Past Year PE1 Question: Gym

Adapted from PE1 of 21/22 Semester 2

Instructions to Past-Year PE1 Question:

1. Accept the repo on GitHub Classroom here
2. Log into the PE nodes and run `~cs2030s/get py1` to get the skeleton for all available past year PE1 questions.
3. The skeleton for this question can be found under `2122-s2-q1`. You should see the following files:
   - The files `Test1.java`, `Test2.java`, and `CS2030STest.java` for testing your solution.
   - One skeleton file is provided for each class/interface required.

Background

You are building a system to help with the management and administration the newly opened SoC gym. This system will need to keep track of the equipment and people in the gym.

Create Equipment, Treadmill, and Dumbbell Classes

We first need to create classes to keep track of the equipment of the Gym. Currently the Gym has two types of equipment: Treadmills and Dumbbells. Create three classes for `Equipment`, `Treadmill`, and `Dumbbell`. Keep in mind that we may want to add new classes later on when the gym gets more different types of equipment.

All `Equipment` may be in use or not in use. The "in use" status of this equipment can be set using the `setInUse` method with takes in a single boolean argument. Implement an `isInUse` method in the class `Equipment` which returns a `boolean`. `Equipment` also needs to be repaired from time to time, and this is achieved by using the `repair` method which takes in no arguments. In order to repair the equipment we need to know what type of equipment it is. Repairs happen instantly and have no affect on use.

Equipment.java

```java
abstract class Equipment {
    private boolean inUse = false;

    public boolean isInUse() {
        return this.inUse;
    }

    public void setInUse(boolean inUse) {
        this.inUse = inUse;
    }

    abstract void repair();
}
```

A `Dumbbell` has a weight associated with it, represented as a `double` in kilograms. This weight cannot be changed after the `Dumbbell` is created. The `Dumbbell` method has a `getWeight` method which will return the current weight. We also want to keep track of the number of times the `Dumbbell` is repaired as they keep breaking. A method `getRepairCount` will return the number of repairs done on the Dumbbell.

Dumbbell.java

```java
class Dumbbell extends Equipment {
    private final double weight;
    private int repairCount = 0;

    public Dumbbell(double weight) {
        this.weight = weight;
    }

    public double getWeight() {
        return this.weight;
    }

    @Override
    public void repair() {
        this.repairCount++;
    }

    @Override
    public String toString() {
        return "Dumbbell: " + weight + " kg";
    }

    public int getRepairCount() {
        return this.repairCount;
    }
}
```

A `Treadmill` will move at a certain speed (a `double` representing the speed in kilometers per hour), this can be changed by using `setSpeed` method. Implement a `setSpeed` method which takes in a single `double`. Implement a `getSpeed` method that returns the current speed. When a `Treadmill` is repaired, the speed of the device is reset back to zero. We do not need to keep track of the number of `Treadmill` repairs.

Treadmill.java

```java
class Treadmill extends Equipment {
    private double speed = 0.0;

    public void setSpeed(double speed) {
        this.speed = speed;
    }

    public double getSpeed() {
        return this.speed;
    }

    @Override
    public void repair() {
        this.speed = 0.0;
    }

    @Override
    public String toString() {
        return "Treadmill: " + this.speed + " km/h";
    }
}
```

Study the sample calls below to understand what is expected for the constructors, `toString` and other methods of these classes. Implement your classes so that they output they behave the same way.

```
jshell> Equipment e = new Equipment();
|  Error:
|  Equipment is abstract; cannot be instantiated
|  Equipment e = new Equipment();
|                ^-------------^
jshell> Equipment e = new Treadmill();
e ==> Treadmill: 0.0 km/h
jshell> Equipment e = new Dumbbell(2.5);
e ==> Dumbbell: 2.5 kg
jshell> e.isInUse();
$.. ==> false
jshell> e.setInUse(true);
jshell> e.isInUse();
$.. ==> true
jshell> e.setInUse(false);
jshell> e.isInUse();
$.. ==> false
```

```
jshell> e.repair();
jshell> Dumbbell d = new Dumbbell(2.5);
d ==> Dumbbell: 2.5 kg
jshell> d.getWeight();
$.. ==> 2.5
jshell> d.getRepairCount();
$.. ==> 0
jshell> d.repair();
jshell> d.getRepairCount();
$.. ==> 1
jshell> Treadmill t = new Treadmill();
t ==> Treadmill: 0.0 km/h
jshell> t.setSpeed(3.0);
jshell> t
t ==> Treadmill: 3.0 km/h
jshell> t.getSpeed();
$.. ==> 3.0
jshell> t.repair();
jshell> t.getSpeed();
$.. ==> 0.0
jshell> e.getWeight();
|  Error:
|  cannot find symbol
|    symbol:   method getWeight()
|  e.getWeight();
|  ^---------^
jshell> e.setSpeed(3.0);
|  Error:
|  cannot find symbol
|    symbol:   method setSpeed(double)
|  e.setSpeed(3.0);
|  ^--------^
jshell> e.getSpeed();
|  Error:
|  cannot find symbol
|    symbol:   method getSpeed()
|  e.getSpeed();
|  ^--------^
```

You can test your code by running the `Test1.java` provided. Make sure your code follows the CS2030S Java style.

```
$ javac -Xlint:rawtypes -Xlint:unchecked Test1.java
$ java Test1
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml *.java
```

Create a `Trainer`, `Customer`, `CannotTrainException`, and `Gym` class

There are two types of people in the gym, Customers and Trainers. A `Trainer` can only train one `Customer` at a time, but a `Customer` can be trained by many `Trainers`. We may in the future need to create different people that work in the gym such as admin staff.

You may add new classes/interfaces as needed by the design.

Each person has a name. A `Trainer` can train a `Customer` using an `Equipment` if the `Trainer` is not currently training anyone and the `Equipment` to be used is not in use. The `startTraining` method takes in two arguments, a `Customer` and the `Equipment` to be used. If a `Customer` can be trained, the `Equipment` becomes in use. If the `Customer` can not be trained, the `startTraining` method should throw an `CannotTrainException`. The `CannotTrainException` is a checked exception. Note that Java's `Exception` constructor takes in a single `String` which contains the Exception message:

```
public Exception(String message)
```

The `stopTraining` method will free up the `Trainer` and stop the `Equipment` from being in use. A `Trainer` also has a `getStatus` method which takes in no arguments and will return a `String` describing if a `Trainer` is training someone or not.

CannotTrainException.java

```java
public class CannotTrainException extends Exception {
    public CannotTrainException() {
        super("Cannot Train!");
    }
}
```

Person.java

```java
abstract class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return this.name;
    }
}
```

Trainer.java

```java
class Customer extends Person {
    public Customer(String name) {
        super(name);
    }

    @Override
    public String toString() {
        return "Customer: "+ super.toString();
    }
}
```

Trainer.java

```java
class Trainer extends Person {
    private Customer trainee;
    private Equipment equipment;

    public Trainer(String name) {
        super(name);
    }

    public void startTraining(Customer c, Equipment e) throws
CannotTrainException {
        if (!e.isInUse() && this.trainee == null)  {
            this.trainee = c;
            this.equipment = e;
            e.setInUse(true);
        } else  {
            throw new CannotTrainException();
        }
    }

    public void stopTraining() {
        if (this.equipment != null)  {
            this.equipment.setInUse(false);
            this.trainee = null;
            this.equipment = null;
        }
    }

    public String getStatus() {
        if (this.trainee == null) {
            return this.toString() + " not training";
        } else {
            return this.toString() + " training "  + trainee;
        }
    }

    @Override
    public String toString() {
        return "Trainer: " + super.toString();
    }
```

We need a class to represent the `Gym`. For social distancing reasons, this class needs it needs to keep track of the people (the trainers and customers) entering the gym. The constructor of the class `Gym` takes in a single `int` which is the capacity of the gym. The `enter` method takes in either a `Trainer` or `Customer` and prints out whether or not the person can enter the gym using `System.out.println`. Note, you do not need to keep track of which people are already in the gym, merely the number of people in the gym.

Gym.java

```java
class Gym {
    private int peopleCount = 0;
    private final int capacity;

    public Gym(int capacity) {
        this.capacity = capacity;
    }

    public void enter(Person p) {
        if (peopleCount < capacity ) {
            System.out.println(p + " can enter");
            this.peopleCount++;
        } else {
            System.out.println(p + " cannot enter");
        }
    }

    @Override
    public String toString() {
        return "Gym Capacity: " + peopleCount + "/" + this.capacity;
    }
}
```

Study the sample calls below to understand what is expected for the constructors, `toString` and other methods of these classes. Implement your classes so that they behave the same way.

```
jshell> Treadmill treadmill1 = new Treadmill();
treadmill1 ==> Treadmill: 0.0 km/h
jshell> Treadmill treadmill2 = new Treadmill();
treadmill2 ==> Treadmill: 0.0 km/h
jshell> Customer c1 = new Customer("Bob");
c1 ==> Customer: Bob
jshell> Customer c2 = new Customer("Sally");
c2 ==> Customer: Sally
jshell> Trainer t1 = new Trainer("Frank");
t1 ==> Trainer: Frank
jshell> t1.getStatus()
$.. ==> "Trainer: Frank not training"
jshell> Trainer t2 = new Trainer("Sam");
t2 ==> Trainer: Sam
jshell> Exception e = new CannotTrainException();
```

```
e ==> CannotTrainException: Cannot Train!
jshell> t1.startTraining(c1, treadmill1);
jshell> t1.getStatus();
$.. ==> "Trainer: Frank training Customer: Bob"
jshell> t1.startTraining(c2, treadmill1);
|  Exception REPL.$JShell$16$CannotTrainException: Cannot Train!
|        at Trainer.startTraining (#7:16)
|        at (#19:1)
jshell> t1.getStatus();
$.. ==> "Trainer: Frank training Customer: Bob"
jshell> t1.stopTraining();
jshell> t1.startTraining(c2, treadmill1);
jshell> t1.getStatus();
$.. ==> "Trainer: Frank training Customer: Sally"
jshell> t1.startTraining(c1, treadmill2);
|  Exception REPL.$JShell$16$CannotTrainException: Cannot Train!
|        at Trainer.startTraining (#7:16)
|        at (#24:1)
jshell> t1.getStatus();
$.. ==> "Trainer: Frank training Customer: Sally"
jshell> t2.startTraining(c2, treadmill2);
jshell> t2.getStatus();
$.. ==> "Trainer: Sam training Customer: Sally"
jshell> t1.stopTraining();
jshell> t1.getStatus()
$.. ==> "Trainer: Frank not training"
jshell> t2.stopTraining();
jshell> t2.getStatus()
$.. ==> "Trainer: Sam not training"
jshell> Gym gym = new Gym(2);
gym ==> Gym Capacity: 0/2
jshell> gym.enter(c1);
Customer: Bob can enter
jshell> gym;
gym ==> Gym Capacity: 1/2
jshell> gym.enter(t1);
Trainer: Frank can enter
jshell> gym;
gym ==> Gym Capacity: 2/2
jshell> gym.enter(c2);
Customer: Sally cannot enter
jshell> gym;
gym ==> Gym Capacity: 2/2
```

You can test your code by running the `Test2.java` provided. Make sure your code follows the CS2030S Java style.

```
$ javac -Xlint:rawtypes -Xlint:unchecked Test2.java
$ java Test2
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml *.java
```