CSCI 1933 Lab 1

Basic Java Skills

Introduction

Welcome to the first lab of CSCI 1933! This lab will introduce you some basic Java skills. Many of these steps you will need to repeat in future labs like creating a Class and compiling your code. You will be using the CSE Labs machines to work on your assignments, unless you bring your own computer. You should be familiar with them from the prerequisite course, but just in case, you use your x500 and corresponding password to sign-in.

A CSE Labs account is required if you plan on using the CSE Labs machine, so if you do not have one, you can submit an application online or go to the operator office, pick up a form, and have one of your TAs authorize your account.

The Java SDK

- If you are planning to use your own machine to work on assignments, make sure you have Java SDK 1.7 or 1.8 installed on your machine. There is a link to SDK 1.8 on Moodle and instructions on how to set it up.
- If you plan on using both your machine and the CSE machines (i.e. transferring files via usb or google drive), you must use SDK 1.8 since the CSE machines have SDK 1.8 installed.
- Although Java is extremely portable, we recommend you to port your code over to the CSE machines before submission and make sure that it compiles and runs on the machines. For labs, you do not have to worry about this since labs are a “checkoff process.” For projects, however, you should verify that your code runs on the machines.
  - If you find that your code runs on your machine but doesn’t on the CSE machines and cannot figure out why, ask a TA for assistance.

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 **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 **class**es. 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 try to make a `BankAccount` class together. For the purposes of this course, every Java class we write will be in its own file. First, we need to set up your file structure. It's up to you how you want it to look, but we recommend the following:

1. Create a `csci1933` directory (folder)
2. Create a `labs` directory in `csci1933`
3. For each lab, create a `lab#` directory in `labs`, in today's case, `lab1`

Next, we need to make a new class. To do so,

1. Open your favorite text editor (gedit, vim, emacs, notepad++, sublime, etc.) and make a new file
2. Save your file as `[classname].java`, in this case since we are making a `BankAccount` class, type in `BankAccount.java` and save the file in your `lab1` directory.

Great, we have our `BankAccount` class, but it is looking a bit... empty. Remember, almost every object in the real-world has instance variables (state) 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!

```java
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;`

One other thing, every line of code in Java will either end in a semicolon or a curly brace, get used to it.

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.

2 The main idea ★

Every Java application you write in this course will need a **main method**. The main method is like an entry point for your application to run. The main method can be placed anywhere in any of your classes. For example, you can place a main method in your `BankAccount` class, but for this lab, let's put it in a different class to show that a main method can be separated from your `BankAccount` code. First, create an `Application` class. In `Application`, type in the following **method signature** with the appropriate opening and closing curly brace:

```
public static void main(String[] args)
```

Do not worry about what `static` and `String[] args` does for now, we will go over that in future labs. To run your first Java application,

1. Open your terminal/command line and navigate to the directory where your java file is. If you don't know how to navigate via command line, refer to lab0, there is also a small primer on Moodle
2. Run `javac Application.java`
3. Now run `java Application`

The output may seem underwhelming at first, after all, your main method is empty. Just to exhaust this overused example, type in the following into the body of the main method and run it again.

```
System.out.println("Hello world!");
```

Before we get to instantiating our first `BankAccount` object, let's get a bit more comfortable with declaring and using variables in Java first.

- Make two integer variables, the first storing the current year and the second storing the year you were born. Compute the number of days between the two years, ignoring leap years, and print out the result.
- Make two integer variables, numerator and denominator. Make the numerator odd and the denominator even. Print out their quotient. Did it give you a number you were expecting?
- Make two `float` variables with the same numbers as numerator and denominator. Print out their quotient. Did it give you a number you were expecting? What happens if you mix and match (i.e. dividing the `int` numerator by the `float` denominator)?
- Make two `String` variables storing your first and last name. Make another String variable storing your full name by concatenating your two variables. **Hint:** I wonder if we can add two Strings together...
- Print your three String variables in the following format without changing their values and making new String variables:

```
My first name is - Hello
My last name is - World
My full name is - Hello World
```

- Print out your full name followed by the number of days between the current year and the year you were born in using a single `System.out.println()` statement. The format should look like:

```
Name: Hello World. The number of days between 2000 and 2017 is 6205
```

3 Your first BankAccount ★

Let's get down to business and instantiate our first `BankAccount`. The steps to instantiating a **primitive** variable is rather simple, it is the type of variable, followed by the variable name, `=`'s, and then the value. Instantiating an object is almost identical, it is 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`:

```
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. When we compile our `Application` class this time, we need to add an additional argument to out compiling command.

1. Open your terminal/command line and navigate to the directory where your java file is.
2. Run `javac Application.java BankAccount.java`
3. Now run `java Application`

We add our `BankAccount` class into the compilation argument in order to tell java what class to use in our `Application` class.

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 withdrawnd deposit from our accounts, but couldn't someone change our password without us knowing? For example: `myAccount.password = "1 haxed y0ur pazwrd n00b";` This is bad, 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!

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 initPassword, double initBalance) {
    this.name = initName;
    this.password = initPassword;
    this.balance = initBalance;
}
```

Two things to note:

- The `this` keyword is unnecessary here, see if you can figure out when you would use the `this` keyword.
- 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:

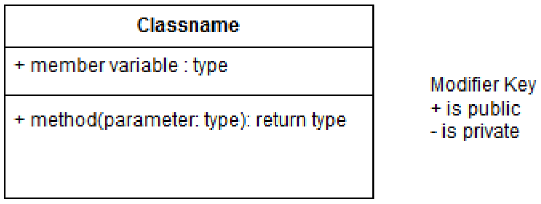
```
BankAccount myAccount = new BankAccount("Java", "CSCI1933 rules!", 100.50);
```

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.

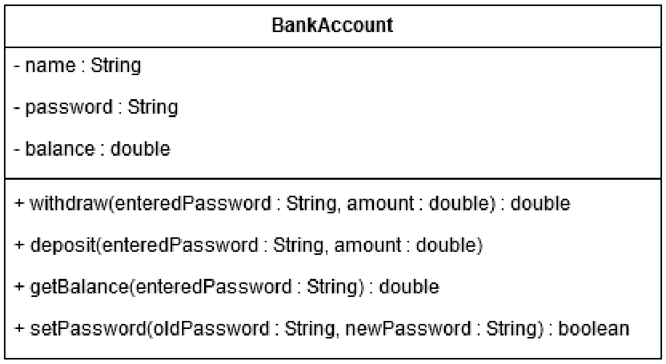
Note: If you don't have a custom constructor, Java, by default, will implicitly include a null constructor. A null constructor is like a custom constructor but without any parameters and code in its body. We will touch on this in a future lab, but Java allows for multiple custom constructors alongside 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?

4 Representing Shapes

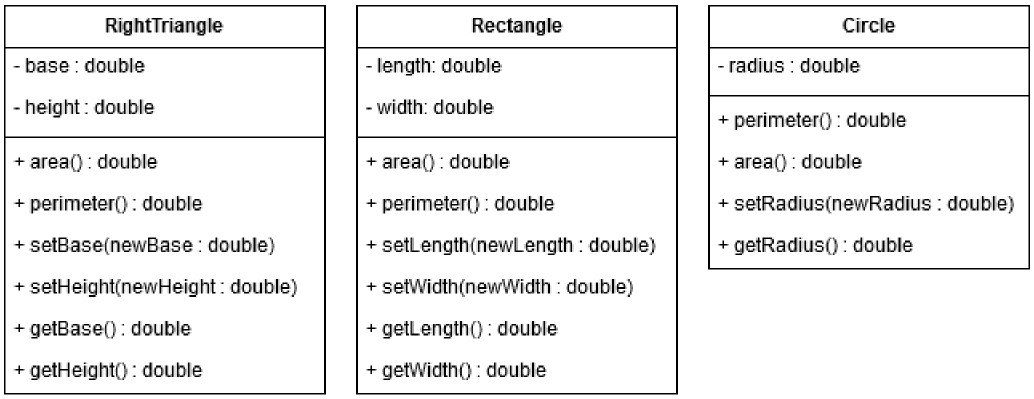
Now that you have learned the basics of creating a class, instantiating an object, and running a Java application, let's put your new-found skills to the test! We want you to model the following three shapes: **RightTriangle**, **Rectangle**, and **Circle**. In some labs and projects, we will use a **UML diagram** to list out the minimum requirements for your classes. For this course, the UML diagrams are rather simple, here is an anatomy of a UML diagram for a single class:



Using our `BankAccount` class as an example:


BankAccount
- name : String
- password : String
- balance : double
+ withdraw(enteredPassword : String, amount : double) : double
+ deposit(enteredPassword : String, amount : double)
+ getBalance(enteredPassword : String) : double
+ setPassword(oldPassword : String, newPassword : String) : boolean

So, the minimum requirements for the three shape classes are as follows:



If you are unclear as to what a method is supposed to do, ask a TA for clarification. You might want to check out the next page before you start programming.

4.1 Reading – A programmer's perspective

As you get more familiar with programming, you will find that you need to rely on various **APIs** to provide extra functionality. For example, if we want to compute the hypotenuse of a right triangle, we need to use the square-root function. Instead of having to write one yourself, you can use the `Math` API to your advantage.

There are tons of APIs available to us, the only hard part is determining which one serves our need. Fortunately, that's a simple Google search away. The next hard part is determining what methods in the API we need and how to use them. To do so, we read the API documentation. The `Math` documentation can be found here. You can typically find an API's documentation by searching "[API] documentation" where you replace [API] with the API you want to find the documentation for.

Caution: Some APIs are third party which requires you to download their API and then link them during compilation. You **do not** have to do this for this course, so don't download third party APIs.

4.2 "Testing"

An important part of writing production code, or any code for that matter, is to test your code. In this course, buggy code can lead to bad project/lab scores. One way to avoid bug-ridden code is test your code through main. That is, write code that tests your methods in your main method.

Testing your code through main is rather simple. Suppose we have a `Calculator` object called `calc` and we want to test its `add` method. The following is one way to test it:

```java
int result = calc.add(2, 3);
int expected = 5;
System.out.println("Testing Calculator Add");
System.out.println(result == expected);
```

When we go to run our main method, we would then see `true` or `false` indicating whether the test passed or not.

Write a few code snippets to test the methods of one of your shapes (you might need to use a calculator determine the expected values)