Object Oriented Programming – I

Review-Part 3

MODULE – VI: Java Non-Primitive Type Rules:

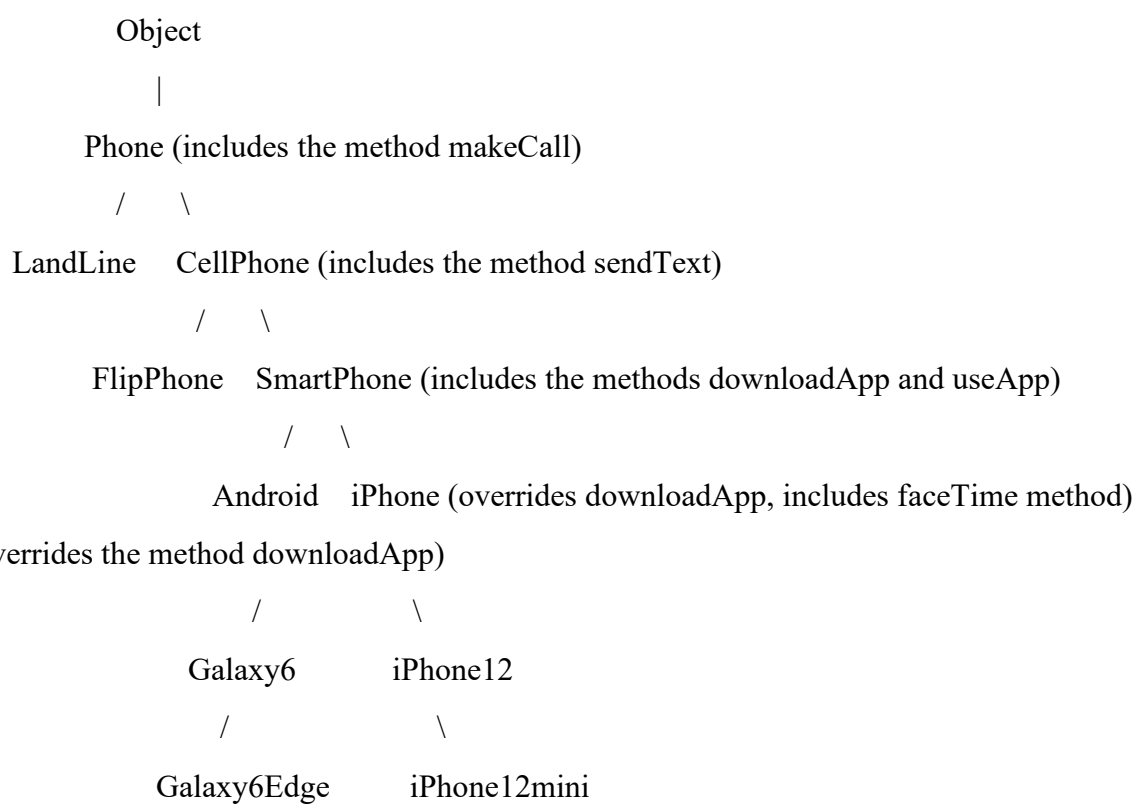
Let's create a Phone class to represent telephones:

```
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



<u>Java Non-Primitive Type Rules:</u> 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

MODULE – VI: Good Object-Oriented Coding, Part 1:

To take advantage of the nice features of the Java language, we should follow some rules when writing our types. Here are the first rules:

1. Any data that must be stored in a type should be stored in a private field.
2. Any access of the data (by code outside the class) should be implemented using a public (or protected) method.
3. Anytime we want to access the field, we should use the public method instead of the field.

Building a Class from Scratch

If we want to model a game die, we need to create a class, and we need the properties of:

1) rolling a die

2) getting a die face / value

3) setting the die face

For each behavior/property, we need to create an appropriate method. What information will we need to remember for the die:

1) the current face value (we have to remember the result of a roll)

2) the size of the die (maybe we want a 4-sided die, a 6-sided die, etc.)

What should the class extend?

If there is nothing in the API that matches the Die type, we should just extend Object. **Note:** do NOT extend a class unless it makes sense to say Die "is a" ... for the class you are extending.

Ex: If Die extends JFrame, then we are saying that all game dice are JFrames. That would indeed be bizarre.

```
public class Die extends Object{ Java will include “extends Object” if we do not write
        private int currentValue;
        public int getValue() {
                return this.currentValue;
        }
```

(**Please note that the "this." is not needed. If omitted, Java automatically adds it, but it is included here for completeness**.)

Also, we should add comments above each method and field so that someone reading the code understands the purpose:

```
/* returns the current face value */
public int getValue() {
        return this.currentValue;
}
```

```
/* sets the value of the die */
public void setValue(int newFace) {
        this.currentValue = newFace;
}
```

(Also note, that setValue allows any value to be set.

A better solution would be to use an if statement so that if the value is between 1 and the maximum allowed value.

For the roll, we use the random() method of the Math class: the random() method takes no input and returns a random double value in the range [0.0, 1.0). A little math and knowing our types lets us convert the value in [0.0, 1.0) to a value in {1, 2, 3, 4, 5, 6}.

```
public int roll() {
        this.currentValue = (int)(Math.random() * 6.0 + 1);
        return this.currentValue;
}
```

Initial field/variable values

All local variables: do not get default values. When you create a local variable, the first use of the variable must be to assign a value to it.

Fields, on the other hand, are given a default value of 0, false, or null, depending on its type. If we do not give currentValue an initial value, it will get value 0. 0 is not a good value for a die. So, we should give the field an initial value.

```
        private int currentValue = 1;
```

Magic Numbers

The use of 6 is not good. It is a "magic number". A "magic number" is a number that has a special meaning to the programmer. Here, you need to know that 6 is the size of the die. Magic numbers should be avoided and replaced with variables.

(Any number that is not 0 or 1 is usually magic.)

```
public static int numberOfSides = 6;

public int roll() {
        this.currentValue = (int)(Math.random() * Die.numberOfSides + 1);
        return this.currentValue;
}
```

That is better because we gave it a name. We can make this better. The size of a die does not change, so let us indicate that in the field.

Final: Marking something as final means its value will not be changed.

- A final variable will not change its value after the first assignment.
- A final method will not be overridden.

- A final class will not be extended.

If we want all die to have 6 sides, we can also make the field "static" to indicate that it belongs to a class: public static final int numberOfSides = 6;

```
public int roll() {
        this.currentValue = (int)(Math.random() * Die.numberOfSides + 1);
        return this.currentValue;
}
```

Notice that we made numberOfSides public instead of private. That is okay because we are dealing with a static (i.e. class) field.

Unlike with instance methods and fields, static methods behave exactly the same as static fields in that a class has access to the static methods/fields of its super class, but it does not inherit them.

Finally, the method roll() violated one of our rules for O-O coding:

It violates the last rule: we are accessing the currentValue field directly instead of the getter/setter methods we wrote. Let's fix that:

```
public int roll() {
        this.setValue((int)(Math.random() * Die.numberOfSides + 1));
        return this.getValue();
}
```

Now, it satisfies all of our rules.

Notice that we do not have any methods for numberOfSides because we (currently) do not intend for code outside this class to access that data.

If we did, then we should create a getter method and use that here.

MODULE – VII: Constructors

Recall how the new operator works: Die d = new Die();

1) Allocates space for the object.

2) Initializes the object based on the inputs to new → it does this by calling an appropriate constructor method with the given input.

3) Returns the address (memory location) for the object.

- A constructor is a special method that is called by the new operator to initialize a class.
- A constructor must have the same name as the class and no return type:
- So, a constructor has the form: public ClassName(inputs) {
- A constructor is not inherited by classes that extend this class.
- Each class must define its own constructors.

Default Constructors:

If you do not write a constructor, Java provides a default constructor that takes no input.

Writing a Constructor: Here is a good constructor to specify the size of the die:

```
private int currentValue = 1;

private final int numberOfSides;

public Die(int numberOfSides) { //this is the constructor that takes a single value
        this.numberOfSides = numberOfSides;
}
```

Note that the parameter variable and the instance field have the same name "numberOfSides". Java allows local variables inside methods to have the same name as fields.

Inside the method, numberOfSides refers to the closest definition (the parameter). If we wanted the field, we needed to use this.numberOfSides. Java does not allow two fields with the same name nor two local variables (including input parameter variables) with the same name.

Now we can use the constructor to create different die of different sizes:

```
Die d8 = new Die(8);

Die d20 = new Die(20);
```

What happened to the default constructor?

However, this code will now give an error: Die d6 = new Die(); It worked before, what happened?

IMPORTANT: If you do not define a constructor, Java provides a default one that takes no input. Once you create a constructor for your class, you lose the default constructor. It would be nice to still have a default constructor, so we must write our own.

Method overloading:

We can create multiple methods of the same name (including constructor methods) as long as their "**parameter signature**" differs.

The "parameter signature" is the type, number, and order of the input values.

Since we have a constructor that takes an int: Die(int), we can write another construct as long as the input is not a single int.

```
public Die() {
        this.numberOfSides = 6;
}
```

Two important methods of Object and Overriding methods

An example class: Employee. The Employee will have names, salaries, and numbers. We use the proper Java coding:

```
/* This class represents employees */
public Employee { // the extends Object is automatically added
        // a fild to store the employee name
          private String name;

          // a field to store the employee salary
          private double salary;

          // a field to store the employee number
          private int number;

          // a field to store the last employee number used
          private static int lastEmployeeNumber = 0;

          public String getName() {
              return name;
          }

          public void setName(String name) {
              this.name = name;
          }

          -etc-
```

Note that the body of getName returns name. We also could have written "return this.name;", and it would be exactly the same. Java automatically prefixes "this." in front of a method call or field access if you do not provide it.

Note that the body of setName has another variable (the input parameter) that is called name. This is ok because we can distinguish the input parameter variable "name" from the field "this.name". Here we have to explicity use the "this." because Java does not automatically add it.

➔ **Why?** Because there is already a variable called name, and since that declaration is closer,

Java will assume "name" just refers to the input parameter.

➔ **Why did we do the assignment?**

Input parameters as well as any variable declared inside a method only exists as long as the method is run. So once setName is done, the name variable along with its contents are lost. To keep the contents around, we copy it into the field that exists as long as the object does.

The Constructor:

In class, we decided that we want to require an employee to have a name a salary and a number. However, we decided that employee numbers may be assigned automatically. We do this by

writing a constructor method that takes a String as input. In writing a constructor, we lose Java's default constructor that takes no input.

```
public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
        this.number = ?????
}
```

➔ How do we get the number to be assigned automatically?

We need the class itself to save the next employee number to be used. That means we need a static field!

```
        private static int last = 0;

public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
        this.number = Employee.lastEmployeeNumber + 1;
        Employee.lastEmployeeNumber = this.number;
}
```

Overriding methods of Object

Employee class has a method which is called getSalary(). This method uses the salary assigned to the employee. However, we do not have a single type of employment. For instance, we can have an HourlyEmployee who gets paid by the hours worked. We can still take advantage of Employee class.

```
public class HourlyEmployee extends Employee{
        private int hoursWorked = 10; // this is set to default for teaching purpose
        private double hourlyWage = 35.5;  // we do not assign values here, we use constructor
        public HourlyEmployee(String name){
                super(name,0); // first thing in constructor is to call another constructor
        }
}
```

Now HourlyEmployee class uses the getSalary() method inherited from Employee. However, if we create an hourly employee instance, then the salary will be 0 since the employee has not worked. Therefore,

HourlyEmployee h = new HourlyEmployee("Orhan");

h.getSalary() → this will return 0. However, hourly employee worked 10 hours and makes 35.5TL per hour. Therefore, the salary should be 10x35.5=355. This is because we are still using the getSalary() method from parent class. In order to change the behabiour of the inherited methods, we **override** them. IDE can help us determine any mistake on the override using an annotation. This particular annotation is **@Override**, and if we place it before a method that is

overriding an inherited method, the compiler, upon seeing the annotation, will verify that we really are overriding. If we are not, it will give a compiler error instead of allowing the overload.

```
public class HourlyEmployee extends Employee{
	private int hoursWorked = 10; // this is set to default for teaching purpose
	private double hourlyWage = 35.5;  // we do not assign values here, we use constructor
	public HourlyEmployee(String name){
		super(name,0); // first thing in constructor is to call another constructor
	}

	@Override
	public double getSalary(){
		return hoursWorked* hourlyWage;
	}
}
```

Now that we override the getSalary method, the version which is called will be determined by the true type.
HourlyEmployee h = new HourlyEmployee("Orhan");

h.getSalary() → will return 10x35.5=355. This is because h has true type of HourlyEmployee and it will use the getSalary() method from the HourlyEmployee class.

MODULE – VIII: More on Constructors:

How Constructors Work:

1) The first line of a constructor must be a call to another constructor. (This is also the only place in the code were we can have a constructor call.)

These are the two possibilities

super() ← possibly with input in the parentheses. This calls the constructor of the parent class.

this() ← possibly with input in the parentheses. This calls a constructor of the same class.

If you do not explicitly have a constructor call as the first line of your constructor body, Java automatically places super(), with no input, there.

WHY? Recall polymorphism. An object of type Employee is also type Object. An object of type GeometricFrame is also JFrame, Frame, Window, Container, Component, and Object.

The purpose of the constructor is to initialize the object. Before we can initialize the objects as Employee, it must first initialize itself as Object.

2) The constructors do the following when they are run:

a) The first line of the constructor that calls another constructor is executed. (Recall that Java adds super() if you omit this line.)
b) All fields of the instance are initialized.
c) The rest of the constructor body is executed.

Note point (b) above. Java basically takes any assignment statements on your fields and places that code after the constructor call that is the first line of your constructor and before the rest of the constructor code. This is important to remember for the situations where you care about the order that things are being done in your program.

Let's create a class that extends Employee:

```
public class HourlyEmployee extends Employee {
}
```

This class will not compile as written! The reason is that we did not write a constructor so Java provided a default constructor. The default constructor takes not input and calls super() with no input. Something like this:

```
public HourlyEmployee() {
        super();
}
```

Now we see why HourlyEmployee fails to compile. The first line of the HourlyEmployee constructor is super();

super() calls the constructor of Employee that takes no input. But there is no constructor of Employee that takes no input.

The Employee constructor requires a String and a double as input. So, we have to have a String and double as a parameters to super();

One thing we could do is to create a constructor that calls

```
        super("Some Dumb String", 0);
```

And the code will compile because the types not match. However, this is does not fit what we want Employee to be. The purpose of the Employee constructor was to require a name for each Employee. We should write our code for HourlyEmployee to follow this idea, require a name as input, and then pass this name along to the Employee constructor.

Here is the correct code.

```
public class HourlyEmployee extends Employee {
        public HourlyEmployee(String name) {
                super(name, 0);
        }
}
```

HourlyEmployee inherits everything from Employee:
> Employee e = new Employee("Orhan", 1000)
> e.geteNumber() → 1
> e.getName() → " Orhan "
> HourlyEmployee h = new HourlyEmployee("Joe")
> h.getName() →"Joe"
> h.getNumber() → 2

Using this()

this() calls a constructor of the same class. We can use it to avoid having duplicate code.Consider the two constructors of Employee:

```
public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
        this.number = Employee.lastEmployeeNumber + 1;
        Employee.lastEmployeeNumber = this.number;
}
public Employee(int number, String name, double salary) {
        this.name = name;
        this.salary = salary;
        this.number = number;
        if (this.number > Employee.lastEmployeeNumber)
                Employee.lastEmployeeNumber = this.number;
}
```

Other than how they handle the employee number, the two are the same. So we can have the first (the one that sets the employee number automatically) call the other (the one that sets it explicitly).

```
        public Employee(String name, double salary) {
          this(Employee.lastEmployeeNumber + 1, name, salary);
        }
```

Now, if we need to change how employees are initialized, there is only one constructor body with the code that we have to change.

MODULE – IX:

The String class

String is a pre-defined class in Java and one of the most important classes.

Strings represent text as a sequence of characters.

Strings have a special means for creating instances.

While we can create instances using the new operator (just as with any class), we can also just place the desired string inside double quotes.

"Hello" ← this is a shortcut to creating a new String using the appropriate sequence of characters as input.

Note that "Hello" acts just like the new operator, it evaluates to the address of the String object storing h,e,l,l,o.

We can use the object just like any other object. We can store it

```
String s = "Hello";
```

Strings have a special operator, the "+" operator.

The result of + is a new String that concatenates the two operands together.

If one operand is not a String, an appropriate String is created (through a sequence of object creation and method calls) before the concatenation.

Ex: "Hello" + "there" → "Hellothere"

"Hello " + "there" → "Hello there"

"x = " + 3 → "x = 3"

Note: you must be careful when mixing Strings and numeric primitives with the +. When is the + meant to be a String concatenation and when is it meant to be a normal addition?

"x = " + 3 + 5 will return "x = 35" (+ is evaluated left to right) but

3 + 5 + " = y" will return "8 = y"

Two important methods of Object and Overriding methods

Employee extends Object and inherits the methods of Object. There are two important methods we should override because their default behavior is not very useful.

1) String **toString()**

This method produces a String representation of the object. It is used anytime we need a String representation of the object such as when writing to the terminal or when used with the String concatenation operator + to concatenate objects to a string.

The method in Object sets the string to be the true type name of the object and it's "hash code" (basically its location in memory).

Usually, we want the string to be something more useful, so we override the method.

In our case, we want it to be the employee number followed by the employee name

```
/*change the behavior of the inherited toString */
public String toString() {
        return getNumber() + ": " + getName();
}
```

2) boolean **equals(Object o)**

This method is used to compare two objects.

➔ If you use the == operator on non-primitive values, it is comparing the addresses of the values. Thus, == only returns true if two objects are the exact same object.

For example, "Hi" == "Hi" only returns true if the two Strings are really the same string stored at the same location in memory.

If you create a String s that contains "Hi" but is stored at a different location in memory,

then s == "Hi" will return false.

The **equals** method is provided to examine the contents of the object to determine if they are structurally equal. However, the **equals** method will not compare the contents of objects by default. The equals method in Object just does an == test**. Instead, we must override the method.** For example, the String class overrides the equals method to compare the individual characters of the string. Thus "Hi".equals("Hi") always returns true.

IMPORTANT: To override any method, we must exactly match the method parameter signature. Many Java textbooks and on-line references do not give the correct way to write an equals. If you seach online, you will often find the suggestion to use

```
public boolean equals(Employee e) {
```

but this does not match the parameter signature of the method in Object, and so it is overloading, and not overriding!

The problem with overloading is that we can change the version of the method we call by typecasting:

```
Employee e1 = new Employee(10, "Mekayla");
Employee e2 = new Employee(10, "Mekayla");
```

e1.equals(e2) and e1.equals((Object)e2) will call different versions of the method! One for input Employee, and one for input Object.

This is bad because we don't want the behavior to change just because the current type of the object changes. (In fact, the first would return true, and the second will return false!)
Because overriding is such an important idea in Java and because it is easy to make a mistake and accidentally overload instead of override, the Java compiler provides an "**annotation**" we can add to the code. An annotation is not part of the Java language, but it is a directive to the compiler.
This particular annotation is **@Override**, and if we place it before a method that is overriding an inherited method, the compiler, upon seeing the annotation, will verify that we really are overriding. If we are not, it will give a compiler error instead of allowing the overload.
A correct equals method that says this Employee is equal to the input if the input is an Employee with the same employee number and the same name:

```
@Override
/* Change the behavior of the inherited equals method. Two employee instances are the same if
they have the same number and name */
public boolean equals(Object o) {
if (o instanceof Employee) {
        Employee e = (Employee)o;
        return this.getNumber() == e.getNumber() && this.getName().equals(e.getName());
}else
        return false;
}
```

instanceof: returns true if object that is the left operand can be typecast as the type that is the right operand. (I.e. is the object's true type equal or "below" the given type in the hierarchy.)
If we do not use **instanceof**, the method will generate a **TypeCastException** when typecasting a non-Employee to Employee.

This equals method has overridden the inherited equals method. And now there is only one version of equals available to Employee, and no matter how Employee is typecast, it will always use this version of the method.

Fields / Variables:

There are 4 kinds of variables, and the kind you create depends on where you create it:

1) class (static) fields: the variable is stored in the class (one copy that all the objects share)
- class fields exist for the duration of the program.
- they are given default initial values of 0, 0.0, false, or null, depending on their type
- null = "not a valid address"

2) instance (non-static) fields: a separate variable stored in each instance of the class. Every instance has its own version.
- an instance field exists as long as the containing instance exists
- they are given default initial values of 0, 0.0, false, or null

3) method parameter: the variable(s) that store an input to the method. It exists only in the body of the method.
- the initial value is set by the method call input

4) local variables: a variable declared inside a method. It exists from the moment created until the end of the compound statement it is declared it.
- they are not given an initial value, and they must be assigned a value as their first use