Lecture Notes for Week#5

Recap of Week #4:

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.

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

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.

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.

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.

Magic Numbers

A "magic number" is a number that has a special meaning to the programmer. Magic numbers should be avoided and replaced with variables.

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

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.

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;
}
```

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.

Week#5 Lecture

SESSION-I

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));
}
```

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

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.

SESSION-II

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.

Week#6 Lecture Sneak Peek

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

Introduction to Loops

Loops are a technique that allow us to execute a statement many times. This is what gives a computer program its true power because computers can execute statements in loops millions of times a second.

There are 4 basic loop structures in Java.

1. while loop
2. for loop
3. do-while loop
4. foreach loop

The first three are general purpose loops. The fourth is a special loop form that can only be used with arrays and Iterable types. We will cover foreach loops later in the term.

1. The while loop:

```
while (condition)
  loop-body-statement
```

loop-body-statement is a single statement, simple or compound, condition is any boolean expression

While loop behavior:

1) the condition is evaluated
2) if the condition is true:
2a) the loop body statement is executed
2b) repeat step 1
if the condition is false, go to the next statement of the program

2. The for loop

for (initial statement; condition; increment statements)

loop-body-statement

-the initial statement is a single statement

- increment statements are 0 or more statements, separated by commas (no terminating semicolons)

condition and loop-body-statement are the same as for while loops.

For loop behavior:

1) the initial statement is executed
2) the condition is evaluated
3) if the condition is true
3a) the loop body statement is executed
3b) the increment statements are executed
3c) repeat step 2
if the condition is false, go to the next statement of the program

3. The do-while loop

```
do {
  loop-body-statement(s)
} while (condition);
```

do-while-loop behavior:

1) Execute the loop-body-statement(s)
2) Evaluate the condition
If the condition is true, repeat step 1
If the condition is false, go to the next statement of the program

The do-while loop should be avoided except in cases where you really need the body of the loop to execute before testing the condition.

The problem with the do-while loop is that, if you accidentally drop the "do", the rest of the code is still valid Java, but it does something entirely different! It becomes a compound statement followed by a while loop!

When to use a while loop and when to use a for loop? Your choice. The two are exactly the same, just ordered differently.

Some guidelines:

* for loop advantage is that the code that describes the loop is all in the header.

  for loops are good when the loop is controlled by variable(s) and there is a simple increment.

* while loop advantage is that the syntax is simpler.

  while loops are good when the increment is complicated