** class and then recompile it:

```
public abstract void update();
```

Try recompiling the **Wall**, **Player**, **Ball**, **Prize** and **Trap** classes. You will get errors like this:

```
Error:(3, 8) java: Wall is not abstract and does not override abstract
method
                    update() in GameObject
```

This is Java's way of telling you that the class did not implement the **update()** method and that it is "supposed to" since it inherits from **GameObject** and we defined an *abstract method* there. Recall that all non-abstract subclasses MUST implement all abstract methods defined in their superclasses, unless there is one that is being inherited.

B. Implement this **update()** method in the **StationaryObject** class but leave the body of the method blank (i.e., do nothing because stationary objects do not need to update since they do not move). You will notice right away that **Wall**, **Prize** and **Trap** classes compile now because they inherit this "blank" **update()** method.

C. Write the following **update()** method in the **MovableObject** class, which will allow **MovableObjects** to move forward and redraw themselves whenever update is called (don't compile yet):

```
public void update() {
    moveForward();
    draw();
}
```

D. Write the following **moveForward()** method in **MovableObject** now. It will move the object to its next location based on its **speed**, **direction** and current **location**:

```
public void moveForward() {
    if (speed > 0){
        location = location.add((int)
(Math.cos(Math.toRadians(direction)) * speed), (int)
(Math.sin(Math.toRadians(direction)) * speed));
    }}
```

E. Create the following **abstract** method in **MovableObject** and then re-compile the class:

```
public abstract void draw();
```

What have we just done? We set our code up so that all **MovableObjects** will update their **location** by moving ahead in the **direction** that it is facing by an amount indicated by the **speed.** Don't worry about the math here. By making the **draw()** method **abstract**, this means that the subclasses **Player** and **Ball** must each implement the method in their own unique way. Of course, in reality we would need to do much more in the **moveForward()** method, such as checking for collisions and whether or not we fell into a trap or found a prize, but for this tutorial we will not go that far.

The code in **MovableObject** now compiles, but the **Player** and **Ball** classes will not because we need to implement a **draw()** method in those classes.

F. Create a simple **draw()** method in these two classes that simply displays (using **System.out.println()**) the **location**, **speed** and **direction** of the object in a format like this:

```
Player is at (110,100) facing 0 degrees and moving at 10 pixels
per second
```

Of course, in a real game this method would actually draw the player or ball on the screen.

Hopefully you see the advantage of abstract classes and methods. If we now decided to add another kind of movable object, we would inherit attributes and behavior for free. All we would need to do is write a **draw()** method and the other methods we would inherit the default behaviour for (e.g., **update()** and **moveForward()**).

---

6) Now we will make a **Ball** and **Player** move in the game.

A. In the **Ball** class, override the default inherited **update()** method from the **MovableObject** class by writing your own **update()** method that causes the ball to decelerate (i.e., slow down) after moving forward. That is, upon updating, **Ball** objects should move forward, then decrease their speed by one and then draw themselves. Make sure that the speed never becomes negative though.

B. Test your code by adding the following lines to the end of your **GameTestProgram**, which will test the updating of the **Player** and **Ball** objects:

```
// Test out some Player and Ball movement
System.out.println("----------------------------------------------");
Player player = new Player("Red Player", Color.RED, new Point2D(100,100),0);
player.speed = 10;
player.direction = 0;
g.add(player);
```

```
Ball ball = new Ball(new Point2D(100,100));
ball.speed = 10;
ball.direction = 0;
g.add(ball);

player.update();
player.update();
player.update();
ball.update();
ball.update();
ball.update();
```

If all is working, you should see something like this as the last 6 lines of your output (depending on how you wrote your **draw()** method):

```
Player is at (110,100) facing 0 degrees and moving at 10 pixels per second
Player is at (120,100) facing 0 degrees and moving at 10 pixels per second
Player is at (130,100) facing 0 degrees and moving at 10 pixels per second
Ball is at (110,100) facing 0 degrees and moving at 10 pixels per second
Ball is at (119,100) facing 0 degrees and moving at 9 pixels per second
Ball is at (127,100) facing 0 degrees and moving at 8 pixels per second
```

Notice that the player and ball both move to the right (because direction is 0) but that the ball slows down after each update.

---

7) In the test code that we just added in part 6, we specifically called **update()** 3 times for the player and 3 times for the ball. We will now do the updating automatically.

A. In the **Game** class, add a method called **updateObjects()** that iterates (i.e., use a **FOR** loop) through the objects and updates all of the **GameObjects**. Your code should be very simple and similar to the **displayObjects()** method in regards to the format. You may notice that the code will not compile. That is because the **Game** class keeps an `Object[]` of *general* **Objects**. Change this type to `GameObject[]` so that it stores a list of **GameObjects** instead of just **Objects**. You will also need to make changes at a few more places in the **Game** class code. Make sure that your code compiles.

B. Remove the last 6 lines of your **GameTestProgram** (i.e., the **update()** lines that we added) and replace it with the following code:

```
// Make some updates
for (int i=0; i<20; i++)
    g.updateObjects();
```

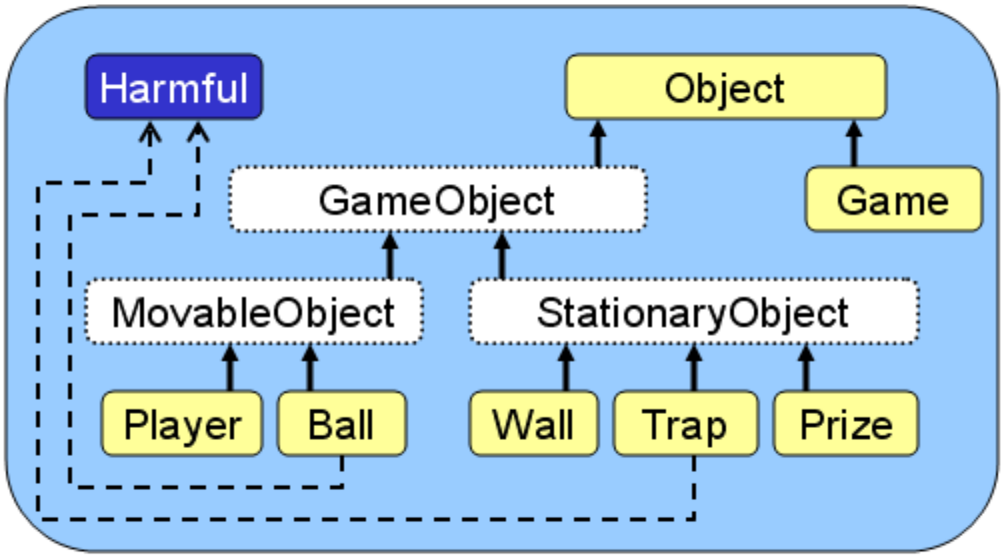
C. Comment out the code that adds the blue, yellow and green players as well as the first ball at (90,90) so that your output will be more clear. Run the code. You should notice that the **Ball** eventually stops but the **Player** keeps moving. Here is the final two lines that you should see in the output:

```
Player is at (300,100) facing 0 degrees and moving at 10 pixels per second
Ball is at (155,100) facing 0 degrees and moving at 0 pixels per second
```

You may not have realized that your code actually updates the **StationaryObjects** as well, but that these don't really do anything when updated, hence you don't see anything printed out for the walls, traps and prizes.

---

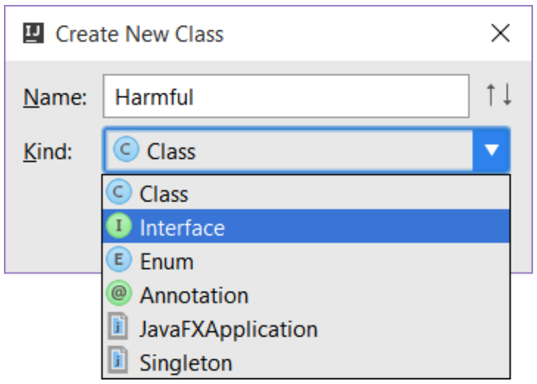
8) Now we will add functionality to allow some game objects (i.e., **Ball** and **Trap**) to be harmful to the players of the game. We will do this by making use of a JAVA ***interface*** definition called **Harmful**. This interface will be implemented by all objects that are "harmful" to the players. That means, we will be able to ask an object how harmful it actually is by asking for its damage amount so that we can update our score.



A. Create an interface called **Harmful**. To do this, create a new Java class as you normally would, but replace the 'class' keyword with 'interface'. Add the method signature for the getDamageAmount method, which should return an int.

Your **Harmful** interface code should look like:

```
public interface Harmful {
    public int getDamageAmount();
}
```



Name: Harmful

Kind: Class

B. Add a minor change to the **Ball** and **Trap** class definitions so that they both implement this interface (i.e., add `implements Harmful` to the class definition). Once you do this, these two classes will not compile. That's because in order to *implement an interface*, a class must have code for the methods in that interface. The **Ball** and **Trap** classes are now forced to write a method called **getDamageAmount()** and your code will not compile until you do so.

C. Write a **getDamageAmount()** method in the **Ball** class so that balls cause a **-200** damage amount (i.e., just return **-200** when asked for the damage amount) and in the **Trap** class so that traps have only a **-50** damage amount. You must put the word **public** before the method's return type, since interfaces require us to write publicly accessible methods.

D. Write the following method in the **Game** class which returns an array of all objects that implement the **Harmful** interface:

```
// Return an array of all Harmful objects in the game
public Harmful[] harmfulObjects() {
    // First find out how many objects are Harmful
    int count = 0;
    for (GameObject g: gameObjects)
        if (g instanceof Harmful)
            count++;

    // Now create the array and fill it up
    Harmful[] bad = new Harmful[count];
    count = 0;
    for (GameObject g: gameObjects)
        if (g instanceof Harmful)
            bad[count++] = (Harmful)g;
    return bad;
}
```

Examine the code to make sure that you understand it. The one aspect of the code that should be *new* to you is the type-casting of **g** to **(Harmful)** when we add it to the **bad** list. This is necessary in order for the compiler to allow you to put general **GameObjects** into a list that expects only **Harmful** objects. What we are trying to do is valid because we just checked (by means of the `instanceof`) that the object was indeed **Harmful**, but the compiler does not realize that we did this "check" on the previous line.

E. Add the following to the end of your **GameTestProgram** class and compile/run it:

```
// Get the harmful objects
System.out.println("\nHere are the Harmful Objects:");
Harmful[] dangerousStuff = g.harmfulObjects();
for (Harmful d: dangerousStuff)
    System.out.println("  " + d);
```

You should see the following:

```
Here are the Harmful Objects:
  Trap at (125,35)
  Trap at (145,145)
  Ball at (155,100) facing 0 degrees going 0 pixels per second
```

Now that we have the harmful objects, we can search through them and ask each one of them what damage amount they have.

F. Write a method in the **Game** class called **assessDanger()** that uses the **harmfulObjects()** method that you just wrote and returns the integer total sum of damage amounts for all the harmful objects. When you write the code that iterates through the harmful objects, you should notice that you do not need to know what kind of actual object it is (i.e., **Trap** or **Ball**). Instead, you just need to call the **getDamageAmount()** method for the object.

Interestingly, you will notice as well that we did not create a **damageAmount** attribute for these **Harmful** objects. We could not create such a shared attribute without altering the class hierarchy. Why? Ask a friend and/or a TA if you do not know. Of course, by having the **getDamageAmount()** method, which returns an appropriate value, we kind of "fake" having such an attribute.

G. Add the following to the end of your **GameTestProgram** class and compile/run it:

```
// Assess the current amount of danger
    System.out.println("\nCurrent Danger Assessment:");
    System.out.println(g.assessDanger());
```

You should see a value of **-300** (from the one ball and two traps in the game).

Submission

Zip your completed tutorial #2 project and submit your **tutorial2.zip** file to the Tutorial #2 submission on Brightspace. Make sure you download and test your submission after you have submitted. Submitting a corrupt zip or a zip file that does not have the correct files will result in a loss of marks.