You can access to the class material from the following link:

https://drive.google.com/drive/folders/1yo4pvaTngQ1L2TkHuWPCzbfapLDyvwUt?usp=sharing

You can check your lab attendance and quiz scores from the following link:

https://docs.google.com/spreadsheets/d/1xKQEqMSGIj3RBo9kLh97UzbTO1fyCvaT/edit?usp=sharing&ouid=104851561367296854968&rtpof=true&sd=true

Lecture Notes for Week#6

Recap of Week #5:

Constructors

- 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.

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.

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.

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.

We can fix this two ways:

Using super()

super() calls a constructor of the parent class. 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

SESSION-I

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.

IMPORTANT: Some Java resources suggest using the getClass() method of Object instead of instanceof, but that is not as nice a solution because getClass returns the true type and we will then be limiting our code to only run on objects with true type Employee. As a result, it won't work with HourlyEmployee! In general, we want to code using the current type as much as possible and ignoring the true type.

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

Example 1: A silly example that prints "Hello" forever:

```
while (true)
  System.out.println("Hello");
```

Note that we can put anything that evaluates to a boolean inside the parentheses.

Example 2: A loop that prints "Hello" never:

```
while (false)
  System.out.println("Hello");
```

Example 3: An example that prints "Hello" 5 times: We need to keep track of how many times we have printed "Hello". So we need a variable to store that number.

```
int count = 0; // count stores the number of times we have printed "Hello"
while (count < 5) {
        System.out.println("Hello");
        count = count + 1;
} // note: at this point in the code, count stores 5
```

What does count remember? The number of times we have printed out "Hello". Are we using count correctly?

- At the beginning, we have not printed anything and count is 0.

- Each time we print "Hello", we add one to count.

So, our use of count matches what we want it to do. What is true at the end of the loop? count stores 5 .We wanted to print 5 times, and count stores 5 so we did print 5 times.

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

Example 4: Write a loop that prints "Hello" 5 times, but this time using a for loop.

```
// numHellos stores the number of times we have printed "Hello"
for (int count = 0; count < 5; count = count + 1)
        System.out.println("Hello");

// note: at this point in the code count does not exist
```

NOTE: While loops and for loops in Java are really the same thing, just different formatting.

```
for (initial-statemnet; condition; increment-statements)
  loop-body-statement
```

is the same as:

```
{
 initial-statement;
 while (condition) {
   loop-body-statement
   increment-statements
 }
}
```

Note the encapsulation in a compound statement. This means that any variables declared in the for statement will not exist once the for statement completes.

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.

SESSION-II

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

While Loop Example 1: Euclid's Greatest Common Divisor algorithm

We want to compute the greatest common divisor of two positive integers: a and b

If a % b = 0 then the greatest common divisor is b
Otherwise gcd(a,b) = gcd(b, a % b)

This gives us a loop. For example:

gcd(30, 12) = gcd(12, 6) = 6
gcd(24, 15) = gcd(15, 9) = gcd(9, 6) = gcd(6, 3) = 3

```
public static int gcd(int a, int b) {
        while (a % b ! = 0) {
                int r = a % b;
                a = b;
                b = r;
        }
        return b;
}
```

This algorithm seems to require a > b. (It does not, trace the loop and see what happens if b > a.)

While Loop Example 2: Computing a square root using Newton's method

A little background. Please look up Newton's method if you do not understand how it works. Let x1 be a guess for the square root of n.

Let f(x) = x * x - a (Note that f(x) = 0 when x is the square root of a.

Since the plot of f(x) is concave up, we can get a new guess by following the tangent to the curve from the current guess (x1, f(x1)) to (x2, 0). x2 is the new, better guess.

Using the slope formula were m is the slope of the tangent, we have

m = (0 - f(x1)) / (x2 - x1)

Note that m = f'(x1) = 2 * x1

2 * x1 = -f(x1) / (x2 - x1) or x2 = x1 - (x1 * x1 - a) / (2 * x1)

We then repeat setting x1 to be x2 and x2 to be the new, even better, guess. **When do we stop?**

Option1: is to stop when the difference in the guesses is really small: x1 - x2 < 1e-10.

Option2: is to stop when the best guess is really close the the square root: x1 * x1 - a < 1e-10.

A good option is to stop when the "relative error" is small instead of the "absolute error" (which is what we are doing). This is a good situation for a while loop because the increment is complicated. Here is the code:

```
public static double squareRoot(double a) {
        double x1 = ???;        // the square root guess of the previous iteration
        double x2 = ???;        // the best current guess for the square root
        while (x1 - x2 > 1e-10) {
                x1 = x2;
                x2 = x1 - (x1 * x1 - a) / (2 * x1);
        }
        return x2;
}
```

What should the initial x1 and x2 be? We should set x2 to be the initial "guess" for the squareroot and set x1 to be larger. If you know how Newton's method work, the initial guess does not have to be that good, just larger than the square root and not orders of magnitude away from the square root. So, using a as the initial value will work if a >= 1. If a < 1, you don't want to use a because then your guess will be smaller than the square root. Let us use a+1.

```
public static double squareRoot(double a) {
        double x1 = a+2;        // the square root guess of the previous iteration
        double x2 = a+1;        // the best current guess for the square root
        while (x1 - x2 > 1e-10) {
                x1 = x2;
                x2 = x1 - (x1 * x1 - a) / (2 * x1);
        }
        return x2;
}
```

It is possible to shorten the code. Can you see why this works?

```
public static double squareRoot(double a) {
        double x = a+1;         // the square root guess of the previous iteration
        while (x * x - a > 1e-20) {
                x = x - (x * x - a) / (2 * x);
        }
        return x;
}
```

Also, note that you could also do:

```
public static double squareRoot(double a) {
        double x = a+1;         // the square root guess of the previous iteration
        while (x * x - a > 1e-20) {
                x = (x * x + a) / (2 * x);
        }
        return x;
}
```

Remember Strings:

Recall that String is a class that represents text. A String object can not be changed once created. Some useful methods and operators:

+: creates a new String that is the result of concatentating two Strings

length(): returns the number of characters in the String

charAt(5): returns the 6th character of the String. (In Java, the first character is at index 0.)

String s = "Hello" s.length() → returns 5 s.charAt(1) → returns 'e'
"".length() → returns 0 "Hello".charAt(0) → returns 'H' "".charAt(0) → Error
"Hello".charAt("Hello".length()) → Error because it asks for the index 5 character in a string of only 5 characters.
"Hello there".charAt(5) → returns ' ' ("hello" + "there").length() → returns 10

For Loop Example 1: Write a method that determines if a String is a palindrome (the same forward as backwards). For example, "radar" is a palindrome. The trick is to first think how YOU would do it. What are the different steps you need?

- You read the string backwards and compare with how it is read forwards.
- That means you first compare the first character with the last, then the 2nd with the 2nd to last, and so on.
- So, we need two variables: one to keep track of where we are at the start of the string, and one to keep track of were we are at the end of the string:

```
        for (int first = 0, last = s.length() - 1; ???; first = first + 1, last = last - 1)
```

What test do we need?

```
        if (s.charAt(first) == s.charAt(last))
          // then keep going
        else
            // it can't be a palindrome
```

There is nothing to do if we need to keep going. The index as to be increased, but that is being done by the increment statement if the for loop. So, our code so far is:

```
public static boolean isPalindrome(String s) {
        for (int first = 0, last = s.length() - 1; ???; first = first + 1, last = last - 1)
                if (s.charAt(first) == s.charAt(last))
                        ; // then keep going
                else
                        return false;    // it can't be a palindrome
```

The empty statement is a little awkward. Sometimes, empty statements are not easy to avoid. Here we can easily avoid it by switching the then and else statements and doing the opposite condition. Since the else is an empty statement, we can drop it.

```
public static boolean isPalindrome(String s) {
        for (int first = 0, last = s.length() - 1; ???; first = first + 1, last = last - 1)
                if (s.charAt(first) != s.charAt(last))
                        return false;    // it can't be a palindrome
```

What is the goal we want to achieve when we get to the end of the loop?

We want our loop to catch every time that s is not a palindrome, so if we get all the way to the end of the loop without returning, we know that s must be a palindrome.

```
public static boolean isPalindrome(String s) {
        for (int first = 0, last = s.length() - 1; ???; first = first + 1, last = last - 1){
                if (s.charAt(first) != s.charAt(last))
                        return false;    // it can't be a palindrome
         }
         return true;        // the loop completed without finding a mismatch, so s is a palindrome
}
```

So, what is our subgoal for each iteration of the loop?

- This is what we want to have completed once we finish the loop body and the increment statements.
- For this loop, we want the characters of s from index 0 to first - 1 to exactly match the characters from s.length - 1 down to last + 1. (Do you see why we have the +1 and -1?)

This logic lets us see when we are supposed to stop the loop.

- If we stop when first == last (and the string has odd length) then we will have tested all but the very center character.
- If we stop when first == last + 1 (and the string has even length) then we will have tested every character.

So, we stop when first >= last, and our loop condition is the opposite: first < last.

```
public static boolean isPalindrome(String s) {
        // compare the characters at 0..(first-1) against the characters (length-1)..(last+1)
        // and exit the method if we find a mismatch
        for (int first = 0, last = s.length() - 1; first < last; first = first + 1, last = last - 1)
                if (s.charAt(first) != s.charAt(last))
                return false;    // it can't be a palindrome
        }
        return true;     // the loop completed without finding a mismatch, so s is a palindrome
}
```

Note that we only need one variable to keep track of which characters we are examining instead of two. Here is the same method, but using a single index variable instead of two.

```
public static boolean isPalindrome(String s) {
// compare the first "offset" characters against the last "offset" characters, exit the method if we find a mismatch
        for (int offset = 0; offset < s.length() / 2; offset = offset + 1)
                if (s.charAt(offset) != s.charAt(s.length() - 1 - offset))
                        return false;    // it can't be a palindrome
        }
        return true;        // the loop completed without finding a mismatch, so s is a palindrome
}
```

NOTE Creating Strings:

When building a string using a loop, we may be tempted to use the + operator.

```
String result = "";
for (....) {
  ....
  result = result + c;
  ....
}
return result;
```

However, this is not a good solution. **The + operator creates a new String each time**. If we are dealing with a very long String, such as if we want to capitalize a DNA code of millions of characters, we will be creating a LOT of unnecessary Strings and using up our memory.

Java provides a StringBuilder class to create Strings. **StringBuilder** has all the same methods as String (charAt, length, etc) plus several others. Once useful one is **append** that adds new characters (or other values) to the end of the string being created.

How to create an empty StringBuilder? Just like you create any other initial instance:

```
StringBuilder result = new StringBuilder();
```

What do we return at the end? We can't return result because it is not a String. But Object has a method that returns a String representation, and every class inherits it from Object.

```
result.toString();
```

For Loop Example 2: Write a method that capitalizes every lower case letter in a string.

How would we do it? Look at each character one at a time.

- If it is lower case, capitalize it.
- If it is not lower case, keep it.

Note that we can not change the String so we must create a new String that is the same as the original, but with capital letters.

Finally, how to we test if letters are lower case, and how to make them upper case?

- Remember that char is a primitive type, and a char is just a number.
- Let us treat them as numbers.

That means we can test to see if the "number" of the letter is in the proper range.

- We can also use math to "shift" a value from one range to another.
- We must remember that that applying a primitive arithmetic operator will result in an int type so we will need to remember to typecast back to char.

(It is also possible to do this using built-in Java methods, but the math trick will work in a lot of languages.)

```
public static String capitalize(String s) {
        StringBuilder builder = new StringBuilder();
        for (int index = 0; index < s.length(); index = index + 1) {
                char c = s.charAt(index);
                if (c >= 'a' && c <= 'z')
                        c = (char)(c - 'a' + 'A');
                builder.append(c);
        }
        return builder.toString();
}
```

Week#7 Lecture Sneak Peek

The Java API (Application Program Interface)

API is typically part of any pre-defined software package that you can use in your programs.
It lists how you are to use the predefined programs.
The Java API lists all the pre-defined classes (and other types) in Java.
In the API you can find:

- the package the class is in (what you need to import)
- the header for the class
- a list of all non-private inner classes, constructors, fields, and methods

Suggestion: Keep the API bookmarked because we will be using it a lot in the course.

RULES FOR O-O CODING:

1. Create private fields
2. Create public getter/setter methods so that classes that extends this class can change behavior if they want.
3. Everywhere in our code that uses a value, we use the getter/setter methods instead of the fields.
4. Except in the constructor where, if we want a field to be initialized, we use the field directly.

O-O TYPE RULES:

1. Every instance is many types at the same time.
2. The "true" type is what the instance is at creation (from: new XXX() -> the true type is XXX).
3. The "current" type is what the instance is typecast as at this particular part of the code.
4. An instance may only call methods and fields that exist for the "current" type.
5. However, the version of the method used is the version of the "true" type. (This applies only to instance methods! Fields and static methods are still used from the current type)

Today we will see why those rules greatly simplifies our coding.
Example: Let's create a method in Employee that compares employee's by how much money they make. We created the method **earnsMoreThan** that compares the salaries of this employee to another employee.o Suppose we have this:

```
public boolean earnsMoreThan(Employee e) {
        return this.getSalary() > e.getSalary();
}
```

There is a problem: whoever created Employee assumed that all employee's have salaries. This is not the case, but we often have situations in Java where the person creating a type does not think of every situation and makes incorrect assumptions on how it is used. Because the earnsMoreThan method is following proper O-O coding (using the getter methods instead of the fields), it is easy for other classes to adjust so that their classes properly work.

We will have every employee type define for itself what "salary" means.
For example, an hourly employee can decide that a salary is hours worked * rate per hour.
A sales employee can decide salary is number of sales * commission, and so forth.
HourlyEmployee will "define" how salary works by overriding the salary method:

```
        public double getSalary() {
          return this.getHoursWorked() * this.getHourlyRate();
        }
```

Because the true type version of an instance method is always used, the hourly employee will always report its salary as the product of its hourly rate and hours worked.

- it does not matter if we typecast the hourly employee
- it does not matter if we try to set the salary of the hourly employee It just works!

BAD IDEA: Use the fields directly:

```
    public boolean earnsMoreThan(Employee e) {
       return salary > e.salary;
    }
```

If we did this, it would take more work to get hourly employee to work with the method. Since fields are not inherited and cannot be overridden, we are going to have to make sure hourly employee calls the inherited setSalary method everytime that its hourlyrate is changed or its number of hours worked is changed.
Not only is that going to mean changes in at least two different places, it means that if Employee changes how earnsMoreThan is computed, we will have to change HourlyEmployee!

The moral: everytime you choose to not follow the O-O guidelines, you make it more challenging for other coders to extend and use your class. So if you want to violate the O-O guidelines, you should think carefully to make sure you have a good reason to do so.