Classes and Objects

The fundamental building blocks of Java
• Any Questions on the project?
• Or the other stuff from class?
• Code for this slide set will be made public later
  – But today I want you to write it with me as we go.
  – Create a new project
    • You can use the command line template.
    • I’ll walk you through the rest when we begin coding after some object oriented basics.
• OOP (Object Oriented Programming)
  – Originated in smalltalk
  – Obscure language in academia in the 1970s
  – Made popular by C++ in late 1980s and early 1990s
  – Became dominant paradigm with Java in late 1990s and through 2000s
  – Now popular, but not like it was even 6-7 years ago

• Ask questions when you have them!
  – I’ve absorbed Object oriented over years, but I remember early on the resources weren’t clear and that was difficult
Main themes of object oriented programming

- Model a program as a collection of interacting objects rather than a bunch of discrete functions
- Encapsulation
  - Package up all of the ‘state’/data needed to model some object with the methods that the object needs to do.
  - Hide away a lot of the details of how an object works from those using it
- Abstraction/code reuse.
  - Through classes and inheritance (discussion deferred) we were promised code reuse.
Your turn

• So
  − given the timeline that I laid out and the reasons for OOP,
  − why did object oriented programming become such and important part of the programming landscape for those 15-25 years or so?
So

- given the timeline that I laid out and the reasons for OOP,
- why did object oriented programming become such and important part of the programming landscape for those 15-25 years or so?

- As a hint – where did most programs run in
  - The 1960s and 1970s
  - The 1980s to early 1990s
  - The late 1990s and 2000s
History of computing

- Fill in any of the gaps in the students answers.
- As a bonus, what technical change in computing made OOP less appealing?
  - After 15-25 years a lot of the flaws became apparent anyway and some people were moving away from OOP even if the technology wasn’t changing.
Programming as Modeling

- There are several useful metaphors for programming
  - Programming is like explaining things to an “Amelia Bedelia” type who takes everything literally
    - One I use for comp151
  - Programming is modeling
    - Good for object oriented.
    - In an o-o project, we write a number of classes which represent or model concepts and real world entities.
    - Need to consider carefully the domain
      - Whenever we model we have to represent just the ‘important’ stuff.
      - discuss
Example: Bank Account

• Lets use a bank account as our first example class.
• What do we need to model a simple bank account? (what kind of data)
Example: Bank Account

• Lets use a bank account as our first example class.
• What do we need to model a simple bank account?
  – Likely much of the following
    • An account balance.
    • Maybe an interest rate
    • Maybe fees
    • Maybe a customer
    • Maybe an account type
Example: Bank Account

- What kind of behavior/operations do we need to model a bank account?
Example: Bank Account

• What kind of behavior/operations do we need to model a bank account?
  – Likely includes:
    • Deposit
    • Withdraw
    • CheckBalance
    • AddInterest
    • Others?
UML

- UML (Unified Modeling Language)
  - Used to provide a diagram version of the code
  - We’ll use UML class diagrams from time to time here
  - Not used in industry as much as they used to be
    - But still useful to learners.
UML Class diagram

- A UML Class diagram gives us a nice pictoral view of a class
- Go over the parts of a class diagram using diagram below
Class vs Object

- So object oriented programming? What is this class class class
So object oriented programming? What is this class class class

In object oriented programming:

- We write/program classes.
  - Which are a description of the object we are modeling, with a description of the data and all the methods.
- Then in the program we use these classes to create objects
- Think of the class as a blueprint/ 3d printer STL file
  - And the objects are built from these files (the things you print from the STL on a 3d printer.)
Class vs Object

- The BankAccount class specifies that every bank account object will have
  - A balance
  - An interest rate.
- BankAccount Objects will all have different values for those.
To begin this series of examples

- I created a new project in intellij
- And chose the command line app template.
- That generated me a Main class

```java
package com.company;

public class Main {
    public static void main(String[] args) {
        // write your code here
    }
}
```
Now create the new class

• Open the src folder in your project
  – Right click on the package inside (com.company by default)
  – Select new class
  – Call it BankAccount
• First we will represent the data
  • Instance variables or fields
  – Any variables that are declared inside of the class, but outside of any
    methods are instance fields.
  – Should be private unless you have a reason not to
BankAccount with fields

- Below is the bankAccount so far with fields:

```java
public class BankAccount {
    private double balance;
    private float interestRate;
}
```

- Note, I’m not using the var notation because I’m not assigning values here.
  - So I have to specify the type explicitly.
Methods

- Now let’s add the deposit method
- We see it is
  - Public
  - Takes one parameter
  - And has a void return type
  - So

```java
public void deposit(double amount){
    balance+=amount;
}
```
Withdraw method

- Tell me the signature for the withdraw method
  - By looking at the UML

```plaintext
BankAccount
- balance : double
- interestRate : float
+ addInterest() : double
+ deposit(amount : double)
+ withdraw(amount : double) : bool
+ checkBalance() : double
```
Withdraw method

- Tell me the signature for the withdraw method
  - By looking at the UML
    ```java
    public boolean withdraw(double amount)
    ```
  - Note that the uml tool contracts boolean to bool (very C++ of it)
  - Now let's discuss how we should implement withdraw
Withdraw

- We want to make sure that no one withdraws more than they have in the account.
  - Maybe later we can add fees in and allow overdrafts
- If the withdraw works, return true, otherwise return false.
  - So let's write withdraw.
The remaining methods

• Lets implement checkBalance to return the value of balance.
• And addInterest to apply a yearly interest rate
  – Overly simplified, but it will do for now
  – Multiply the balance by the interest rate and add that to the original balance
  – Finally return the new balance
Accessors and Mutators

- Jargon Alert!!
  - In Object oriented programming we have special names for methods that work with instance variables
  - The instance variables/fields are private
  - So changing/getting their values requires using public methods
    - (big part of encapsulation)
  - **Mutator**: any regular method that changes the value of a private instance variable.
  - **_accessor**: any method that retrieves the value of a private instance variable.
Accessors and Mutators II

• Which methods in our BankAccount are
  – An accessor method?
  – A mutator method?
Accessors and Mutators II

- Which methods in our BankAccount are
  - An accessor method?
    - checkBalance
  - A mutator method?
    - deposit
    - withdraw
    - addInterest
Neither

- Methods which are neither accessor nor mutators
  - Often have side effects
  - For example
    ```java
    public void printBalance()
    {
      System.out.println(balance);
    }
    ```
    - doesn’t retrieve the balance for the rest of the program
    - Nor does it change the balance.
    - So neither accessor nor mutator.
Mutators

• When you are writing mutators
  – don’t just blindly set the value of the variable.
  – Mutators are sometimes called ‘setters’ because in poor programming, some people will make this style of function
    • public void setXXX(Object newXXX){
      – XXX = newXXX;
      – }
  – Like we did for the bank account (deposit, withdraw)
    • Make your mutators reflect the needs of the program
    • Not the supposedly hidden data
Accessors

- When making an accessor
  - Be careful not to expose too many of the supposedly hidden details of the object
  - It is safe to return values for built-in types and immutable instance variables
  - Be careful about mutable types
    - More later about that.
Using your object

• Assumption
  – Your main.java and bank account are in the same package
  – com.company by default in intellij if you haven’t done packages before. (most of you)
  – Now we need to use the main function in the default Main class to use our bank account.
Here is a sample use in my main package

```java
package com.company;

public class Main {

    public static void main(String[] args) {
        var account = new BankAccount();
        account.deposit(1000);
        account.addInterest();
        var succeeded = account.withdraw(2000);
        if (succeeded)
            System.out.println("You managed to withdraw");
        else
            System.out.println("Couldn't withdraw 2000 when you only have "+
            account.checkBalance() + " in your account");
    }
}
```

Discuss – note problem/error with interestRate
Default values in java

• Java (like go and unlike C/C++) assigns a default value to a variable if you don’t give one
  – Why doesn’t python have this problem?
Java (like go and unlike C/C++) assigns a default value to a variable if you don’t give one

- Why doesn’t python have this problem?
  - Because in order to create a variable, you have to give it a value in python.
- Java default values:
  - All numeric types default to zero (or 0.0)
  - boolean defaults to false.
  - Object variables like Strings and ArrayLists default to null
null in java is a reference (pointer) to an object which doesn’t exist

- Points to the zero position in memory
- Can’t compare against 0 though

If you need to check if an object exists in java

- String name;
- if (name == null)
  • //do something here.
Constructors

• What happens when you create an object?
What happens when you create an object?

```
var account = new BankAccount();
```

A special method called a constructor is called

- Constructor method
  - Has same name as the name of the class
  - Has no return type (not even void)
    - Implicit return type is an object of this class
  - If (and only if) you do not write any constructor at all, java will create one that takes no parameters/arguments
    - And sets all the instance variables to the zero value.
Your Constructor

• But we want to create a constructor
  – Lets assume a default interest rate of 2%
  – So our constructor might look like

```java
public BankAccount(){
    interestRate = 0.02f;
}
```
  – Remember that interestRate is a float.
  – By default decimal literals are doubles
  – So the f at the end of 0.02f says make this a float
More than one constructor

- We can have more than one constructor
  - So long as the parameters are different

```java
public BankAccount(){
    interestRate = 0.02f;
}
```

```java
public BankAccount(double initialBalance, float initialRate){
    balance = initialBalance;
    interestRate = initialRate;
}
```

- Word of warning. **Don’t make the parameter name the same as the instance variable name**
Every object has its own copy of the instance variable.

- You wouldn’t like it if my withdraw lowered your balance.
Creating two objects

• Lets update main

```java
class Main {
    public static void main(String[] args) {
        var account = new BankAccount();
        var account2 = new BankAccount(10000, 0.05f);
        account.deposit(1000);
        account.addInterest();
        var succeeded = account.withdraw(2000);
        if (succeeded)
            System.out.println("You managed to withdraw");
        else
            System.out.println("Couldn't withdraw 2000 when you only have " +
                account.checkBalance() + " in your account");
        account2.addInterest();
        System.out.println("The second account has " + account2.checkBalance() + "after interest added");
        account2.withdraw(5000); //note that I'm ignoring the boolean return value
        System.out.println("The second account balance after withdraw:" + account2.checkBalance());
    }
}
```

• Go over – draw UML object diagrams on board.
Updated UML

- Lets look at what our UML looks like with the new work
- Notice that UML syntax is different than java
  - Constructor return type is explicit
  - Types come after variables in UML
And the class

- Here is the updated complete class:

```java
public class BankAccount {
    private double balance;
    private float interestRate;

    public BankAccount(){
        interestRate = 0.02f;
    }

    public BankAccount(double initialBalance, float initialRate){
        balance = initialBalance;
        interestRate = initialRate;
    }

    public void deposit(double amount){
        balance+=amount;
    }

    public boolean withdraw(double amount){
        if (amount <= balance){
            balance -= amount;
            return true;
        } else 
            return false;
    }

    public double checkBalance(){
        return balance;
    }

    public double addInterest(){
        balance += balance*interestRate;
        return balance;
    }
}
```
What are we missing?

• As we are building our software model
  – We might find things need to change.
  – You remember we wanted an account ID
  – So lets add that
    • But account IDs must be unique
    • So we’ll add a way to do that.
    • First lets add an accountID instance variable
      – we’ll make it an integer to make it nice and easy.
Now what?

• Now we have our accountID instance variable

```java
public class BankAccount {
    private double balance;
    private float interestRate;
    private int accountID;

    public BankAccount(){
        interestRate = 0.02f;
    }

    public BankAccount(double initialBalance, float initialRate){
        balance = initialBalance;
        interestRate = initialRate;
    }
}
```

• But we need to make sure that every Bankaccount object has a unique accountID

• We do that by having a variable which is shared by all BankAccount objects
• Java keyword **static** means that this thing is shared by all objects in the class
  – Sometimes called ‘**class variables**’ or ‘**class fields**’ instead of ‘**instance variables**’ or ‘**instance fields**’

• **public static void main**
  – Is a function in a class, but not called on objects of that class.

• **Now we will add a static class variable**
Adding a static nextID

- Lets add a static nextID field

```java
public class BankAccount {
    private double balance;
    private float interestRate;
    private int accountID;
    private static int nextID = 1000;
}
```

- The UML diagram doesn’t have an explicit way to represent static
  - But we give the static variable a value.
  - That value is set once because it is part of the class
Unique account numbers

- Lets put this to use

```java
public class BankAccount {
    private double balance;
    private float interestRate;
    private int accountID;
    private static int nextID = 1000;

    public BankAccount(){
        interestRate = 0.02f;
        accountID = nextID;
        nextID = nextID+1;
    }

    public BankAccount(double initialBalance, float initialRate){
        balance = initialBalance;
        interestRate = initialRate;
        accountID = nextID;
        nextID++; //note that this is a shortcut for nextID = nextID+1;
    }
}
```

- Here we are using nextID to give a unique accountID to each account object
Accessors or Mutators?

- We just added a new instance variable (accountID)
  - Should we add an accessor? A mutator? Both? Neither?
  - discuss
Accessors or Mutators?

- We just added a new instance variable (accountID)
  - Should we add an accessor? A mutator? Both? Neither?
  - Discuss
- Do we ever want accounts to change account numbers?
Accessors or Mutators?

• We just added a new instance variable (accountID)
  – Should we add an accessor? A mutator? Both? Neither?
  – Discuss

• Do we ever want accounts to change account numbers?
  – Probably not – so no mutators

• Do we want to find out the account number for an account?
Accessors or Mutators?

- We just added a new instance variable (accountID)
  - Should we add an accessor? A mutator? Both? Neither?
  - Discuss

- Do we ever want accounts to change account numbers?
  - Probably not – so no mutators

- Do we want to find out the account number for an account?
  - Probably, so let's make an accessor
Adding the accessor

- Now the accessor is simple to add

```java
public int getAccountID(){
    return accountID;
}
```

- And the complete UML

- Lets be sure to look at the complete class in intellij so we don’t lose the forest in the trees.
Update main

• Lets update our test in main

```java
public class Main {

    public static void main(String[] args) {
        var account = new BankAccount();
        var account2 = new BankAccount(10000, 0.05f);
        account.deposit(1000);
        account.addInterest();
        var succeeded = account.withdraw(2000);
        if (succeeded)
            System.out.println("You managed to withdraw");
        else
            System.out.println("Could'n't withdraw 2000 from account: "+
                               account.getAccountID() + "+"
                               " when you only have " + account.checkBalance() + " in your account");

        account2.addInterest();
        System.out.println("The second account (account ID "+
                           account2.getAccountID() + ")has"
                           "+account2.checkBalance() + "after interest added");
        account2.withdraw(5000); //note that I'm ignoring the boolean return value
        System.out.println("The second account balance after withdraw:"+
                           account2.checkBalance());
    }
}
```

• I’ve added some calls (highlighted) to demo. Lets run it.
Now let's add more classes

- For a complete Object oriented program
  - Want a set of interacting objects.

- For our simple bank
  - Let's have
    - BankAccount (which we just did)
    - Customer
    - Bank
  - Then the main function will start it all off and be done.
Customer

- What data do we need for the customer?
Customer

• What data do we need for the customer?
  – Possibly
    • CustomerID (proxy for social security/tax id)
    • Name
    • A list of accounts that belong to the customer
      – For our super simple example we’ll ignore joint accounts
    • Lots more if we were doing this professionally
      – Age (for retirement savings accounts)
      – FICO score
      – etc
Customer methods?

- What kinds of methods should we have for customer?
Customer methods?

- What kinds of methods should we have for customer?
  - constructor(s) of course
  - OpenAccount (mutator – of which instance variable?)
  - CloseAccount (mutator - same)
  - Accessors:
    - getName
    - getId
    - getAccount?
  - taxIds can’t be changed – and let’s ignore legal name changes for now.
  - Anything else?
Here is the UML for our simplified customer

Now lets create the customer class

- Right click on com.company and choose new class again

you’ll need to import `import java.util.ArrayList;` to get the array list class

No need to import BankAccount

- Customer and BankAccount are in same package

But then write the three instance variables in your customer class.
The Customer has some accounts

Each account is held by only one customer. (no joint accounts right now)
Let's implement the methods

- Let's implement the customer methods now
  - Let's save closeAccount for last
  - Once we get there talk about null (trying to close account that the customer doesn’t own.)
- Then let's update the main function to add some customers and their accounts.
Now let's do the bank class

- we’ll have two instance variables

- And one public method
  - For now

- We will create private methods
  - As needed.

- doBanking will show a ‘menu’ of choices
  - Allow user to choose what to do and then go do it
  - We will need some private methods
    - ‘helper methods’ to help make doBanking work.
    - don’t want one method to get too long
    - Rule of thumb: each method not more than a screenful
doBanking

- Some of the choices should include
  - Add customer
  - Select customer for banking
    - Which should provide a new menu if that customer exists
  - Remove customer (see optional slides next for null issues)
  - Do Yearly Maintenance
  - Exit program.

- Be sure to introduce switch, which students didn’t learn in python.
One of the things we’ll discover in the course of the implementation:

- Close account (in customer) won’t go well when we close a non-existent account
- We could return null
- But then we have to deal with null
  - `var account = myCustomer.closeAccount(1111);`
  - `var finalBalance = account.checkBalance();`
  - `Java.lang.NullPointerException` error if there was no account with accountNumber 1111 for myCustomer.
Optional II

• Dealing with null the old fasioned C/C++/early java way:
  – If(account != null)
    • DoSomething();

• Since java 8 we have a better way in java
  – Optional.
  – Optional is a ‘container’ type
    • It holds a real value, or null.
    • Optional<BankAccount> is a type for ‘maybe there will be a bank account and maybe it will be null.
      • This will be the new return type for closeAccount
Optional III

- Optional has several useful methods
  - `isPresent()`
    - returns true if there is a real object, returns false if a null value
  - `ifPresent(lambda)`
    - Runs the lambda expression if there is an object
  - `IfPresentOrElse()` (introduced in java 9)
    - Runs one expression if the optional has a value, the other for null.
  - `or()`
    - If the object is present, return this optional, otherwise run a function (which should create an object)
  - `empty()`
    - Static method (called on Optional) to create optional with null content
UML class relationships

- Objects of classes are related in OOP
  - Relationships have really important properties
    - Type (how are the two related)
    - Arity (how many of each object in the relationship)
    - And maybe – direction (which object is made up of which)
• Object Oriented programming was really important for a quarter century
  – they’ve been heavily studied
  – Which means lots of jargon
  – And well known interaction patterns
  – Which is a pain for new learners like you
    • But practice makes it all easy.
    • So lets learn
    • And then practice.
• Composition Relationship
  - When object A is composed of object B, then
    • object A has an object B as an instance variable
    • And that instance variable is filled with a new object created in the object B class.
    • Uml for composition – note direction of arrow
    • Possibly build it.
    • Sometimes called
      - Has-a relationship
OO Relationship types II

• Association
  – When an Object A needs to hold on to an object B for a long time, but object B actually belongs to something else, Association
    • Object A needs an instance variable of object B
    • But that instance variable is initialized with a value passed in as a parameter to a function.
    • UML for association (note arrow direction: opposite of composition)
    • Sometimes called knows-a
    • Build it?
OO Relationship types III

- **Dependency**
  - When an object A, needs to use an object of type B
    - Object A has a local or parameter variable of type object B
    - UML (arrow is in same direction as Association)
    - Sometimes called ‘uses-a’ relationship
    - Build it?
• There are a couple of other well studied relationships
  – we’ll touch on some of them later.
  – But these are the ones we can easily implement now.
Relationship diagrams

• The arrows in a relationship diagram often display useful data
  – If A knows-a B and B has-a A, then you can use one arrow with two points.
• Arity
  – If the numbers are known you can represent them on the arrows.
    • 1..3 means one to three of this type of object
    • * means any number (zero to any)
    • 1..* means at least one, but possibly many.
Let's look at project 2
If we haven't already.

Additional Assignment:
- work with this stuff till it makes sense.
- we’ve built something pretty good in class
- If you missed any of the classes get the code and study it.
- Make sure you understand it all