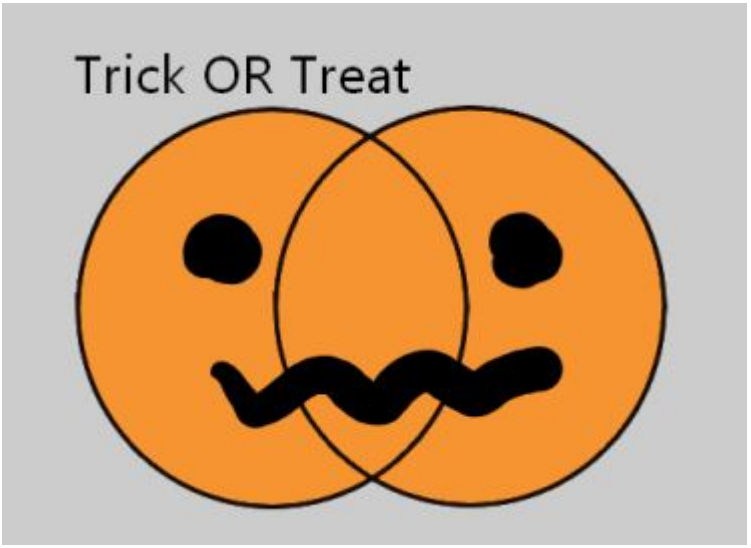
INSTRUCTIONS

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 and should not be submitted to CodeCrunch.
6. The skeleton files for Question 1: `Candy.java`, `MysteryCandy.java`, `CandyBag.java`, `Trickers.java`, `Treaters.java`, and `NoCandyException.java`.
7. The skeleton files for Question 2: `Pair.java`, `Queueable.java`, `EndQueue.java`, and `Sequeue.java`.
8. The file `PE1Q1.java` and `PE1Q2.java` for semi-automated grading on CodeCrunch.
9. You may add new classes/interfaces as needed by the design.
10. 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.
11. Only the files directly under your home directory will be graded. Do not put your code under a subdirectory.
12. Write your student number on top of EVERY FILE you created or edited as part of the @author tag. Do not write your name.
13. 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: Trick or Treat



> This is a BOO-lean algebra

Marking criteria:

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

You are helping your neighbourhood to make trick or treating safer this year by keeping track of the candies and the tricks and ensuring that nobody carries more than they can.

Create Candy, MysteryCandy, and Chocolate Classes

We first need to create classes to keep track of the candies. Currently there are two types of equipment: Mystery Candy and CandyBag. Create three classes for `Candy`, `MysteryCandy`, and `CandyBag`. Keep in mind that we may want to add new candies later on.

All `Candy` has a weight in integer (*i.e.*, `int`) but the actual weight depends on the type of candy. You can always ask for the weight of any candy via the `getWeight` method. All the weights are given in grams. Additionally, all candy can be eaten via the `eat` method. We can check if a candy has already been eaten or not using the method `isEaten`.

There are common behaviour of eating a candy, but in order to fully eat the candy, we need to know what type of candy it is. For instance, we need to opened the wrapper for

some candy before eating it. Eating is instantaneous and after eating a candy, its weight is immediately reduced.

A `MysteryCandy` has a flavour associated with it in addition to the weight. The flavour is represented as a `String`. To eat a candy, we have to first open the wrapper using `open` method. If the candy is already opened/eaten, then nothing happen. Similarly, if we try to eat an unopened candy or an already eaten candy, nothing happen. There is no need to throw any exception for this. Once a candy is eaten, the weight immediately is reduced to zero.

A `CandyBag` has many candies where each candy in the bag has the same weight. It is created using two parameters an `int` number of candies and an `int` weight of each candy. There is no need to open the candy bag to eat the candy. But each time we eat, the number of candies in the bag is decremented by one. This will also reduce the weight of the candy bag. Once there are no more candies, we can say that the candy is already been eaten and `isEaten` should return `true`. When there are no more candies, eating the candy produce no effect.

Study the sample calls below to understand what is expected for the constructors, `toString` and other methods of these classes. In particular, for `CandyBag`, we print the number of candy left over the total candy at the start. Implement your classes so that they output they behave the same way.

```
jshell> Candy c = new Candy()
|  Error:
|  Candy is abstract; cannot be instantiated
|  Candy c = new Candy();
|            ^---------^
jshell> new MysteryCandy("Sweet", 2)
$.. ==> Sweet Candy (2 gram)
jshell> Candy c = new CandyBag(5, 4) // 5 candies, 4 grams each
$.. ==> CandyBag 5/5 (20 gram)

jshell> c.getWeight()
$.. ==> 20
jshell> c.isEaten()
$.. ==> false
c.eat()
jshell> c
$.. ==> CandyBag 4/5 (16 gram)
c.eat()
c.eat()
c.eat()
jshell> c
$.. ==> CandyBag 1/5 (4 gram)
jshell> c.isEaten()
$.. ==> false
c.eat()
jshell> c.isEaten()
```

```
$.. ==> true
jshell> c
$.. ==> CandyBag 0/5 (0 gram)
c.eat()
jshell> c.isEaten()
$.. ==> true
jshell> c
$.. ==> CandyBag 0/5 (0 gram)

jshell> Candy c = new MysteryCandy("Sour", 3)
$.. ==> Sour Candy (3 gram)
jshell> c.isEaten()
$.. ==> false
c.eat() // not yet opened
jshell> c
$.. ==> Sour Candy (3 gram)
jshell> c.open()
|  Error:
|  cannot find symbol
|    symbol:   method open()
|  c.open()
|  ^----^
jshell> MysteryCandy mc = (MysteryCandy) c
$.. ==> Sour Candy (3 gram)
mc.open()
jshell> c
$.. ==> Sour Candy (3 gram)
c.eat() // already opened
jshell> c
$.. ==> Sour Candy (0 gram)
jshell> c.isEaten()
$.. ==> true
```

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 Treaters, Trickers, and NoCandyException Classes

There are two types of people in the neighbourhood doing trick-or-treating. The first is the treater who are giving candies and the second is the tricker who are receiving candies. We also let tricker to give candy to another tricker. Write two classes `Treaters` and `Trickers`. We may in the future need to create different people in the neighbourhood such as an adult who accompanies trickers.

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

Each person has a name. A `Treaters` has one type of candy but he/she has bought many. In this case, a single `CandyBag` is considered one candy. We can construct a `Treaters` using three arguments, the name, the candy, and the number of candies.

A `Treaters` will give a candy to a `Trickers` if the `Treaters` has some candy left. The `give` method takes one argument which is the `Trickers` that will receive the candy. However, if the `Treaters` has no candy left, the method will throw the checked exception called `NoCandyException`. Note that Java's `Exception` constructor takes in a single `String` which contains the Exception message:

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

We need to keep track of how many candies have been given and how many are left. This information can be obtained from the method `getStatus` describing how many candies have been given away out of the total number of candies the `Treaters` has.

There is a possible complication when a `Treaters` try to give a candy to a `Trickers`. What happen if the `Trickers` does not have enough remaining weight in the bag? We can *attempt* to have a `Trickers` to receive a candy via `receive` that takes in a single parameter `Candy`. This `receive` method in the class `Trickers` returns

- `true` if the `Trickers` can receive the candy and modify the last received candy and the remaining weight of the bag (*see description below*).
- `false` if the `Trickers` cannot receive the candy and do not modify the `Trickers`.

A `Trickers` has a single trick described by a `String` as well as a single bag with a limited capacity. The capacity is given in `int` and it captures the maximum weight the `Trickers` can carry. We do not need to remember all candies that the `Trickers` has received. Instead, we simply has to keep track of only the latest candy that the `Trickers` has received.

If the `Trickers` received another candy, then we simply forget any previous candies. In any case, as soon as a `Trickers` receive a candy, we record the remaining weight the bag can carry. This latest candy can also be given to another `Trickers`, thus increasing the remaining weight the bag can carry. After the `Trickers` give away the candy to another `Trickers`, we can simply let the value of the latest candy to be `null`. Now, if there is no latest candy (*i.e.,* `null`), then we throw `NoCandyException`.

A `Trickers` may also play trick using the method `playTrick` that takes in a single argument. This argument may be a `Trickers` or a `Treaters`, let's call this a trick-or-treaters. When the `Trickers` play a trick to a trick-or-treaters, the trick-or-treaters attempt to give a candy to the `Trickers`. If the trick-or-treaters has no candy to give

regardless of whether the `Trickers` *can actually receive it or not*, then the `Trickers` will actually perform the trick to the trick-or-treaters by returning a `String` describing the trick. Otherwise, we simply return a `String` describing the candy the `Trickers` receive (*again, regardless of whether or not the* `Trickers` *can actually receive because the remaining weight may not be enough to receive*).

For each time a `Trickers` actually perform the trick to the trick-or-treaters, we record the number of performance. This information can be obtained from the method `getStatus` describing how many tricks are played and the remaining weight in the bag.

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> Candy c = new MysteryCandy("Choco", 3)
$.. ==> Choco Candy (3 gram)
jshell> Treaters adi = new Treaters("Adi", c, 5)
$.. ==> Treater Adi
jshell> Trickers ida = new Trickers("Ida", "Spin", 5)
$.. ==> Trickers Ida
jshell> Trickers ada = new Trickers("Ada", "Jump", 5)
$.. ==> Trickers Ada
jshell> Trickers idi = new Trickers("Idi", "Boo!", 20)
$.. ==> Trickers Idi

jshell> ida.receive(c)
$.. ==> true
jshell> ida.getStatus()
$.. ==> Tricker Ida performed 0 tricks and has bag size 2 remaining
jshell> ida.receive(c)
$.. ==> false
jshell> ida.getStatus()
$.. ==> Tricker Ida performed 0 tricks and has bag size 2 remaining

jshell> ada.playTricks(ida) // Ida has candy
$.. ==> Tricker Ada receives from Tricker Ida
jshell> ida.getStatus()
$.. ==> Tricker Ida performed 0 tricks and has bag size 5 remaining
jshell> ada.getStatus()
$.. ==> Tricker Ada performed 0 tricks and has bag size 3 remaining

jshell> ada.playTricks(ida) // Ida has no candy
$.. ==> Tricker Ada perform the trick Jump to Tricker Ida
jshell> ida.getStatus()
$.. ==> Tricker Ida performed 0 tricks and has bag size 5 remaining
jshell> ada.getStatus()
$.. ==> Tricker Ada performed 1 tricks and has bag size 3 remaining

jshell> ida.give(ada)
|  Exception REPL.$JShell$14$NoCandyException: No More Candy!
|        at Trickers.give (#6:42)
|        at (#15:1)
```

```
39
40  jshell> idi.playTricks(adi)
41  $.. ==> Tricker Idi receives from Treater Adi
42  jshell> idi.playTricks(adi)
43  $.. ==> Tricker Idi receives from Treater Adi
44  jshell> idi.playTricks(adi)
45  $.. ==> Tricker Idi receives from Treater Adi
46  jshell> idi.playTricks(adi)
47  $.. ==> Tricker Idi receives from Treater Adi
48  jshell> idi.playTricks(adi)
49  $.. ==> Tricker Idi receives from Treater Adi
50
51  jshell> idi.getStatus()
52  $.. ==> Tricker Idi performed 1 tricks and has bag size 5 remaining
53  jshell> adi.getStatus()
54  $.. ==> Treater Adi gave 5 away out of 5 Choco Candy (3 gram)
```

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

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

QUESTION 2: Queue Using Pair

Marking criteria:

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

Recall the Stack, a First-In-Last-Out (FILO) data structure. On the other hand, a Queue is a First-In-First-Out (FIFO) data structure. You can enqueue using the method `enq` to the back of the queue and dequeue using the method `deq` from the front of the queue. In this question, we will implement a generic queue using the Pair generic class.

In this question you are not permitted to use `java.util.Stack`, `java.util.ArrayList`, `java.util.LinkedList`, *etc*. Additionally, if implemented correctly, your queue should not even contain any fields! Furthermore, you should make your methods as flexible as possible following PECS. This is especially important in the second part of the question.

While useful, there is no need to make the class immutable as long as you can pass the test cases. The class `Pair<T,S>` is given for your reference in the file `Pair.java` and reproduced below for convenience. You should not modify this file.

```
public class Pair<T,S> {
  private T fst;
  private S snd;

  public Pair(T fst, S snd) {
    this.fst = fst;
    this.snd = snd;
  }

  public T getFst() {
    return this.fst;
  }
  public S getSnd() {
    return this.snd;
  }

  @Override
  public String toString() {
    return this.fst + "; " + this.snd;
  }
}
```

Create a new generic interface `Queueable` as well as `EndQueue` and `Sequeue` classes

We first create a `Queueable<T>` interface. It is a generic interface, with four abstract methods:

1. An `enq` method to enqueue an element into the queue.
    - The method takes in an element of type `T` and returns a new `Queueable<T>` with the element added to the back of the queue.
2. A `deq` method to dequeue an element from the queue.
    - The method takes in no argument returns a new `Queueable<T>` with the front of the queue removed from the queue.
3. A `peek` method to get the first element of the queue.
    - The method takes in no argument and returns an element of type `T` corresponding to the front of the queue.
4. A `size` method to get the size of the queue.
    - The method takes in no argument and returns an `int` corresponding to the size of

the queue.

Next, we create two classes:

1. `EndQueue<T>` that extends from `Pair` and implements `Queueable<T>`.
    - This corresponds to the empty queue, which marks the end of the queue.
2. `Sequeue<T>` that extends from `Pair` and implements `Queueable<T>`.
    - This corresponds to a non-empty queue.

For simplicity, you may assume that we will never dequeue or peek from an empty queue in our test. However, you should still implement these two methods but the implementation is left to your discretion. Also, we will only use `Queueable<T>` as our compile-time type to prevent the use of `getFst` and `getSnd` from the `Pair` class.

The basic idea here is that we can represent a queue as a pair containing the front element and the rest of the queue.

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 `Queueable`. Implement your class so that it outputs in the same way.

```
jshell> Queueable<Integer> qInt = new EndQueue<>()
$.. ==> {END}
jshell> qInt = qInt.enq(1)
$.. ==> 1; {END}
jshell> qInt = qInt.enq(2)
$.. ==> 1; 2; {END}
jshell> qInt = qInt.enq(1)
$.. ==> 1; 2; 1; {END}
jshell> qInt.size()
$.. ==> 3
jshell> qInt.peek()
$.. ==> 1
jshell> qInt = qInt.deq()
$.. ==> 2; 1; {END}
jshell> qInt.size()
$.. ==> 2
jshell> qInt.peek()
$.. ==> 2
jshell> Queueable<String> qStr = new Sequeue<>("A", new EndQueue<>())
$.. ==> A; {END}
jshell> qStr = qStr.enq("B")
$.. ==> A; B; {END}
jshell> qStr = qStr.enq("A")
```

```
$.. ==> A; B; A; {END}
jshell> qStr.size()
$.. ==> 3
jshell> qStr.peek()
$.. ==> A
jshell> qStr = qStr.deq()
$.. ==> B; A; {END}
jshell> qStr.size()
$.. ==> 2
jshell> qStr.peek()
$.. ==> B
```

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
```

Zipping and Merging Queue

We will now implement two static methods to zip and merge the queue. In both cases, you may assume that the two queues will have the same size.

Zipping a queue is done by the *static* method `zip`

- The method takes in two parameters of type `Queueable` containing type `T` *but* should be as flexible as possible.
- The method returns a new `Queueable` containing type `Pair` *but* should be as flexible as possible.
- The method works by taking the front of the two queues from the parameters and forming a pair from these two elements. The pair is then enqueued into the result. This operation is repeated until the two queues are empty (*they are always of the same size*).

Merging a queue is done by the *static* method `merge`

- The method takes in two parameters of type `Queueable` containing type `T` *but* should be as flexible as possible.
- The method returns a new `Queueable` containing type `T` *but* should be as flexible as possible.
- The method works by:
    - Taking the front of the queue from the first parameter and enqueueing this to the

resulting queue.

- Followed by taking the front of the queue from the second parameter and enqueueing this to the resulting queue.
- Remember, the two parameters will be queues of the same size.

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

```
jshell> Queueable<Integer> qInt = new Sequeue<>(1, new Sequeue<>(2, new
Sequeue<>(3, new EndQueue<>())))
$.. ==> 1; 2; 3; {END}
jshell> Queueable<String> qStr = new Sequeue<>("A", new Sequeue<>("B",
new Sequeue<>("C", new EndQueue<>())))
$.. ==> A; B; C; {END}
jshell> Queueable<?> zQue = Queues.zip(qInt, qStr)
$.. ==> 1; A; 2; B; 3; C; {END}
jshell> zQue.peek()
$.. ==> 1; A
jshell> zQue.deq().peek()
$.. ==> 2; B
jshell> Queueable<?> mQue = Queues.merge(qInt, qStr)
$.. ==> 1; A; 2; B; 3; C; {END}
jshell> mQue.peek()
$.. ==> 1
jshell> mQue.deq().peek()
$.. ==> A

jshell> class A {
   ...>   @Override
   ...>   public String toString() {
   ...>     return "{A}";
   ...>   }
   ...> }
jshell> class B extends A {
   ...>   @Override
   ...>   public String toString() {
   ...>     return "{B}";
   ...>   }
   ...> }
jshell> class C extends B {
   ...>   @Override
   ...>   public String toString() {
   ...>     return "{C}";
   ...>   }
   ...> }
jshell> Queueable<B> qB1 = new Sequeue<>(new C(), new Sequeue<>(new C(),
new Sequeue<>(new C(), new EndQueue<>())))
$.. ==> {C}; {C}; {C}; {END}
jshell> Queueable<B> qB2 = new Sequeue<>(new C(), new Sequeue<>(new C(),
new Sequeue<>(new C(), new EndQueue<>())))
$.. ==> {C}; {C}; {C}; {END}
jshell> Queues.<A>zip(qB1, qB2)
$.. ==> {C}; {C}; {C}; {C}; {C}; {C}; {END}
```

```
jshell> Queues.<A>zip(qB1, qB2).peek()
$.. ==> {C}; {C}
jshell> Queues.<A>merge(qB1, qB2)
$.. ==> {C}; {C}; {C}; {C}; {C}; {C}; {END}
jshell> Queues.<A>merge(qB1, qB2).peek()
$.. ==> {C}
```

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
```