INSTRUCTIONS

1. This Practical Assessment consists of two questions. Answer ALL questions.
2. The total mark for this assessment is 30. Answer ALL questions.
3. This is an OPEN BOOK assessment. You are only allowed to refer to written/printed notes. No online resources/digital documents are allowed, except those accessible from the PE nodes (peXXX.comp.nus.edu.sg) (e.g., man pages are allowed).
4. You should see the following in your home directory.
5. The files Test1.java, Test2.java, Test3.java, Test4.java, and CS2030STest.java for testing your solution.
6. The skeleton files for Question 1: `Equipment.java`, `Dumbbell.java`, `Treadmill.java`, `Customer.java`, `Trainer.java`, `CannotTrainException.java`, and `Gym.java`
7. The skeleton files for Question 2: `ArrayStack.java`, and `Stack.java`
8. You may add new classes/interfaces as needed by the design.
9. Solve the programming tasks by creating any necessary files and editing them. You can leave the files in your home directory and log off after the assessment is over. There is no separate step to submit your code.
10. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
11. Write your student number on top of EVERY FILE you created or edited as part of the @author tag. Do not write your name.
12. Important: Make sure all the code you have written compiles. If one of the Java files you have written causes any compilation error, you will receive 0 marks for that question.

QUESTION 1: Gym System

Marking criteria:

- correctness (3 marks)
- design (10 marks)

- style (2 marks)

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.

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.

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.

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.

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.

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
```

QUESTION 2: ArrayStack

Marking criteria:

- correctness (10 marks)
- design (3 marks)
- style (2 marks)

Recall the Stack, a First-In-Last-Out (FILO) data structure. You can pop an item off the top of the stack, and push an item on to the stack. In this question, we will implement a generic stack using an array.

In this question you are not permitted to use `java.util.Stack` or `java.util.ArrayList`.

Create a new generic interface `Stack` and an `ArrayStack`

We first need to create a `Stack<T>` interface. It is a generic interface, with three abstract methods: - A `pop` method which returns an object of type `T` and has no arguments - A `push` method which returns nothing and has a single argument of type `T` - A `getStackSize` method returns an `int` and has no arguments

Next, create a class `ArrayStack<T>` which implements `Stack<T>` using an array. The order of the items in the array dictates the order of items in the stack. This class has a constructor which takes in a single `int` which represents the maximum depth of the stack. The `push` method should put an item on top of the stack. If there is no more space in the stack, the `push` method should disregard the item being pushed on to the stack. The `pop` method should remove an item from the top of the stack and return it. If there are no items on the stack, the `pop` method should return `null`. The `getStackSize` method should return how many items are in the stack. Finally, the `toString` method should show the contents of the stack.

If you find yourself in a situation where the compilers generate an unchecked type warning, but you are sure that your code is type safe, you can use `@SuppressWarnings("unchecked")` (responsibly) to suppress the warning.

Study the sample calls below to understand what is expected for the constructor, `toString` and other methods of `ArrayStack`. Implement your class so that it outputs in the same way.

```
jshell> Stack<Integer> st = new ArrayStack<>(3);
st ==> Stack:
jshell> st.push(1);
jshell> st;
st ==> Stack: 1
jshell> st.push(1);
jshell> st;
st ==> Stack: 1 1
jshell> st.push(2);
jshell> st;
st ==> Stack: 1 1 2
jshell> st.getStackSize();
$.. ==> 3
jshell> st.push(3);
jshell> st;
st ==> Stack: 1 1 2
jshell> st.pop();
$.. ==> 2
jshell> st;
st ==> Stack: 1 1
jshell> st.getStackSize();
$.. ==> 2
jshell> st.pop();
$.. ==> 1
jshell> st
st ==> Stack: 1
jshell> st.getStackSize();
$.. ==> 1
jshell> st.pop();
$.. ==> 1
jshell> st
st ==> Stack:
jshell> st.pop();
$.. ==> null
jshell> st
st ==> Stack:
jshell> st.pop();
$.. ==> null
jshell> st
st ==> Stack:
jshell> st.push(2);
jshell> st;
st ==> Stack: 2
jshell> Stack<String> st2 = new ArrayStack<>(10);
st2 ==> Stack:
jshell> st2.push("Hello");
jshell> st2;
st2 ==> Stack: Hello
jshell> st2.push("World");
jshell> st2;
st2 ==> Stack: Hello World
jshell> st2.pop();
$.. ==> "World"
jshell> st2.pop();
$.. ==> "Hello"
```

You can test your code by running the `Test3.java` provided. Make sure your code follows the CS2030S Java style.

```
$ javac -Xlint:rawtypes -Xlint:unchecked Test3.java
$ java Test3
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
*.java
```

Creating a factory method `of` and a `pushAll` method

We will now implement a factory method `of`, this method will take in an array of items and an `int` which represents the maximum depth of the stack, and return a `ArrayStack` with the items pushed onto the stack in the order that they are present in the array. If the array length is greater than the size of the stack, only include the first `n` items of the array, where `n` is the stack size. For compatibility with `Test3.java`, you should not make your original constructor private.

We will also create a `pushAll` method that has a single argument which is an `ArrayStack`. `pushAll` repeatedly pops one item from the given `ArrayStack` and pushes them onto the target `ArrayStack`, until the given `ArrayStack` is empty. Note that if the target `ArrayStack` is full, the pushed items will be lost.

In addition, we will create a `popAll` method that has a single argument which is an `ArrayStack`. `popAll` repeatedly pops one item from the target `ArrayStack` and pushes them onto the given `ArrayStack`, until the target `ArrayStack` is empty. Note that if the given `ArrayStack` is full, the pushed items will be lost.

Study the sample calls below to understand what is expected for the new methods of `ArrayStack`. Implement your class so that it outputs in the same way.

```
jshell> ArrayStack.of(new Integer[] {1, 2, 3}, 10);
$.. ==> Stack: 1 2 3
jshell> ArrayStack.of(new Object[] {1, "foo", "bar"}, 10);
$.. ==> Stack: 1 foo bar
jshell> ArrayStack<Integer> as0 = ArrayStack.of(new Integer[] {1, 2, 3,
4}, 2);
as0$ ==> Stack: 1 2
jshell> ArrayStack<Integer> as1 = ArrayStack.of(new Integer[] {4, 5, 6},
10);
as1 ==> Stack: 4 5 6
jshell> ArrayStack<Integer> as2 = ArrayStack.of(new Integer[] {1, 2, 3},
10);
as2 ==> Stack: 1 2 3
jshell> as2.pushAll(as1);
jshell> as2;
as2 ==> Stack: 1 2 3 6 5 4
```

```
jshell> as1;
as1 ==> Stack:
jshell> as1 = ArrayStack.of(new Integer[] {4, 5, 6}, 10);
as1 ==> Stack: 4 5 6
jshell> ArrayStack<Integer> as3 = ArrayStack.of(new Integer[] {1, 2, 3},
5);
as3 ==> Stack: 1 2 3
jshell> as3.pushAll(as1);
jshell> as3;
as3 ==> Stack: 1 2 3 6 5
jshell> ArrayStack<Number> asn = new ArrayStack<>(10);
asn ==> Stack:
jshell> asn.pushAll(as2);
jshell> asn
asn ==> Stack: 4 5 6 3 2 1
jshell> ArrayStack<String> as4 = ArrayStack.of(new String[] {"d", "e",
"f"}, 10);
as4 ==> Stack: d e f
jshell> ArrayStack<String> as5 = ArrayStack.of(new String[] {"a", "b",
"c"}, 10);
as5 ==> Stack: a b c
jshell> as4.popAll(as5);
jshell> as5;
as5 ==> Stack: a b c f e d
jshell> as4 = ArrayStack.of(new String[] {"d", "e", "f"}, 10);
as4 ==> Stack: d e f
jshell> ArrayStack<String> as6 = ArrayStack.of(new String[] {"a", "b",
"c"}, 5);
as6 ==> Stack: a b c
jshell> as4.popAll(as6);
jshell> as6;
as6 ==> Stack: a b c f e
jshell> ArrayStack<Integer> as7 = ArrayStack.of(new Integer[] {7, 8, 9},
5);
as7 ==> Stack: 7 8 9
jshell> as7.popAll(asn);
jshell> asn;
asn ==> Stack: 4 5 6 3 2 1 9 8 7
```

You can test your code by running the `Test4.java` provided. Make sure your code follows the CS2030S Java style.

```
$ javac -Xlint:rawtypes -Xlint:unchecked Test4.java
$ java Test4
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
*.java
```