

Thanks to Dominic Rosato for creating the original version of this assignment.

BACKGROUND

An *abstract class* is designed specifically to be inherited, and it is used for a specific situation: when you aren't providing the implementation of a method (called an *abstract method*), but want to require that that specific method(s) be implemented by a subclass. An abstract class can still have concrete instance variables, non-abstract methods, and even a constructor, but we will not be able to construct an object of the abstract class. *We can only construct objects of subclasses that inherit from the abstract class.*

OBJECTIVE

To learn about abstract classes by writing a racing simulation, in which three different kinds of moving objects will compete to see who first crosses the finish line.

PROCEDURE

1. Complete the `AbstractRacer` class using the reference code given below.
2. Develop three subclasses that inherit from the `AbstractRacer` class. The `Tortoise` (“slow and steady wins the race”), a `Hare` (fast, but quickly runs out of energy and needs to rest), and some other subclass of `AbstractRacer` that has its own particular motion. Each of these characteristics are coded in the `move()` method for that class.
3. Write a `Race` class that manages a race. It should have an `ArrayList` of `AbstractRacer` objects that it manages, an `update()` method that triggers a move for every `AbstractRacer` when called, and any other methods that would be appropriate.
4. Finally, write a `RacerRunner` class with a `main()` method that creates the racers and a race event and then runs the simulation.
5. As you test the simulation, you can make the race more interesting by defining move characteristics such that a different `AbstractRacer` may win each time the simulation is run. Experiment with your classes! The different characteristics of a *Champion* in League of Legends or of a *Hero* in Overwatch are what make those games so interesting to play.

OPTIONAL PROCEDURE DETAILS

6. Displaying the status of the race by printing the `toString()` method of each racer is informative, but not terribly interesting to watch. Create a `GraphicalDisplay` class with a method that can be called after each update to display the entire length of the racetrack and icons for each racer placed at their relative positions along the racetrack. This display might be in text form in a Terminal window, or it might be more sophisticated graphics in a Processing display.
7. Include a wagering component by giving the person running your program an initial balance of “Tortoise Tokens” or something similar. Before each race the user can choose who to wager on, and how many tokens to wager. If their racer wins, the amount of their wager is added to their balance—otherwise it is subtracted. They can keep wagering until they run out of tokens.

TRACKING REVISIONS WITH `git` AND GitHub

This project is a great opportunity to get more experience using **git** and **GitHub** to manage your project with a partner. Because there are a number of different aspects to the project, it's reasonable that you might start creating branches as work proceeds. By the time your work is in full development you might have the following **git** branches:

- `main` - This “production” branch which has work from the other branches that has been thoroughly tested and then merged with this branch
- `runner` - a branch for writing the `RacerRunner` class which has the `main` method that will initialize all instances and run the simulation.
- `race` - This branch includes development of a class that will keep track of the racers, the total distance of the track, and (during the event), the relative positions of the racers
- `tortoise / hare / other` - You might have branches for developing each of these subclasses, or you might just develop them in a generic `development` branch used for all your ongoing work.
- `textDisplay` - Used for developing a class that will allow for text-based representation of the characters as they move across a Terminal window
- `graphicDisplay` - Used for creating a series of file that will work in a Processing environment
- `wagers` - a branch that modifies the `RacerRunner` class to include gambling
- `documentation` - This branch includes the writing of JavaDocs as the program proceeds

This might seem like a lot of branches! Wouldn't it be easier just to have two branches—`main` and `development`—and leave it at that? While having more branches does require a bit more switching back and forth, it's *much* easier to coordinate work with your partner if you're each working on different branches: you're less likely to find that their work is conflicting with yours.

Consider starting work with your partner on a `main` branch, and develop together the `AbstractRacer` class. Push that `main` to the GitHub repository, and from there you can both pull down the most recent version of `main`, create a branch for whatever it is that you want to work on, and go from there.

REFERENCE

```
/**
 * The abstract class AbstractRacer is used to describe a participant
 * in a race. The abstract method move() is defined differently
 * according to each different subclass that inherits from Racer.
 */

public abstract class AbstractRacer
{
    // instance variables
    private String name;
    private int position;

    /**
     * Constructs an object according to the subclass
     * that inherits from Racer. (You can't construct an
     * actual Racer object because it's abstract. Go ahead,
     * try it!)
     */
    public Racer(String name)
    {
        this.name = name;
        position = 0;
    }

    /**
     * The abstract method move() must be defined by
     * subclasses.
     */
    public abstract void move();

    /**
     * Overrides the toString method
     */
    public String toString()
    {
        // to be completed
    }

    /**
     * The getPosition() method returns the current position
     * of the racer.
     */
    // to be completed

    /**
     * The setPosition() method takes a parameter specifying the
     * next position and mutates position to that specified location.
     */
    // to be completed
}
```

SAMPLE OUTPUT

The standings are:

```
Racer[name=Hank the Tortoise,position=0]
Racer[name=Willy the Hare,position=0], Energy: 15
Racer[name=Freddy the Frog,position=0]
```

The standings are:

```
Racer[name=Hank the Tortoise,position=1]
Racer[name=Willy the Hare,position=3], Energy: 14
Racer[name=Freddy the Frog,position=0]
```

The standings are:

```
Racer[name=Hank the Tortoise,position=2]
Racer[name=Willy the Hare,position=6], Energy: 13
Racer[name=Freddy the Frog,position=-1]
```

The standings are:

```
Racer[name=Hank the Tortoise,position=3]
Racer[name=Willy the Hare,position=9], Energy: 12
Racer[name=Freddy the Frog,position=3]
```

.
.
.

The standings are:

```
Racer[name=Hank the Tortoise,position=56]
Racer[name=Willy the Hare,position=45], Energy: 0
Racer[name=Freddy the Frog,position=58]
```

The standings are:

```
Racer[name=Hank the Tortoise,position=57]
Racer[name=Willy the Hare,position=45], Energy: 0
Racer[name=Freddy the Frog,position=58]
```

The standings are:

```
Racer[name=Hank the Tortoise,position=58]
Racer[name=Willy the Hare,position=45], Energy: 15
Racer[name=Freddy the Frog,position=58]
```

The standings are:

```
Racer[name=Hank the Tortoise,position=59]
Racer[name=Willy the Hare,position=48], Energy: 14
Racer[name=Freddy the Frog,position=57]
```

.
.
.

The standings are:

```
Racer[name=Hank the Tortoise,position=98]
Racer[name=Willy the Hare,position=45], Energy: 0
Racer[name=Freddy the Frog,position=71]
```

The standings are:

```
Racer[name=Hank the Tortoise,position=99]
Racer[name=Willy the Hare,position=45], Energy: 0
Racer[name=Freddy the Frog,position=72]
```

The winner is: Racer[name=Hank the Tortoise,position=100]