Object Oriented Programming – I

Review-Part 2

MODULE – V: Java Programming:

1) A simple Java statement ends with a ;
2) A compound Java statement is bracketed by { } and contains 0 or more statements (simple or compound).
3) All Java programming consists of writing compound/reference types. **The basic reference type is the class.** A class definition consists of a header followed by a compound statement.

```
public class MyFirstClass extends Object {

}
```

- The part starts with "public" is called the "class header".
- The part between the { } is called the "class body". The class body is by definition a compound statement, and all elements of the class (fields, method, other compound types) go in the body (between the { }).

The parts are:

- **public:** the access modifier
- **class:** this is a class
- **MyFirstClass:** the name of the class (a name can be almost anything, but professional Java style is to always start with a capital letter)
- **extends Object:** indicates the super class of this class. Every class has exactly 1 super class (also called the "**parent**" class).

NOTE: The access modifier determines where the code can be directly accessed. (This is not a security feature. All code can still be indirectly accessed.)

The access modifier can be any of the four:

- **public:** the code can be used/accessed anywhere in the program
- **protected:** the code can be used in this type and in any type that extends this type (or in any type in the same folder as this type)
- **private:** the code can only be used in the body of this type

If you omit the access modifier, the default is "**package**": the code can be used in any type that is in the same package (i.e. in the same folder) as the containing type.

What can go in a class body?

- field declarations (these can include assignment operators)
- methods
- other non-primitive type definitions (these are called inner types or nested types)
- **constructors**: special methods used to initialize instances of the class

4) Each public class must go in its own file. The file name must be the same as the class name, and the file extension must be .java. The class must be compiled before you use it. The compiler creates a file with the same name but .class extension that contains the Java bytecode for the class.

5) What's the point of a super class? (More about this later)
- Every class "inherits" all public and protected instance methods of the super classç
- Every class has access to all public and protected constructors, fields, class (i.e. static) methods, and nested types of the super class.

<u>Fields:</u>

There are two kinds of fields.

- Instance fields are memory that is allocated inside each instance. Thus, every instance has its own copy of an instance field.
- Class fields are memory that is allocated outside of the instances. There is only one copy of a class field that all the instances share.

To create an instance field is the same as declaring a variable except you add an access modifier:

access-modifier type name

To create a class field, you do the same but add the word "static" before the variable type:

```
public class MyFirstClass extends Object {
      private int myInstanceField;
      private static int myClassField;
   }
```

Every instance gets its own copy of the instance fields. (Note you will have to change the access modifiers to public if you want to test this in the interactions pane.)

```
MyFirstClass c1 = new MyFirstClass();
c1.myInstanceField = 5;
MyFirstClass c2 = new MyFirstClass();
c2.myInstanceField = 6;
c1.myInstanceField   ← returns 5
c2.myInstanceField   ← returns 6
  -- these are two different variables with the same name stored inside two different objects
c1.myClassField = 10;
c2.myClassField = 20;
c1.myClassField    ← returns 20 because there is only 1 copy of this variable.
MyFirstClass.myClassField  ← also returns 20.  Since the field belongs to the class, you can
use the classname to access it
```

<u>Methods:</u>

An instance method definition is a method header followed by a compound statement.

access-modifier return-type name(input parameters) {

```
}
```

Here is an example that takes two inputs of type int and returns a value of type int

```
public int myMethod(int input1, int input2) {
```

A class method uses the same declaration but adds "static" before the return type:

```
public static int myClassMethod(int input1, int input2) {
```

(An instance method "operates" on an instance while a class method does not. More on this very soon!)

- the input parameters are a sequence of 0 or more variable declarations separated by commas. There is one variable declaration for each input your method will take.
- the return type can be any type, or if the method will not return a value, it is "void".
- the part starting with the access modifier is called the "method header"
- the part between the { } is called the "method body".
- all code describing the behavior of the method goes in the body. The body is by definition a compound statement.

```
public class MyFirstClass extends Object {
    public int average(int input1, int input2) {
        int sum = input1 + input 2;
        int avg = sum / 2;
        return sum;
    }
}
```

A return statement must be included in any method that has a non-void return type. The return statement gives the output of the method.

Now we can use this method:

```
> MyFirstClass c = new MyFirstClass();

> c.average(300,500)  ← 400
```

When you call the method, Java assigns the first value to the first variable of the input parameters and the second value to the second variable. **All type rules apply!** So we have to use two values that are type int (or whose type Java will automatically convert to int).

We also have to give two values as input.

```
> c.average(300)  ← Error no method with input (int), only (int, int)
```

What is the point of the super class? Your class inherits all the public and protected instance methods of the super class:

```
public class MyFirstFrame extends JFrame {

}
```

→ Now MyFirstFrame inherits all the methods of JFrame, and so we can use them:

```
MyFirstFrame f = new MyFirstFrame();
f.setVisible(true);
f.setSize(300,500);
```

<u>MODULE – V:</u> "this"

An Interesting Java Class, inherited methods, the keyword "**this**"

Let's build GeometricFrame, a class that adds new features to JFrame. First, we create the class structure:

```
    public class GeometricFrame extends JFrame {

    }
```

Now, all the new features of that we want to add to GeometricFrame go in the class body: between the { and }

The super/parent class:

Every class (except one) in Java has exactly one super, or parent, class. The super class is set by the "**extends** ..." portion of the class header. If you do not write the "**extends** ...", then your class extends the Object class by default.

(Object is the only class in Java that does not have a super clsas.)

A class inherits all the public and protected instance methods of its super class.

A class has access to all the public and protected class methods, instance and class fields, nested classes of the super class.

Essentially a class has access to everything public and protected of the super class, but the instance methods are special.

Instance methods are inherited from the super class.

You will see the difference between "access" and "inherited" in a lecture coming soon. In our example, GeometricFrame's super class is JFrame. That means GeometricFrame inherits all the public and protected instance methods of JFrame

(plus access to the other public and protected things).

So the GeometricFrame can do everything that a JFrame can do. It can become visible, change its size, etc.

How do we get the GeometricFrame to do something JFrame cannot?

We put a method inside the GeometricFrame class body. Everything we place inside the class will be indented. This is the professional Java style (and general coding style) used to make clear what is inside the class and what is not.

We added three methods:

1) transpose: flips the height and with width of the window.

What access modifier? → public: we want to be able to use this method anywhere.

What return type? → void: there is nothing we need to return.

(Hint: a method should just do what its name says it will do. Any additional actions will only confuse the programmer.)

What name? → transpose: a great choice!

What input values? → No input. We want to just type j.transpose() to flip the window, similar to j.getHeight() to get the height.

One way to think of it: the window knows its own height and width. We should only provide as input data that the window does not already know.

public void transpose() {

→ Now, how do we get the height and width of the window?

We use the **getHeight()** and **getWidth()** methods. But whose height and width?

(write it in the interactions pane)

Interactions Pane:

```
>  GeometricFrame g = new GometricFrame()

> int height = g.getHeight()

> int width = g.getWidth()

> g.setSize(height, width)
```

When we type **g.transpose()** we want g's width and height, but g is not declared inside the method body or the class.

The Java keyword we need is "**this**". Java provides it to us.

- **this** is a special variable that exists inside instance (non-static) methods.
- **this** stores the address of the instance/object that the method is acting on.
- **this** acts a hidden parameter to the method.

We do not see it as input to the method, but it is.

When we call **g.transpose()**, Java will take the value stored in **g** (an address for an object) and copy it into the special variable called **this**.

```
public void transpose() {
        int height = this.getHeight();
        int width = this.getWidth();
        this.setSize(height, width);
}
```

2) scale: scales a window by a scaling factor.

What access modifier? ← public: we want to be able to use this method anywhere.

What return type? ← void: there is nothing we need to return

What name? ← scale

What input values? ← A single double to indicate how much to scale the window dimensions by.

This code is very similar to transpose, but notice that we must use a typecast. (It would be even nicer if we rounded,)

```
public void scale(double factor) {
        this.setSize((int)(this.getWidth() * factor), (int)(this.getHeight() * factor));
}
```

3) isEqualArea: returns true if this window is equal in area to another window

What access modifier? ← public so we can use the code anywhere

What return type? ← boolean, to say true or false

What input values? ← We need to know what window we are comparing **this** window to. So we need a single input value

What should I use for the input type? ← I decided on GeometricFrame but JFrame also works since GeometricFrame is also a JFrame

```
public boolean isEqualArea(GeometricFrame input) {
```

        Now, to compare the area, we need the width and the height

```
public boolean isEqualArea(JFrame other) {

        int myArea = this.getWidth() * this.getHeight();

        int inputArea = input.getWidth() * input.getHeight();
```

Now, what do we return? The result of comparing the two areas.

(Remember, that we can use any expression anywhere in the code as long as the types match. Here, we need to return a boolean.

```
        return myArea == otherArea;
```

So, here is the full code:

```
public boolean isEqualArea(JFrame input) {

        int myArea = this.getWidth() * this.getHeight();

        int inputArea = input.getWidth() * input.getHeight();

        return myArea == inputArea;

 }
```

Again, we can write this without using the two variables "myArea" and "inputArea". How do we call the method?

```
GeometricFrame frame1 = new GeometricFrame(100, 100);
GeometricFrame frame2 = new GeometricFrame(1000, 10);
frame1.isEqualArea(frame2) ← this should result false
frame2.isEqualArea(frame1) ← this should result false
```

MODULE – VI: Java Non-Primitive Type Rules:

Let's create a Phone class to represent telephones:

```
public class Phone extends Object  {
        /* a method to make a call to a particular number */
        public void makeCall(PhoneNumber number) {
        /* some code goes here */
        }
}
```

What does the "extends Object" mean?

- It says that any instance that is type Phone also is type Object.
- The Phone class "**inherits**" all the public and protected instance methods of Object.
- The Phone class "**has access to**" all the public and protected fields (both class and instance), class (i.e. static) methods, constructors, and nested classes of Object.

Here is a map of the class heirarchy with a few extra classes thrown in

```
             Object
                |
          Phone (includes the method makeCall)
             /    \
    LandLine    CellPhone (includes the method sendText)
                   /    \
          FlipPhone    SmartPhone (includes the methods downloadApp and useApp)
                          /    \
                   Android    iPhone (overrides downloadApp, includes faceTime method)
(overrides the method downloadApp)
                      /              \
                 Galaxy6          iPhone12
                    /                  \
              Galaxy6Edge         iPhone12mini
```

Java Non-Primitive Type Rules:

1. When you create an instance of a class:

```
new iPhone()
```

the instance is type iPhone, but it is also the type of every class it extends (all the way up to Object), so it is also type SmartPhone, CellPhone, Phone, and Object. It is all these types at the same time. **This is called "polymorphism" for "many types".**

2. The "**true type**" is the type it is created as: new iPhone() means the instance will have true type iPhone.

(The "true type" is called the "run-time type" in Java references or sometimes just "class".)

Every object knows its true type (it is stored in the object's data), and the true type does not change.

3. The "**current type**" is which of its polymorphic types it is typecast to.

Every place that value is used in the code will have a current type associated with it. (The "current type" is called the "compile-time type" in Java references or sometimes just "type".)

4. You can typecast a non-primitive value to any of its valid polymorphic types.

(An object created with "new iPhone()" can be typecast as any of Object, Phone, CellPhone, SmartPhone, or iPhone, and no other class.)

5. A typecast that goes "up" the hierarchy (wider) is automatic, by a typecast that goes "down" the hierarchy (narrower) must be explicit.

6. The current type determines what methods you are allowed to call as well as what fields and inner classes you will access.

For example,

```
CellPhone c = new iPhone();
```

The object stored in c has true type iPhone, but the current type of c is CellPhone.

The typecast from iPhone to CellPhone is automatic and legal because CellPhone is wider than (up the hierarchy from) iPhone.

You can only call methods valid for CellPhone on c:

c.sendText(...)   is legal

c.downloadApp(...) is not legal. Even though the object stored in c is currently an iPhone, you can't write this code because it will not work for *all* cell phones. Later in the program, c might store a FlipPhone.

7. The true type determines the version of an instance method that is called.

For example:

```
SmartPhone p = new iPhone();
```

p.downloadApp(...) ← this will get the app from the AppStore

← why? Because the true type of the object stored in p is iPhone and the iPhone class overrode the downloadApp method to use the AppStore

p = new AndroidPhone();

p.downloadApp(...) ← Same code, but now it will download from Google Play

← why? Even though the current type of p is SmartPhone, the object stored in p has true type AndroidPhone, and the AndroidPhone class overrode the downloadApp method to use GooglePlay.

Why have this hierarchy?

It enables us to write a method once and have it works for lots of different kinds of types:

```
public void securePhone(SmartPhone p) {
        p.downLoadApp(SheildApp);
}
```

Now we can write a program: For every phone in our database, call:

securePhone(phone)

and it will load the ShieldApp onto the phone. We don't need to write special code to test whether the phone we are loading the app onto is Android, iOS, BlackBerry, or whatever. We just call the downloadApp method of SmartPhone and count on the instance to know it's true type and thus to know which overridden version of the method to use.

A summary of the rules:

1) Every object is created as a specific type using the new operator. It is called “true type".

The true type determines how the object will behave. It does this by determining what version of a method is run. The true type never changes.

Ex: new MyFirstWindow() creates an instance whose true type is MyFirstWindow.

new JFrame() creates an instance whose true type is JFrame.

2) The object is not only its "**true type**". It is also the type of the super class of the true type, that class's super class, and so on up to Object. This property of being many types at the same time is called "polymorphism".

3) A typecast does not change the object. The typecast does not change the true type of the object. Instead, the typecast determines which of the legal polymorphic types for the object is the object acting as at this line of code.

I call the type that the object is acting as the "**current type**". The compiler uses the current type to determine what methods you are allowed to call and what fields you can access. **You can only call methods and access fields that exist for that current type.** The specific field or class method accessed will depend on the current type, but the instance method that is used will be the version of the true type.

Example:

```
public class MyFirstClass extends Object {
        public int add(int x, int y) {
                return x + y;
          }
}
public class MySecondClass extends MyFirstClass {
        public int mult(int x, int y) {
                return x * y;
        }
}
```

MySecondClass c2 = new MySecondClass();
c2.mult(5, 6) ← legal, MySecondClass has a mult method
c2.add(5, 6) ← legal, MySecondClass has an add method (inherited from MyFirstClass
MyFirstClass c1 = c2 ← legal, this is widening so the typecast is automatic
c1.add(3, 4) ← legal, MyFirstClass has an add method
c1.mult(3, 4) ← illegal, MyFirstClass does not have a mult method (it does not matter that the object currently stored in c1 has true type MySecondClass
c2 = c1 ← illegal, this is narrowing so an explicit typecast is needed
c2 = (MySecondClass)c1 ← legal because the true type of the right hand side is MySecondClass
c2 = (MySecondClass)new MyFirstClass() ← illegal, the true type of the object on the right hand side is MyFirstClass