Lecture Notes for Week#4

Recap of Week #3:

Java Programming:

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 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).

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

Fields:

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:

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.)

Methods:

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

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

}

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.

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, 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.

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.

```
public void transpose() {
    this.setSize(this.getHeight(), this.getWidth());
}
```

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**.

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

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

So, here is the full code:

```
public boolean isEqualArea(JFrame input) {
        return this.getWidth() * this.getHeight() == input.getWidth() * input.getHeight();
 }
```

Week#4 Lecture

SESSION-I

Java Non-Primitive Type Rules:

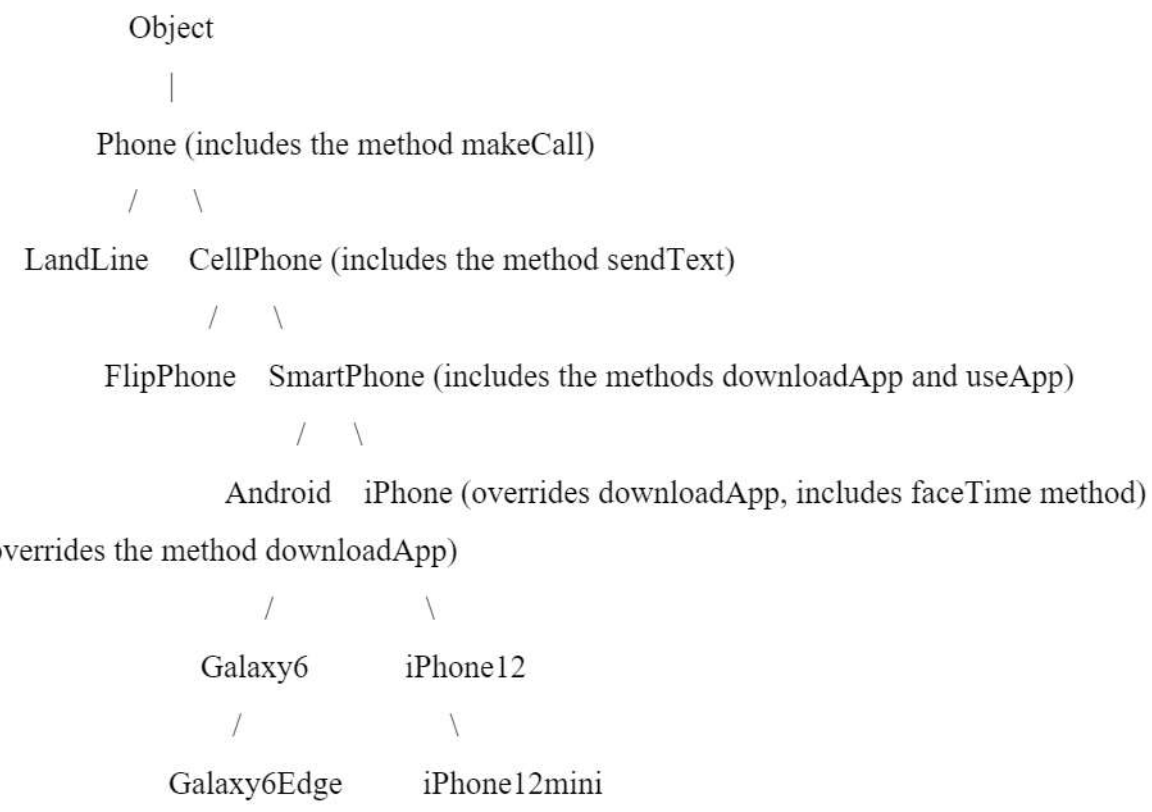
Let's create a Phone class to represent telephones:

```
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



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

SESSION-II

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.

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.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;

}
```

Conditional statements:

```
if (condition)
        then-statement
else
        else-statement
```

- condition is an expression with a boolean type
- then-statement and else-statement can be either simple or compound (but they must be a single statement)
- The "else else-statement" part is optional.

How it works:

1) the condition is evaluated

2a) if the condition is true, the then-statement is executed

2b) if the condition is false and the else-statement exists, the else-statement is executed

3) the next statement of the program is executed

Problem: A year is a leap year if it is divisible by 4 but not divisible by 100 unless it is divisible by 400

Attempt 1:

```
public boolean leapYear(int year) {
        if (year % 4 == 0) {
                if (year % 100 == 0) {
                        if (year % 400 == 0)
                                return true;
                        else
                                return false;
                }
                else
                        return true;
         }
         else
                return false;
}
```

Attempt 2: The same but now we can keep track of the cases that are "removed" to make it easier to see the logic

```
public boolean leapYear(int year) {
        if (year % 4 != 0)    // take care of the case where we know the value right away : when year is not divisible by 4
                return false;
        else {                  // from here on we know that year IS divisible by 4
                if (year % 100 != 0)      // this also removes a case where we know the answer
                        return true;
                else {                  // from here on, year IS divisible by 4 AND is divisible by 100
                        if (year % 400 != 0)
                                return false;
                        else
                        return true;
                }
         }
}
```

Attempt 3: Notice above that the "else" statement is a single conditional statement, even if it takes more than one line.In this case, we can not only remove the { }, but we can move the "if" up to be on the same line as the "else". That formatting looks a little better and is similar to languages that have the "elseif" feature. Java does not have an "elseif", but with our formatting, we get something just as nice:

```
public boolean leapYear(int year) {
        if (year % 4 != 0)
                return false;
        else if (year % 100 != 0)
                return true;
        else if (year % 400 != 0)
                return false;
        else
                return true;
}
```

Attempt 4: The same, but without an if statement. Sometimes the if statement is not even needed!

```
public boolean leapYear(int year) {
        return (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0));
}
```

Week#5 Lecture Sneak Peek

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"

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 (i.e. executes) the constructor of the parent class (the class this class extends).

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.