

# COMP 1406

## Winter 2020 - Tutorial #2

---

### Objectives

- Learn how to initialize your own objects with constructors
  - Practice writing instance methods to manipulate objects
  - Writing your own toString() methods
  - Dealing with NullPointerExceptions
- 

### Problem 1 (Basic Class Creation and Manipulation)

Create a new **Java Class** called **Customer**. The code for the **Customer** class should simply be:

```
public class Customer {  
  
}
```

Let's make the **Customer** object more interesting by adding some attributes (also known as *instance variables*). Assume that you want to specify the name of the **Customer**. Create a class called **CustomerTestProgram** as follows:

```
public class CustomerTestProgram {  
    public static void main(String args[]) {  
        Customer c;  
  
        c = new Customer();  
        c.name = "Bob";  
        System.out.println(c.name);  
    }  
}
```

The dot `.` after the variable `c` means that we are trying to access a piece (or attribute) of the object - in this case the customer's **name**. The code generates a compile error which says:



```
cannot find symbol  
symbol:   variable name  
location: variable c of type Customer
```

This error occurs because you have not told JAVA that your **Customer** objects are supposed to have names. You need to go back to your **Customer** class definition and add an attribute called **name**.

Go to your **Customer** class. Add an instance variable (or attribute) called **name** which is of type **String** as follows:

```
public class Customer {
    String name;
}
```

Go back and run the **CustomerTestProgram** class. It should work now, displaying **Bob**.

Add 2 more attributes (instance variables) to keep track of the Customer's **age** (an **int**) and **money** (a **float**). Now test the following code:

```
public class CustomerTestProgram {
    public static void main(String args[]) {
        Customer c;

        c = new Customer();
        c.name = "Bob";
        System.out.println(c.name);
        System.out.println(c.age);
        System.out.println(c.money);
    }
}
```

The code should compile and run. Notice the values of the **name**, **age**, and **money** variables. In JAVA, all unassigned variables are set to a default value (e.g., 0 for numbers, null for objects). Try assigning values to the variables so that **Bob** is a **27** year old with **\$50**. Then re-run to see the results.

## Problem 2 (Constructors)

Now let's see how to initialize our objects by using constructors. Create a class called **CustomerConstructorTestProgram** with the code shown below:

```
public class CustomerConstructorTestProgram {
    public static void main(String args[]) {
        Customer c1, c2, c3;
        // Create Bob
        c1 = new Customer();
        c1.name = "Bob";
        c1.age = 17;
        c1.money = 10;

        // Create Dottie
        c2 = new Customer();
    }
}
```

```

        c2.name = "Dottie";
        c2.age = 3;
        c2.money = 0;

        // Create blank customer
        c3 = new Customer();

        System.out.println("Bob looks like this: " + c1.name + ", " +
            c1.age + ", " + c1.money);
        System.out.println("Dottie looks like this: " + c2.name + ", "
            + c2.age + ", " + c2.money);
        System.out.println("Customer 3 looks like this: " + c3.name +
            ", " + c3.age + ", " + c3.money);
    }
}

```

Compile and run the code. Notice that we explicitly specified the name, age, and money for both customers. This is called *initializing* the object and it is usually done at the point that we create the object. Unfortunately, it takes **3** lines of JAVA code to initialize each customer. This is a little annoying and tedious, but recall that if we did not set the values, they would be set to zeros.

A better way to initialize an object with less code is to create our own *constructor*. Recall that a constructor is a special block of code that takes **0** or more *parameters* and it is called automatically when we create a new object. Let's make a constructor that takes a single name parameter. Write the following code in your **Customer** class just below the list of instance variables:

```

public Customer(String initName) {

}

```

Go to the **CustomerConstructorTestProgram** and re-compile it. You should get 3 errors like this:

```

!      java: constructor Customer in class Customer cannot be applied to given types;
        required: java.lang.String
        found: no arguments
        reason: actual and formal argument lists differ in length

```

What happened? Well, because you created your own constructor, JAVA decided that you are no longer allowed to use the default constructor (i.e., the one that it provided to you before you wrote your own). Notice that your test code calls **new Customer()** which takes no parameters. However, the constructor that you just made requires a **String** parameter, so JAVA does not recognize it.

You can call your new constructor by supplying a **String**, which should be the name of the customer. So, change the line **c1 = new Customer();** to **c1 = new Customer("Bob");** and re-compile. You will now see that only 2 errors remain, because JAVA now understands that you are calling your newly created constructor in the first line. Fix the remaining two errors by supplying two Strings **"Dottie"** and **"Jane"** to the other two constructor calls. The code

should now compile. However, the code from your CustomerTestProgram will no longer work as well. You will need to supply a String name for that constructor call as well.

Of course though, your constructor does not really do anything yet. Assume that we moved the line `c1.name = "Bob";` into the constructor within the Customer class. Can you explain why the code below would not compile?

```
public Customer(String initName) {  
    c1.name = "Bob";  
}
```

Right! `c1` is not defined here in the `Customer` class because `c1` is just a *variable* that we are using in our test program. Change the constructor code to be as shown below, then the code will compile:

```
public Customer(String initName) {  
    name = "Bob";  
}
```

Go back to the test code and remove the following 2 lines that set the name:

```
e1.name = "Bob";  
e2.name = "Dottie";
```

Run the test code and look at the output. You should notice that everyone is called "Bob". That is because our constructor for the `Customer` object assigns the name attribute to be "Bob".

Change your code to use the parameter `initName` instead of the fixed String `"Bob"` as follows:

```
public Customer(String initName) {  
    name = initName;  
}
```

Now run the test code again. The three unique customer names should now be correct.

In a similar way, we can set the initial values for all of the other attributes as well. You can create as many constructors as you want, as long as their list of parameters is different. It is always a good idea to set ALL of the attribute values in each of your constructors. If the attribute value is not supplied as a parameter, set the attribute to some default value.

Properly complete your `Customer` class constructor as follows:

```
public Customer(String initName) {  
    name = initName;  
    age = 0;  
    money = 0.0f;  
}
```

Run the test program again. You should notice that customer **c3** has the same default values now as indicated in the constructor. Why don't customers **c1** and **c2** have these default values? Do you know? Make sure to ask the TA if you don't understand.

## Problem 3 (More Constructors)

Create 3 more constructors with parameters as follows (make sure to complete the code in each one so that each one has exactly 3 lines of code that set the values of all 3 attributes to proper values):

- One that takes the name and age of the customer:

```
public Customer(String initName, int initAge) {...}
```

- One that takes the initial name, age, and money values of the customer.

```
public Customer(String initName, int initAge, float  
initMoney) {...}
```

- One that takes *no parameters* (i.e., uses default values that are reasonable)

```
public Customer() {...}
```

Make sure to write the missing code in all three of the above constructors. Then change your test code to look as follows (copy/paste it):

```
public class CustomerConstructorTestProgram {  
    public static void main(String args[]) {  
        Customer c1, c2, c3, c4;  
  
        c1 = new Customer("Bob", 17);  
        c2 = new Customer("Dottie", 3, 10);  
        c3 = new Customer("Jane");  
        c4 = new Customer();  
  
        System.out.println("Bob looks like this: " + c1.name +  
            ", " + c1.age + ", " + c1.money);  
        System.out.println("Dottie looks like this: " + c2.name +  
            ", " + c2.age + ", " + c2.money);  
        System.out.println("Jane looks like this: " + c3.name +  
            ", " + c3.age + ", " + c3.money);  
        System.out.println("Customer 4 looks like this: " + c4.name +  
            ", " + c4.age + ", " + c4.money);  
    }  
}
```

This test code should compile and run. If the values are not all set properly, you likely made some mistakes in your constructor code. Notice how "clean" it looks when compared to our initial testing code that set each value explicitly on separate lines.

## Problem 4 (Adding Methods)

Consider now writing some instance methods for the **Customer** class. In the **Customer** class, create a method called **computeFee()** that returns a float:

```
public float computeFee() {  
  
}
```

Complete the method so that a customer's fee will be returned as follows. The basic adult fee is **\$12.75** and applies to anyone **18** years of age or older. Children **3** and under have no fee and anyone of age **65** or older receives a **50%** discount. Children and youths (between the age of **4** to **17** inclusively) pay only **\$8.50**.

Test your code by adding the following lines to the end of your **CustomerConstructorTestProgram**:

```
System.out.println("Bob's fee is $" + c1.computeFee());  
System.out.println("Dottie's fee is $" + c2.computeFee());  
c3.age = 23;  
System.out.println("Jane's fee is $" + c3.computeFee());  
c4.age = 67;  
System.out.println("No Name's fee is $" + c4.computeFee());
```

In the **Customer** class, create a method called **spend(float amount)** that allows the customer to spend the amount of money indicated in the parameter. If the customer has enough money, the money should be spent and the customer should have that amount of money less. If the **Customer** did not have enough money, no spending should take place. Test your code by adding the following to the bottom of your test program:

```
c2.spend(3);  
System.out.println("Dottie's money remaining is $" + c2.money);
```

Consider what would happen if you added these lines to your test program:

```
c2.spend(-80);  
System.out.println("Dottie's money remaining is $" + c2.money);
```

Can you account for this situation and deal with it appropriately?

When we call the **spend()** method, we do not really know if the operation was ignored (due to insufficient funds) or if the operation was successful. It is better to give some kind of "feedback" to the test method to indicate whether or not the spending operation was successful. One way of doing this is to simply examine the customer's money variable *before* the call and then *after* the call and compare them to see if there was a change but this is cumbersome. A better way to provide feedback to the caller of the method (i.e., the test program) would be to return a value from the method to indicate whether or not it worked. We can return a **boolean** indicating whether or not the spending was successful. Modify your **spend()** method so that it returns **true** if the spending was

successful and **false** otherwise. Create, save, compile and run the following test program to test your new code:

```
public class CustomerSpendingTestProgram {
    public static void main(String args[]) {
        Customer c1, c2, c3;

        c1 = new Customer("Bob", 25, 50.00f);
        System.out.println(c1.money);

        c2 = new Customer("Dottie", 53, 100.00f);
        System.out.println(c2.money);

        c3 = new Customer("Jane", 21, 25.00f);
        System.out.println(c3.money);

        if (!c1.spend(10.00f))
            System.out.println("Unable to spend $10");
        if (!c1.spend(4.75f))
            System.out.println("Unable to spend $4.75");
        if (!c1.spend(15.25f))
            System.out.println("Unable to spend $15.25");
        if (!c1.spend(100.00f))
            System.out.println("Unable to spend $100");

        System.out.println(c1.money);
    }
}
```

In the **Customer** class, create a method called **hasMoreMoneyThan(Customer c)** that returns a boolean value of **true** if the customer has more money than the customer passed in as parameter **c** and **false** otherwise. Add the following two lines of code to the bottom of your **CustomerSpendingTestProgram** to make sure that the answers are **false** and **true**, respectively:

```
System.out.println("Bob has more money than Dottie: " +
    c1.hasMoreMoneyThan(c2));
System.out.println("Dottie has more money than Jane: " +
    c2.hasMoreMoneyThan(c3));
```

Now we will modify our test code to make use of our **computeFee()** method to gain admission into an event (i.e., such as a circus or movie). Below is the test code that we would like to have working. Create that test program, but it will not yet compile. Notice that there is a new method called **payAdmission()** which simulates a person paying admission to the circus.

```
public class CustomerAdmissionTestProgram {
    public static void main(String args[]) {
        Customer c1, c2, c3, c4;
```

```

c1 = new Customer("Bob", 17, 100);
c2 = new Customer("Dottie", 3, 10);
c3 = new Customer("Jane", 24, 40);
c4 = new Customer("Sam", 72, 5);

System.out.println("Here is the money before going into the
circus:");
System.out.println("  Bob has $" + c1.money);
System.out.println("  Dottie has $" + c2.money);
System.out.println("  Jane has $" + c3.money);
System.out.println("  Sam has $" + c4.money);

// Simulate people going into the circus
c1.payAdmission();
c2.payAdmission();
c3.payAdmission();
c4.payAdmission();

System.out.println("Here is the money after going into the
circus:");
System.out.println("  Bob has $" + c1.money);
System.out.println("  Dottie has $" + c2.money);
System.out.println("  Jane has $" + c3.money);
System.out.println("  Sam has $" + c4.money);
}
}

```

Write the **payAdmission()** method. It should have a **void** return type. Your method should be one line long ... making use of the **computeFee()** and **spend()** methods that you wrote earlier. Compile your code, test it by running the **CustomerAdmissionTestProgram** and then examine the output to make sure that it is correct.

We will now adjust our **Customer** object so that we can keep track of who has paid for their admission and who has not. Add a **boolean** attribute (i.e., instance variable) to the **Customer** class called **admitted**. Set this attribute to **false** in ALL of your constructors. Modify your **payAdmission()** method so that this variable is set to **true** when the customer has successfully paid the proper **fee** amount. Adjust your **CustomerAdmissionTestProgram** code to display this new attribute after each call to **payAdmission()** so that you can see if it is set properly. Here is the portion of code to change:

```

// Simulate people going into the circus
System.out.println("  Bob has been admitted ... " + c1.admitted);
System.out.println("  Dottie has been admitted ... " + c2.admitted);
System.out.println("  Jane has been admitted ... " + c3.admitted);
System.out.println("  Sam has been admitted ... " + c4.admitted);

c1.payAdmission();
System.out.println("Bob has been admitted ... " + c1.admitted);
c2.payAdmission();
System.out.println("Dottie has been admitted ... " + c2.admitted);
c3.payAdmission();

```



```

System.out.println("Jane has been admitted ... " + c3.admitted);
c4.payAdmission();
System.out.println("Sam has been admitted ... " + c4.admitted);

System.out.println("  Bob has $" + c1.money);
System.out.println("  Dottie has $" + c2.money);
System.out.println("  Jane has $" + c3.money);
System.out.println("  Sam has $" + c4.money);

```

Try adjusting the **money** amounts and see what happens.

## Problem 5 (Adding a toString Method)

Now we will adjust how a **Customer** object appears when displayed. We saw earlier that a **Customer** object appears like this when displayed: **Customer@140e19d** . We change the "look" of the customer by defining the String that will be returned when the customer is displayed. We do this through the **toString()** method. Write a **toString()** method so that customers appear in this format when displayed (color added for emphasis):

```

Customer Bob: a 17 year old with $0.0 who has not been admitted
Customer Dottie: a 3 year old with $10.0 who has been admitted

```

Test your code by adding these two lines to the bottom of your **CustomerAdmissionTestProgram**:

```

System.out.println(c1);
System.out.println(c2);
System.out.println(c3);
System.out.println(c4);

```

## Problem 6 (Null Pointer Exceptions)

Let us look further now into what can go wrong with **null** objects. Add a variable **c5** to the list of **Customer** variables in the **CustomerAdmissionTestProgram**. Add the following line to the end of the program:

```

System.out.println("Customer 5 looks like this: " + c5);

```

Compile the code. You should get a compile error saying:

 Error:(36, 61) java: variable c5 might not have been initialized

Change **c5** to **c5 = null** where you declared the variable, and then re-compile. Run the test program. Did you expect **null** to be returned ? Change the last line to this:

```

System.out.println("Customer 5 looks like this: " + c5.toString());

```

Re-compile and run the code. You should see this error occur:

```

java.lang.NullPointerException

```

```
at CustomerAdmissionTestProgram.main(CustomerAdmissionTestProgram.java:36)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)
```

Notice that the error is called a **NullPointerException**. JAVA is telling you that you are trying to access an attribute or a method for an object that is actually **null** - that is, you are trying to *use* an undefined object (i.e., before you created it). Notice in the code that we did not create a **Customer** object for **c5**, we only reserved space *to hold* a Customer object (i.e., we only *declared* the variable). The error message even tells us the exact line that the error occurred at (line 36 in my case, but it may differ in your program). If you go to that line, you will see that we are indeed trying to call **toString()** on **c5**, which is **null**. **You may get lots of NullPointerException errors in this course. It ALWAYS means that something in front of the dot . operator is not yet defined. It means that you forgot to create/initializer an object somewhere in your code.**