CSCI 1933 Lab 2
Basic Java Skills

Introduction

Welcome to the first exclusively Java lab of CSCI 1933! This lab will introduce you to some basic Java skills. Many of these steps you will have seen from last week's lab, and will be using a lot in the future, such as creating a Class and compiling your code. We'll also go over the basic usage of the `Scanner` class, which helps read input from the user of your program.

Lab Rules

You may work individually or with *one* partner in lab. We suggest that you have a TA evaluate your progress on each milestone before you move onto the next. The labs are designed to be completable by the end of lab. If you are unable to complete all of the milestones by the end of lab, you have **until the last office hours on Monday** to get those checked off by **any** of the course TAs. We suggest you get your milestones checked off as soon as you complete them since Monday office hours tend to become crowded. If you worked with a partner, both of you must be present when getting checked off to receive credit. You will only receive credit for the milestones you have checked off by a TA. There is nothing to submit to Canvas for this lab.

Attendance

Per the syllabus, students with 4 unexcused lab absences over the course of the semester will automatically fail the course. Meaning if you have missed 3 labs without excuses you are OK, but if you miss a fourth lab (without a formal excuse) you would fail the course. The TAs will take attendance during lab; it is expected that students will be actively working on the lab material. *Your physical presence in lab is not sufficient to be marked as present for the purposes of attendance.*

1 Welcome to class

In your life outside of Computer Science, you encounter thousands of **objects** on a daily basis. Objects like dogs, people, trees, cars, and so on. Almost every object we encounter has a **state** and **behavior**. An object's state describes the object. Take dogs for example, they have a name, age, weight, breed, and so on. An object's behavior describes what the object can do. For example, a dog can bark, walk, eat, sleep, and so on.

Since Java is an **object-oriented language**, we can model these real-world objects via code. In order to model these objects, Java requires us to write **classes**. A class is like a blueprint for an object, it lays out the various **instance variables** (state) of an object and its **methods** (behaviors). An object is an **instance** of a class, the actual model built from the class.

Let's start by making a `BankAccount` class. Create a `BankAccount.java` file, similar to the `Application.java` file you created for last lab. It's a little empty for the moment, though. In the real world, almost every object has instance variables (states) and methods (behaviors). To make our `BankAccount` useful, let's have it keep track of our `name, password`, and `balance` – these sound like instance variables!

```
public class BankAccount {
    String name;
    String password;
    double balance;
}
```

Note: If you are coming from Python, this might look weird. Java is a **statically-typed** language. This means that you have to explicitly give your variables their types before you can use them. For example, if we want `name` to be one of our instance variables, we need type `String name;`, not just `name;`. Also, every line of code in Java will either end in a semicolon or a curly brace, so get used to that.

Next, we need to withdraw and deposit money into our `BankAccount`; these sound like methods!

```
public class BankAccount {
    String name;
    String password;
    double balance;

    public void withdraw(String enteredPassword, double amount) {
        // Only people with the right password and sufficient funds can
            withdraw
        if (password.equals(enteredPassword) && balance >= amount) {
            balance = balance - amount;
```

```
        }
    }

    public void deposit(String enteredPassword, double amount) {
        if (password.equals(enteredPassword)) {
            balance = balance + amount;
        }
    }
}
```

Note: Lines that start with `//` are comments, they don't affect how your code runs.

Note: A good programming practice is to write your methods and variable names in camel-case. If your method/variable has two or more words, e.g. `enteredPassword`, the first letter of the first word is lowercase and the first letter of each word thereafter is uppercase.

IMPORTANT: When you are checking for equality between Java **primitives**, use `==`. When you are comparing Java objects, like `String`, use `.equals()`. Here's why:

- Using `.equals()` on objects compares the content of the two objects. Suppose we have two `String` variables, `a` and `b`, both of which have the value `"hi"`. If we do `a.equals(b)`, this will return `true`. However, `a == b` will return `false`.
- Using `==` on objects compares their memory location. Every time you make a new object, it is given a unique memory address. So in the example above, `a` and `b` have different memory addresses, hence `a == b` returning `false`
- For this course, there is only one reason you should be using `==` on objects and that is to check if an object is `null`. If you attempt to call a method on `null`, you will get the infamous null pointer exception (NPE) which will cause your program to break.

Awesome, we have a `BankAccount` class! Now it's time to **instantiate** (create\model) our bank accounts.

Milestone 1:
To get credit for this milestone, raise your hand and show a TA your `BankAccount` class.

2 Your first BankAccount

The steps to instantiating a **primitive** variable are rather simple - write the type of variable, followed by the variable name, ='s, and then the value. Instantiating an object is almost identical. Write the name of the class you want to instantiate, followed by the variable name, ='s, the keyword new, the class name again, followed by `();` So to instantiate a `BankAccount`, in your main method:

```
BankAccount myAccount = new BankAccount();
```

Now that we have our first object, we can access any of its instance variables and methods through the dot operator. If you type in `myAccount` followed by a dot, you can use any of the instance variables and methods from our `BankAccount` class. For example, if we wanted to set the password of `myAccount`, we do:

```
myAccount.password = "CSCI1933 rules!";
```

and if we wanted to deposit:

```
myAccount.deposit("CSCI1933 rules!", 100.50);
```

and we can verify that it deposited the right amount by printing out the balance:

```
System.out.println("My account's balance is: " + myAccount.balance);
```

If you create another `BankAccount`, say, `myOtherAccount`, and withdraw and deposit from it, you will see that the two accounts behave independently of each other.

Construction ahead

If we were to print out the member variables of `myAccount` before we set any of its member variables, we see something weird called `null` for our String variables. The reason is because we have not given `myAccount`'s member variables their initial values, so they default to some preset value. For numbers, it's 0, for Objects, like Strings, it's null. In the real-world, whenever someone creates a bank account, the name and password already have their values. But up until now, we've been creating bank accounts without a name and password and then setting their name and password **after** their creation.

Fortunately, we can force our objects to have initial values upon instantiation with **constructors**. The constructor signature is rather simple:

```
[access modifier] [name of your class]([Type] [parameter], ...) {
    // Any initialization goes here
 }
```

So if we wanted to initialize all three member variables in our `BankAccount`:

```
public BankAccount(String initName, String initPass, double initBalance) {
    this.name = initName;
    this.password = initPass;
    this.balance = initBalance;
}
```

Two things to note:

- The `this` keyword is unnecessary here. Are there any reasons it might be required?
- Your constructor's access modifier will always be public for the purpose of this course.

By having the above constructor, we can now force every `BankAccount` instance to be created with initial values. If you go back to your main method, you have to fill in what is between the parenthesis for each `BankAccount` instantiation. For example:

```
BankAccount myAccount = new BankAccount("Java", "CSCI1933 rules!", 100.50);
```

If you now print out the values of the member variables right after instantiation, you will see that they have been given the initial values you passed into the constructor.

Note: If you don't have a custom constructor, Java, by default, will implicitly include a null constructor. A null constructor is like a custom constructor but without any parameters and code in its body. We will touch on this in a future lab, but Java allows for multiple custom constructors alongside the null constructor so long as they meet the **overloading** constraint. Feel free to look this up in your own time. Why might having multiple constructors be beneficial?

Security concerns

There is something odd about our bank accounts. It is great that we added a layer of security by requiring a password to withdraw and deposit from our accounts, but couldn't someone change our password without us knowing? For example: `myAccount.password = "1 h4xed y0ur pazwrd n00b";` This is bad, and we need to do something about this.

Luckily, Java has **access modifiers**:

- `public` - Anyone can access and use this instance variable/class/method
- `private` - Only the class itself can access and use this instance variable/ method
- `protected` - Only classes within the same package can access and use this instance variable/-class/method
- `package-protected` - Only classes within the same package, excluding sub-classes, can access and use this instance variable/class/method

For the purposes of this course, you only have to worry about the first two.

Since we do not want our name, password, and balance to be open to the public, we need to make them private. It is as simple as typing in the keyword private before each of the types of the instance variable, e.g. `private String password;` Now if you go back to your main method, you will see that you cannot publicly access your private **member variables** (instance and member variables are interchangeable).

Now you might be wondering, "if my member variables are private, how do I get and set them in the first place?" To do so, we use **getter** and **setter** methods (formally known as **accessor** and **mutator** methods, respectively). Here are a few examples:

```
public double getBalance(String enteredPassword) {
    if (password.equals(enteredPassword)) {
        return balance;
    } else {
        return -1;
    }
}

public boolean setPassword(String oldPassword, String newPassword) {
    if (password.equals(oldPassword)) {
        password = newPassword;
        return true;
    } else {
        return false;
    }
}
```

A few things to note:

- You do not need a getter and setter for each member variable, only those that require them
- For a majority of this course, your getter and setter methods will be as simple as:

```
public void setName(String newName) {
    this.name = newName;
}
public String getName() {
    return name;
}
```

- The `BankAccount` getter and setter example above are just to show you how they can be more elaborate than just one line of code.

Now that you have the appropriate getters and setters, go and fix what's in your main method!

Milestone 2:
To get credit for this milestone, raise your hand and show a TA your updated `BankAccount` class, updated main method, and finally run your main method for them.

3 Working with Scanner

Over the course of this lab, you may have been slightly bothered by how often you are required to recompile. Every time you change your code, even if it's just to slightly change, say, your BankAccount password, you'll have to recompile. There is, however, a better way to do this. We can use the Scanner class.

First off, you'll need to import the Scanner class. This is what allows you to use the class in your program. To do this, at the very top of your file, even above the public class BankAccount line, you should add this line:

```
import java.util.Scanner;
```

This tells your computer to add the Scanner class to the resources available in your program. Then we can actually use it in our program. For now, we'll just use the absolute basics of the Scanner class. Inside your main method, add this:

```
System.out.println("Type something, then press enter");
Scanner myScanner = new Scanner(System.in);
String input = myScanner.nextLine();
System.out.println("You input " + input);
```

This might not make a lot of sense, initially, so let's go through the two new lines. The first new line is the instantiation of our Scanner class.

```
Scanner myScanner = new Scanner(System.in);
```

That's pretty standard for an instantiation. We declare our variable to be a Scanner, then, using the new keyword, we create a new Scanner object. However, you may notice the argument we pass in is a little bit strange. That System.in argument is telling our new Scanner object to read from the user's input. We'll go into other uses of Scanner later in the course, but for right now, this is likely the only argument you'll pass in.

```
String input = myScanner.nextLine();
```

This is where we finally get user input. The program will wait on the nextLine() method until something is input, at which point it will assign that to the input variable as a String. Note that the nextLine() method will **always** return a String. There are other methods for other types, such as nextInt() or nextDouble.

Milestone 3:
Modify your main method to ask your user for their password using the Scanner class, then attempt to print out their account balance. Use your getBalance() method. Show a TA your main method functioning with and without entering the correct password.

4 Honors Section

Note: This section and milestone are only required for students in the honors section. *Students in other sections do not need to do this, but are still encouraged to work on it if interested and time permits.*

Sometimes we want to pass in multiple pieces of information to the scanner, or convert that input to numeric types (e.g. a double).

We can separate a string into an array by choosing a delimiter and calling

```
String[] splitstring = some_input_string.split(" ");
```

then convert a string using

```
double aDouble = Double.parseDouble(some_input_string);
```

After asking the user for their password, ask them what it is they would like to do via keywords: deposit, withdraw, or getbalance. If it is deposit or withdraw, the input should be followed by a number and the respective function of BankAccount should be called. After processing the command, print the current balance.

Here is some example output with a starting balance of 100:

```
What is your password?
abc123
What do you want to do? (deposit, withdraw, getbalance)
withdraw 1000
Your balance: -900.0
```

Milestone 4:
To get checked off for this portion of the lab, create a series of test cases for your function and demonstrate their correctness to a TA.