6.178: Introduction to Software Engineering in Java

Lecture 4: Object-Oriented Programming

What is it?

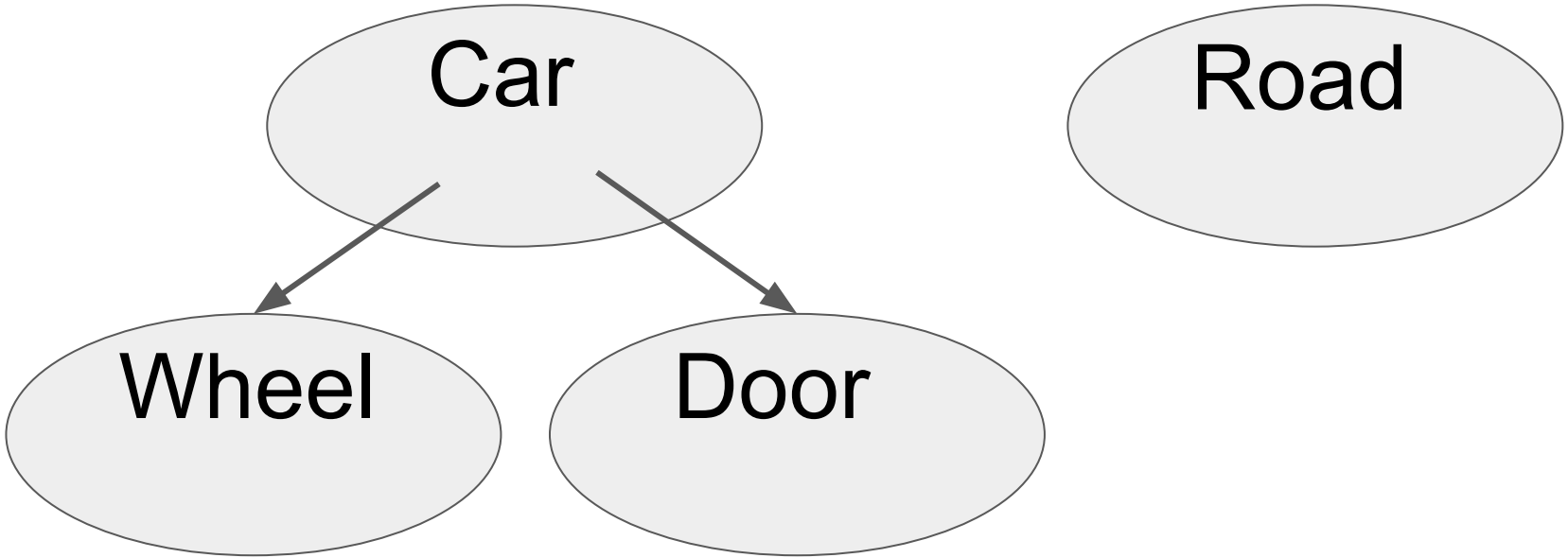
“**Object-oriented programming (OOP)** is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as *attributes*; and code, in the form of procedures, often known as *methods*.”

-Wikipedia

What is it actually?

user defined “objects” interacting with each other

abstraction of handling pieces of data and operations on them



How do we make objects?

classes!

each class is its own object

Object structure

```
public class MyObject {
    // fields (attributes) go here
    int myInt;
    String myString;

    // constructor
    public MyObject(...params…) {
        // some code here
    }

    // instance methods (procedures) go here
    public void myMethod(...params…) {
        // some other code
    }
}
```

Fields

```
String name;
int major;
ArrayList<String> currentCourses;
double gpa;
```

attributes of an object - a student has a name, major, list of current classes, and gpa

can access throughout the class like any other variable, or by saying this

```
major = 16;
this.gpa = 4.3;
```

Fields

right now, if we create an *instance* of Student, we can also access the fields

```
Student alice = new Student(“Alice”, 18);
alice.major = 6;
```

this means anyone can change these attributes of alice

how do we prevent that?

Access modifiers

public - anything anywhere can access it

protected - anything in the same package or any subclass can access it

package-private (no modifier) - anything in the same package can access it

private - only that class itself can access it

right now Student’s fields are package-private

Constructor

a constructor creates an instance of the class

they are called after using the `new` keyword

ex: `Integer a = new Integer(5);`

they look like other methods but don't specify a return type

they must be named exactly the same as the class

a class can have multiple constructors with different parameters

if you don't specify one, Java will provide a default one that does nothing

Constructor

```
public Student(String name, int major) {
    this.name = name;
    this.major = major;
    currentCourses = new ArrayList<>();
}
```

often set or initialize fields, especially when given them as parameters

if you don’t initialize an object and try to use it later, you’ll get a NullPointerException

note how fields are being called

Instance Methods

methods that can be called on specific instances of an object

ex: `list.add(5);`

the opposite of a static method that can be called without an instance of an object

ex: `String.valueOf(5);`

the same access modifiers can be applied to instance methods as fields

common ones include getter/setter methods, which return or modify an instance's fields

Instance Methods

```
public void addCourse(String courseName) {
    currentCourses.add(courseName);
}
```

includes the return type and name like any other method

just like constructors, can have methods with same name and different parameters and/or return types

don't use the static keyword

classes *can* contain a mix of static and instance methods

Object methods

every class you write automatically *inherits* methods from Object.java

- common ones: toString, equals, hashCode

these methods exist for every class without explicitly writing them

oftentimes you do want to *override* them to be useful to your class

- use the annotation `@Override` before the method to show that you are changing the behavior of the default method

Object Contract: if you override equals, you must override hashCode

- you want to do this if you want to be able to compare objects or use them in data structures like Lists, Maps, Sets, etc.

toString()

provides a way to nicely convert an instance of an object to a String

default toString returns something like Student@677327b6
Class@hexNumber

use it to give useful info about an object

remember to return a String, don’t print anything

```
@Override
public String toString() {
    return “Student: “ + name;
}
```

The Object Contract: equals(...)

on any reference values x, y, and z:

- ***reflexive*** - x.equals(x) should return true
- ***symmetric*** - x.equals(y) should return true iff y.equals(x) returns true
- ***transitive*** - if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true
- ***consistent*** - multiple invocations of x.equals(y) return the same result, provided no information used in equals comparisons on the objects is modified
- ***null comparisons*** - x.equals(null) should return false

equals(Object thatObject)

a custom equality method for Objects

default is ==

you decide what it means for objects to be equal
    all fields equal, some fields equal?

it’s important that the parameter has type Object, not Student

Common equals(Object thatObject) template

```
@Override
public boolean equals(Object thatObject) {
    if (!(thatObject instanceof Student)) {
        return false;
    }
    Student thatStudent = (Student) thatObject;
    // check for field equality, ex:
    boolean namesEqual = this.name.equals(thatStudent.name);
    boolean gpasEqual = this.gpa == thatStudent.gpa;
    return namesEqual && gpasEqual;
}
```

The Object Contract: hashCode()

on any reference values x and y:

- ***consistent*** - multiple invocations of x.hashCode() consistently returns the same integer for the duration of the application execution provided no information used in equals comparisons on the objects is modified
- ***follows equals()*** - If x.equals(y) is true, x.hashCode() == y.hashCode() must be true

if x.equals(y) is false, x.hashCode() == y.hashCode() does *not* have to be true, but producing distinct integer results for unequal objects may improve the performance of hash tables

hashCode()

returns an integer representation of that object

important for hash tables, used in HashMaps, HashSets, etc.

MUST use **SAME** fields as equals(...)

```
@Override
public int hashCode() {
    return Objects.hash(name, gpa);
}
```

Primitives vs. Objects

int, double, float, boolean, char, short, long, byte

- start with lowercase letters
- don’t have any member functions or fields

String, ArrayList, HashMap, etc.

- start with uppercase letters
- have member functions and fields

Equality

what should the following programming segment print?
(remember that = is for assignment, and == checks equality)

```
int a = 4;
int b = 4;
System.out.println(a == b);
```

what does it print? try it!

Equality

now try this one:

```
String a = new String(“abc”);
String b = new String(“abc”);
System.out.println(a == b);
```

Note: it is *extremely* important you write this exactly. Do **NOT** use the shorthand

```
String a = “abc”;
```

Equality

let’s fix that last one…

```
String a = new String(“abc”);
String b = new String(“abc”);
System.out.println(a.equals(b));
```

what’s going on here?

== vs. .equals(...)

== tests for *referential equality*

are the variables stored in the same place in memory?

.equals(...) tests for *object equality*

are the objects equal, as defined by those objects?

in general, always use == for comparing primitives, and always use .equals(...) for objects

Primitive wrapper classes

all primitives have a corresponding object representation - a wrapper class

Integer, Double, Character, etc.

can be used in place of primitives where an object is needed

ex: `ArrayList<Integer>`

always use primitives when doing iterations in for loops, not their wrapper classes

Autoboxing/Unboxing

autoboxing = conversion from primitive to wrapper

ex: `Integer a = 5;`

unboxing = conversion from wrapper to primitive

ex:
```
List<Integer> list = new ArrayList<>();
list.add(5); // also autoboxing
int a = list.get(0); // unboxing
```

Other type conversions

use *static factory methods* provided in the documentation

example: Strings!

to a String:
```
int a = 5;
String intValue = String.valueOf(a);
```

from a String:
```
String a = "5";
int stringValue = Integer.valueOf(a);
```

usually methods to convert between common types that make sense

look in documentation - Eclipse autocomplete is your friend!

Pass-by-reference vs. pass-by-value

Java is **always** pass-by-value

an object points to a place in memory where its fields and methods are stored

if you pass an object into a method, you get the same one out in the end - you can’t change its place in memory

Pass-by-value example (PassByValue.java)

```
public static void main(String[] args) {
    ArrayList<String> test = new ArrayList<>();
    test.add("original");
    changeVar(test);
    System.out.println("in main: " + test);
}

public static void changeVar(List<String> list) {
    list = new ArrayList<>();
    list.add("changed");
    System.out.println("in changeVar: " + list);
}
```

A sneaky bug

take a look at Sneaky.java

uncomment the section of code labelled Sneaky in Main.java

what happens?

add the following at the end of main in Main.java

```
List<String> sneakyList = sneaky.getMyList();
sneakyList.add("e");
System.out.println(abc);
```

Catch the bug

the first problem is this line in Sneaky.java: `this.myList = yourList;`

- this means that whatever list you pass in to the Sneaky constructor belongs to that Sneaky object
- we really just want to make a *copy* of that list, not pass in the same one

the second problem is this line in getMyList(): `return myList;`

- this returns the exact list that belongs to the sneaky instance, so we can modify it outside of the class

Kill the bug

We can solve both the problems by making *defensive copies* of the lists

change `this.myList = yourList;` to

```
this.myList = new ArrayList<>(yourList);
```

change `return myList;` to

```
return new ArrayList<>(myList);
```

run Main.java again

What's going on?

List is a *mutable* type - you can change what's in it (e.g. `list.add(object)`)

if a list is shared, modifying it in one place modifies it in the other

also applies to objects like Maps and Sets

deep copying - copying all of the objects in the List/Map/Set as well as the actual object

this can be a subtle source of bugs, so be careful!

Modifying Student.java

finish TODOs 1-6

- take a look at your constructor and methods to view the fields and make sure to not fall into the trap we just talked about

work on TODOs 7-8

add any other methods you think would be useful to you

fix any other classes that break because of your changes

play around with Student in Main.java to see what all your changes do