61B Note 7: OOP

Kevin Moy

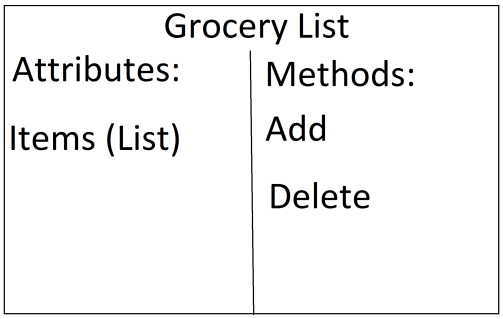
Abstract

Java is an object-oriented programming language at all levels. This note will discuss the basics and implications of this quality.

1 What is OOP?

Java is an **object-oriented programming language, or OOP** for short. This means that Java programs focus on objects having both data and being able to execute certain actions through methods. Functions in OOP are always associated with some kind of object, or type (also known as a *class*). This is in direct contrast with **function-oriented programming languages** like Python, where functions are considered separate from objects and can just be executed by themselves.

Let's take a conceptual example with a grocery list. Suppose we want to add and delete things from this grocery list. In OOP, a grocery list would be represented as an object with attributes and methods *specific* to that single object (i.e. they'd only operate on that object):



However, function-oriented programming would have standard grocery list methods as separate functions that take a grocery list object as a parameter. So these would be `static` functions:

```
static void addItem(GroceryList lst, String item) {
    lst.add(item);
}
static void deleteItem(GroceryList lst, String item) {
    lst.delete(item);
}
```

2 Another Example: Account Class (61A)

OOP was covered as a section in 61A. Their main example was the `Account` class, which represented a bank account object that held some balance and allowed deposits and withdrawals- the same exact logic as our grocery list object. Here is the code for the `Account` class as displayed in Python for 61A:

```
class Account:
    balance = 0
    def __init__(self, b):
        self.balance = b
    def deposit(self, amount):
        self.balance += amount
        return self.balance
    def withdraw(self, amount):
        if self.balance < amount:
            raise ValueError("Insufficient funds")
        else:
            self.balance -= amount
            return self.balance
```

Notice that the class contains an attribute `balance`, a constructor specified by `__init__` that initializes the balance, and two methods for deposits and withdrawals. It is fairly straightforward to convert this to Java code for 61B

purposes:

```
public class Account {
    public int balance;
    public Account(int b) {
        this.balance = b;
    }
    public int deposit(int amount) {
        balance += amount;
        return balance;
    }
    public int withdraw(int amount) {
        if (balance < amount)
            throw new IllegalStateException("Insufficient funds");
        else balance -= amount;
        return balance;
    }
}
```

The only real difference between the usage of these objects is in how they're created. For example, to create an `Account` object `acct` with initial balance of 100 dollars, Python would use `acct = Account(100)`, while Java requires the `new` keyword and type specification: `Account acct = new Account(100);`. Methods in both Java and Python are used in the same way through dot notation with created objects.

3 Classes: Breakdown

All `.java` files must define at least one `class` with the same name as the file. In doing so, we are effectively defining a new data type- a new *structured container* which contains other structured and simple containers. So when we define `public class Account`, is is our class declaration.

3.1 Instance Variables and Methods

Most classes also have some sort of **instance variables**, attributes that are specific to an object created from that class. These are also known as *fields*

or *components* of a class. `balance` is the instance variable of an `Account` object, since each account should have a specific balance.

With instance variables come **instance methods**, functions that are specific to and likely only operate on an object. Such instance methods actually take an invisible parameter `this` in Java to refer to the specific object calling that method (`this` is denoted *explicitly* in Python as `self`). `deposit` and `withdraw` are the instance methods of an `Account` object.

3.2 Constructors and Object Creation

Constructors are special object initialization methods. It is the method immediately called after an object is created with the `new` keyword. The arguments specified in object creation are immediately applied to the constructor. For example, in the `Account acct = new Account(100);` first allocates sufficient space for an `Account` object, creates this object, then immediately enters the `Account` constructor with `b` set to 100:

```
public Account(int b) {
    this.balance = b;
}
```

After this, the object is fully created and initialized, and is now free to call whatever methods it wishes.

3.3 Calling Methods and Accessing Fields

Any object can call methods defined in its class via dot notation. For example, `acct.deposit(10);` would call `acct`'s `deposit` method and deposit 10 dollars, increasing total balance to 110.

Any object can also access its fields or instance variables in the same way. However, accessing fields may be potentially limited by Java's **access specifiers** (`public, protected, private`), which may or may not allow `object.field` to be accessed. How would we access or edit them when needed then?

4 Getters and Setters

The answer is through implementing **getter and setter methods** for an object. Essentially these methods exist for the **sole** purpose of accessing (getting) and editing (setting) a single instance variable. For example, let's say we want to make our `balance` private (and we absolutely should for privacy reasons). Then, we'd have to implement getters and setters like this:

```
public class Account {
    private int balance;
    public Account(int b) {
        this.balance = b;
    }
    public int deposit(int amount) {
        balance += amount;
        return balance;
    }
    public int withdraw(int amount) {
        if (balance < amount)
            throw new IllegalStateException("Insufficient funds");
        else balance -= amount;
        return balance;
    }
    public int getBalance() {
        //some user verification code
        return balance;
    }
    public void setBalance(int newBalance) {
        //DEFINITELY some user verification code
        balance = newBalance;
    }
}
```

Why would using these methods work? `private` variables are still *in scope* of methods in the same class (`protected` variables are in scope of same class and subclasses, while `public` variables are in scope in any class). So calling these methods would access the access-protected instance variable.

So if we had `Main.java` running our `main` method and we ran `Account acct = new Account(100);` then `acct.balance` would be an illegal expression anywhere in `Main.java`.

5 Static

Static variables, also known as **class variables**, are like instance variables but the same for *all* instances of a single class.

In Java, the `static` keyword specifies static or class variables. A common example for *any* class is a counter to track how many instances of a class have been created. So for our `Account` class, we can add this as a static variable `numAccountsMade`:

```
public class Account {
    private static int numAccountsMade;
    private int balance;
    public Account(int b) {
        this.balance = b;
        numAccountsMade += 1; //this.numAccountsMade
                              //OR Account.numAccountsMade also works
    }
    (...)
    //Getter for class variable
    public static int accountsMade() {
        return numAccountsMade;
    }

}
```

We access (private) static variables in the same way as instance variables- through getter methods.

5.1 Static Methods To Instance Methods

The `static` keyword can also apply to methods. **Static methods** are like the objectless functions we saw in Python- i.e. we don't need an object created to call these methods. A very common static method you all know is

`System.out.println`: we simply call the static method from the `System.out` class in a single line.

Instance methods and static methods are completely interchangeable. For example, our `deposit` method can either be written in instance method form:

```
public int deposit(int amount) {
    balance += amount;
    return balance;
}
```

OR it could be written in static form, which would then have to specify the `Account` object to deposit to as a parameter:

```
public static int deposit(Account acct, int amount) {
    acct.setBalance(acct.getBalance() + amount);
    return acct.getBalance();
}
```

It is recommended to implement and use instance methods like a normal person because with access specifiers it can get very annoying to repeatedly use getters and setters to edit fields from a static context, as you can see above.

5.2 Calling Methods from Class

We can actually alternatively call instance methods with `Class.method(instance, args...)`. For example, instead of `acct.deposit(10);` we could instead call `deposit` from the class with `Account.deposit(acct, 10);`. Such calls expect an instance of the class as its immediate first argument. Python enforces this to be done explicitly with `self` as the first argument.

However, note that we CANNOT refer to instance variables from a static context (this is an incredibly common mistake). There IS no `this`, i.e. inherent instance, in a static method. Instead, we must use the parameter instance we pass in, i.e. `acct`.

6 Constructors

The `new` operator first allocates space for and creates a new object, then immediately calls the constructor, which is a special "initializer" method. We already saw an example in the `Account` class. However, there are some special quirks about constructors to be mindful of.

6.1 Default Constructors

All classes must have at least one constructor, as a part of adhering to the OOP concept (creatable objects). However, if you don't write an explicit constructor for a class, a **default constructor** is automatically made by the compiler. It is basically a constructor that does nothing: no arguments and no code.

```
public class Bike() {
    //This is automatically created.
    public Bike() {
        //Nothing executes.
    }
}
```

6.2 Multiple Constructors

Classes may also have *multiple* constructors. The rule here is simple: the **function signatures** of each constructor must be different, i.e. the number OR type of arguments in the constructor parameters must differ between each constructor.

```
public class Bike() {
    private int speed;
    private int numGears;
    public Bike(int s) {
        this.speed = s;
    }
    public Bike(int s, int n) {
        this.speed = s;
```

```
        this.numGears = n;
    }
}
```

Interestingly, constructors can actually call each other through `this(constructor signature)`. So the first Bike constructor can call `this(10, 10)` to call the *second* Bike constructor with `s=10` and `n=10`. Similarly, the second Bike constructor can call `this(100)` to call the first Bike constructor with `s=100`. However, this is kind of weird, and you should try your best to implement all necessary constructor initialization in a single constructor.