Quiz Preparation for Week 8

Concepts

- Abstract
- Interface
- Anonymous Class
- Lambda Expression

EMP Session Links

1. EMP 10-15: Topics including reference/abstract/interface
2. EMP 10-20: Topics including interface/lambdas.

Quick Concept Overview & Practice

Abstract

1. Can be applied to both classes and methods, e.g.:

```
public abstract class Shape {
  // remember, abstract method doesn't have a body
  public abstract int sides();
}
```

2. An *abstract class* may or may not contain *abstract methods*, but if you want to have *abstract method(s)*, the class needs to be abstract. The following is wrong:

```
public class Shape {
  public abstract int sides();
}
```

3. An *abstract class* cannot be instantiated (you can't create an instance for an abstract class).
4. An *abstract class* can be extended (you can have another class extends an abstract class and calls its methods). Children classes need to implement the abstract methods, else they also need to be abstract.

Practice 1

Design an abstract class Type with two methods: one is an *instance method* named receipt that takes in an amount of money (double), and returns a String as "Total is xxx dollars"; the other is an *abstract instance method* named getType() which you are too lazy to implement yet (you are smart and know that you shouldn't implement it yet).

▶ Spoiler!

```
public abstract class Type {
  public String receipt(double money) {
    return "Total is " + Double.toString(money) + " dollars";
  }
  public abstract String getType();
}
```

Now you want to create some children classes which extend Type and implement the getType() method! Create two different classes extending Type, one named Chicken and the other named Beetroot. For Chicken, getType() should return a string "Meat"; for Beetroot, getType() should return a string "Vegatable".

▶ Spoiler!

```
public class Chicken extends Type {
  @Override
  public String getType() {
    return "Meat";
  }
}
public class Beetroot extends Type {
  @Override
  public String getType() {
    return "Vegetable";
  }
}
```

Interface

1. A java class can only extends from one parent, but it can implement multiple interfaces. It means it supports the functionalities declared in the interfaces. E.g.:

```
interface A {
  int funcA();
}
interface B {
  int funcB();
}
public class Parent { }
public class Alphabet extends Parent implements A, B {
  public int funcA() {
    return 1;
  }
  public int funcB() {
    return 2;
  }
}
```

2. Interface works similarly with polymorphism/inheritance. You can have a variable of an interface type and it can refer to the classes that implements the interface. E.g.:

```
// given the codes in (1)
A a = new Alphabet();
// this code returns 1
int some = a.funcA();
// this code gives error because interface A does not have funcB() supported
int other = a.funcB();
```

3. Comparable is a built-in java interface that allows you to implement the *compareTo()* method for instance comparison.
4. Iterator, Iterable are two other built-in java interfaces that allow you to implement methods for enhanced for-loop (iterating items).

Practice 2

Design a class named Sequence implementing Comparable. It contains an int array of length 5 named seq. The constructor should take in an int array, assert it to be not null and of length 5, and assign seq to its value.

What's more, you need to override the compareTo(Object o) method, which should make sure o is Sequence type and do the following comparison:

- if all elements in seq are larger than or equal to (except all are the same) the corresponding ones in seq of o, then return 1;
- if ... are the same as ..., then return 0;
- if any is smaller, then return -1;

▶ Spoiler!

```
public class Sequence implements Comparable {
  private int[] seq = new int[5];
  public Sequence(int[] toSet) {
    assert (toSet != null && toSet.length == 5);
    seq = toSet;
  }

  @Override
  public int compareTo(Object o) {
    assert o instanceof Sequence;
    Sequence s = (Sequence) o;
    int equalCount = 0;
    for (int i = 0; i < 5; i++) {
      if (seq[i] < s.seq[i]) {
        return -1;
      } else if (seq[i] == s.seq[i]) {
        equalCount++;
      }
    }
    if (equalCount == 5) {
      return 0;
    } else {
      return 1;
    }
  }
}
```

Anonymous Class

1. An *anonymous class* does not have a name, must extend a class or implement an interface, is created immediately, and cannot provide a constructor. Syntax for example:

```
interface Test {
  void test();
}
Test t = new Test() {
  @Override
  public void test() { }
};  // don't forget the semi-colon!
t.test();
```

2. *Anonymous classes* can be used to flexibly create inline instances that perform certain tasks the caller defines.

Please review the lecture on 10/15/2020 - "Uses for Anonymous Classes".

Practice 3

```
interface Relation {
  int relation(int a, int b);
}
```

You are provided with an interface named Relation. Design three anonymous classes to implement the following functionalities for the method relation(int a, int b):

- One should get the remainder of a dividing b.
- One should get the larger value of a and b.
- One should get the closer value to 100 of a and b.

▶ Spoiler!

```
Relation remainder = new Relation() {
  @Override
  public int relation(int a, int b) {
    return b % a;
  }
};
Relation larger = new Relation() {
  @Override
  public int relation(int a, int b) {
    if (a > b) {
      return a;
    } else {
      return b;
    }
  }
};
Relation closerTo100 = new Relation() {
  @Override
  public int relation(int a, int b) {
    if (Math.abs(a - 100) < Math.abs(b - 100)) {
      return a;
    } else {
      return b;
    }
  }
};
```

Lambda Expression

1. Functional programming is a style of programming for which programs are constructed by applying and composing functions, while object-oriented programming (e.g. Java) is a style for which programs are constructed by manipulating objects.
2. Java cannot store functions in variables (as opposing to what functional programming). However, Java can achieve some similar expression style called lambda expression. For example:

```
interface Increment {
  int increment(int value);
}
// method 1 - using anonymous class
```

```
Increment a = new Increment() {
  @Override
  public int increment(int value) {
    return value + 1;
  }
};
// method 2 - using lambda expression
Increment b = (value) -> value + 1;

// the following codes both print 11
System.out.println(a.increment(10));
System.out.println(b.increment(10));
```

Practice 4

See if you can convert the anonymous classes in practice 3 to lambda expressions.

▶ Spoiler!

```
interface Relation {
  int relation(int a, int b);
}

// Remainder using lambda expression
Relation remainder = (a, b) -> b % a;

// Larger than using lambda expression
Relation larger = (a, b) -> {
  if (a > b) {
    return a;
  } else {
    return b;
  }
};

// Closer to 100 using lambda expression
Relation closerTo100 = (a, b) -> {
  if (Math.abs(a - 100) < Math.abs(b - 100)) {
    return a;
  } else {
    return b;
  }
};
```