CSE 1223: Introduction to Computer Programming in Java
Object-oriented Programming Basics

Jeremy Morris
Object-oriented programming

You’ve all heard this “buzzword” before
  - If nowhere else, you heard it the first day of class!

What do we mean by it?
  - A form of programming based around viewing software as modular objects instead of just as procedural lines of code
  - Software as “black boxes”
Objects

- So what is an “object”?
  - In the real world, *things* are objects
  - Real world objects have *state* and *behavior*
    - State: A configurations of attributes
    - Behavior: Things that the object can do
  - Consider a car:
    - A car’s *state* is a combination of the car’s attributes (color, make, model, current speed, current direction, current acceleration, etc.)
    - A car’s *behavior* are the actions that can be performed to modify its state (accelerate, brake, turn, etc.)
Objects

- Software objects are similar to real-world objects
  - Also have a *behavior* and a *state*
    - Each object has a set of associated data values. The configuration of these values determines its *state*
    - Each object also has a set of associated methods. These methods define its *behavior*
    - We call this *encapsulation*
      - Meaning “putting things into a capsule (container)”
      - We stuff all of these things (methods & data) into one container – that’s an object
Object Example - Strings

- Java Strings are an example of an object
  - Provide both data and behavior:
    ```java
    String userName = "bob";
    ```
    - The object is a String with the name “username”
    - The data is the sequence of characters ‘b’, ’o’, ’b’
      - Alternatively, as an array of characters [‘b’, ’o’, ’b’]
    - What is the behavior? Methods!

    ```java
    int x = userName.length(); // x=3
    char y = userName.charAt(0); // y=’b’
    String z = userName.substring(1,2); // z = "o";
    ```
Object Example - ArrayLists

- Java ArrayLists are another example of an object

```java
ArrayList<Integer> list =
    new ArrayList<>();
```

- The object is an ArrayList with the name “list”
- The data is the elements in the list
- Methods provide “list” behavior

```java
int x = list.size();
list.add(0);
int y = list.get(0);
```
Object Example - Scanner

- Java Scanner is yet another example of an object

```java
Scanner in = new Scanner(System.in);
```
- The object is a Scanner with the name “in”
- The data is a bit less obvious in this case, but it is basically the connection between the input (keyboard) and the program
- Methods provide I/O behavior

```java
String str = in.nextLine();
int y = in.nextInt();
```
Classes

- So what is a Class?
  - Again consider real-world objects
  - There are many kinds of cars in the world
    - They share the same behavior
    - They may share some of the same attributes
      - Two cars with the same make, model and color might only differ in their vehicle ID numbers
  - They’re all the same “kind of thing”
    - They belong to the same “class of objects”
  - Software classes are similar
    - They define “kinds of objects” with the same behavior and same types of attributes
Classes

- **classes** work as a type of “software blueprint”
  - Used to create software **objects** for use in code
  - Creating an object from a class is called **instantiation** – (i.e. “creating an instance”)
  - Each **instance** is a separate **object**

  ```java
  String msg1 = “Hello”;
  String msg2 = “Goodbye”;  
  ```

- **msg1** and **msg2** are each **String objects**
  - The String class is the blueprint that says how to build (instantiate) these objects
Why use classes?

- User-defined types
  - Allows programmers to extend the language (almost) arbitrarily

- Code re-use
  - Many problems can be described by the same data types – why reinvent the wheel?

- Real-world problem solving
  - Thinking of problems in terms of “objects” can make it easier to model problems in the real world
Public interfaces

- Every class has a public interface
  - This the set of items that are usable by programmers
    - Classes have private elements too – we’ll talk about those later
  - Public interfaces from the Java Standard Library are described in the Java documentation
    - Strings: [http://docs.oracle.com/javase/6/docs/api/java/lang/String.html](http://docs.oracle.com/javase/6/docs/api/java/lang/String.html)
    - ArrayLists: [http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html](http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html)
    - Scanners: [http://docs.oracle.com/javase/6/docs/api/java/util/Scanner.html](http://docs.oracle.com/javase/6/docs/api/java/util/Scanner.html)
    - The documentation provides a list of methods offered by the class and a description of what each method does
Each class has two types of methods

- **Constructor methods**
  - Used only to “construct” a new instance of a class

- **Non-constructor methods (or class methods)**
  - All the other methods used by an object

### Constructor Summary

- **ArrayList**
  - Constructs an empty list with an initial capacity of ten.

- **ArrayList(Collection<? extends E> c)**
  - Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

- **ArrayList(int initialCapacity)**
  - Constructs an empty list with the specified initial capacity.

### Method Summary

- **boolean add(E e)**
  - Appends the specified element to the end of this list.

- **void add(int index, E element)**
  - Inserts the specified element at the specified position in this list.
Constructors

- A constructor method is called when the object is instantiated
  - When we declare a new `ArrayList()` for example
  - The constructor is a method and can take parameters
  - A class can have multiple constructor methods
    - Each provides different behavior when building a new instance

**Constructor Summary**

- `ArrayList()`
  - Constructs an empty list with an initial capacity of ten.

- `ArrayList(Collection<? extends E> c)`
  - Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

- `ArrayList(int initialCapacity)`
  - Constructs an empty list with the specified initial capacity.
Constructors you’ve already used

```java
import java.util.ArrayList;

... 

public static void main(String[] args) {
    ArrayList<String> stringList = new ArrayList<String>();
    Scanner keyboard = new Scanner(System.in);
    String input = "";
    while (!input.equals("stop")) {
        input = keyboard.nextLine();
        stringList.add(input);
    }
    int i = 0;
    while (i < stringList.size()) {
        System.out.println(stringList.get(i));
        i = i + 1;
    }
}
```
Constructors you’ve already used

```java
import java.util.ArrayList;
...

public static void main(String [] args) {
    ArrayList<String> stringList = new ArrayList<String>();
    Scanner keyboard = new Scanner(System.in);
    String input = "";
    while (input.equals("stop") == false) {
        input = keyboard.nextLine();
        stringList.add(input);
    }
    int i = 0;
    while (i<stringList.size()) {
        System.out.println(stringList.get(i));
        i = i + 1;
    }
}
```
Constructors you’ve already used

```java
import java.util.ArrayList;

public static void main(String[] args) {
    ArrayList<String> stringList = new ArrayList<String>();
    Scanner keyboard = new Scanner(System.in);
    String input = "";
    while (input.equals("stop") == false) {
        input = keyboard.nextLine();
        stringList.add(input);
    }
    int i = 0;
    while (i < stringList.size()) {
        System.out.println(stringList.get(i));
        i = i + 1;
    }
}
```
Constructors you’ve already used

import java.util.ArrayList;
...

public static void main(String[] args) {
    ArrayList<String> stringList = new ArrayList<String>();
    Scanner keyboard = new Scanner(System.in);
    String input = "";
    while (input.equals("stop") == false) {
        input = keyboard.nextLine();
        stringList.add(input);
    }
    int i = 0;
    while (i < stringList.size()) {
        System.out.println(stringList.get(i));
        i = i + 1;
    }
}

This last one is a special case – the Java compiler treats this as the same as:

new String("");

We call this “syntactic sugar” because it makes programming easier.
Using Constructors

- **We instantiate** objects by calling their constructors:
  ```java
  ArrayList<String> stringList = new ArrayList<String>();
  ```
- You must use the `new` keyword to create an instance of an object
- Notice the syntax after the `new` keyword:
  - Parentheses – because we are calling a constructor method
  - This method will return a new ArrayList object
Using Constructors

- Classes may have more than one constructor method
  - ArrayList (above) has 3
    - One with no arguments (the “empty” or “default” constructor)
    - One that takes a Collection as an argument
    - One that takes an “initialCapacity” as an argument
  - How do we know which one to use?
    - Read the documentation!
    - See which one fills the requirements that you need for it.
### Using Constructors

ArrayList<String> stringList = new ArrayList<String>();

- **How do we know which constructor will be called?**
  - Look at the arguments (or parameters) to the constructor method
    - Remember! Constructor is a method!
  - This one has no arguments, so the “default” constructor will be called
  - The Java compiler figures out which one you mean based on which arguments you use
Using objects

Once an object has been *instantiated* we can use it

- We use an object by making calls to its *public methods*

- Public methods can do many things, but two categories generally stand out:
  - Change the data inside an object (*mutators* or *setters*)
  - Access the data inside an object (*accessors* or *getters*)
import java.util.ArrayList;

public static void main(String [] args) {
    ArrayList<String> stringList = new ArrayList<String>();
    Scanner keyboard = new Scanner(System.in);
    String input = "";
    while (input.equals("stop") == false) {
        input = keyboard.nextLine();
        stringList.add(input);
    }
    int i = 0;
    while (i<stringList.size()) {
        System.out.println(stringList.get(i));
        i = i + 1;
    }
}
import java.util.ArrayList;

... public static void main(String [] args) {
    ArrayList<String> stringList = new ArrayList<String>();
    Scanner keyboard = new Scanner(System.in);
    String input = "";
    while (input.equals("stop") == false) {
        input = keyboard.nextLine();
        stringList.add(input);
    }
    int i = 0;
    while (i<stringList.size()) {
        System.out.println(stringList.get(i));
        i = i + 1;
    }
}
Public methods you’ve already used

```java
import java.util.ArrayList;

...

public static void main(String[] args) {
    ArrayList<String> stringList = new ArrayList<String>();
    Scanner keyboard = new Scanner(System.in);
    String input = "";
    while (input.equals("stop") == false) {
        input = keyboard.nextLine();
        stringList.add(input);
    }
    int i = 0;
    while (i < stringList.size()) {
        System.out.println(stringList.get(i));
        i = i + 1;
    }
}
```

Mutator
import java.util.ArrayList;

...

public static void main(String[] args) {
    ArrayList<String> stringList = new ArrayList<String>();
    Scanner keyboard = new Scanner(System.in);
    String input = "";
    while (input.equals("stop") == false) {
        input = keyboard.nextLine();
        stringList.add(input);
    }
    int i = 0;
    while (i<stringList.size()) {
        System.out.println(stringList.get(i));
        i = i + 1;
    }
}

Accessor
public static void main(String [] args) {
    ArrayList<String> stringList = new ArrayList<String>();
    Scanner keyboard = new Scanner(System.in);
    String input = "";
    while (input.equals("stop") == false) {
        input = keyboard.nextLine();
        stringList.add(input);
    }
    int i = 0;
    while (i<stringList.size()) {
        System.out.println(stringList.get(i));
        i = i + 1;
    }
}
Using objects

- Mutator methods (aka setter methods)
  - Used to *make changes to* (or *mutate*) an object
    - Also known as “setting values” of an object
  - Just like a static method, class methods have *parameters* and a *return type*
  - Some ArrayList mutator methods include:
    - `boolean add(E obj)`
    - `void add(int index, E obj)`
    - `E set(int index, E obj)`
  - Note that we can have two methods with the same name that take different *parameters*
    - This is called *overloading* the method
Using objects

- **Accessor methods (aka getter methods)**
  - Used to *access the data* inside of an object
    - Also known as “getting values” from an object
  - Some accessor methods include:
    - ArrayList
      - E get(int index)
      - int size()
    - String
      - char charAt(int index)
      - int length()
      - String substring(int beginindex)
      - String substring(int beginindex, int endidx)
Your Turn

For the code on the following slide, identify:

- Class names
- Constructor methods
- Public methods
  - Provide a guess to whether they are accessors or mutators
  - HINT: Use their names and what they look like they’re doing to figure out what kind of method they are

- You’ll be seeing some things you’ve never seen before
  - Based on what we’ve seen so far you should be able to figure out what’s going on here.
```java
import java.util.*;
import java.io.*;

public class ObjectExample01 {

    public static void main(String[] args) {
        Map<String, Double> myMap = new TreeMap<String, Double>();
        File fileHandle = new File("datafile.txt");
        try {
            Scanner inFile = new Scanner(fileHandle);
            while (inFile.hasNext()) {
                String key = inFile.nextLine();
                String value = inFile.nextLine();
                Double val = new Double(value);
                myMap.put(key, val);
            }
            inFile.close();
        }
        catch(IOException e) { System.out.println("ERROR: "+e); }
        for (String k: myMap.keySet()) {
            double val = myMap.get(k);
            System.out.println(k + " -> " + val);
        }
    }
}
```
Primitive Types

- You should note that not every variable in Java is an *object*
  - *Primitive types* are not objects
  - What are the primitive types?
    - `int`
    - `char`
    - `boolean`
    - `double`
    - There are others (long, float, etc.) but for our purposes this list is what we’ve been working with.
Primitive Types

- In Java, primitive types only hold data
  - They do not have associated methods

- How do we deal with shared methods then?
  - External libraries – we’ve seen some of these in our labs:
    - Character.isDigit(c)
    - Math.sqrt(d)
    - User-defined methods (see your own projects)
  - These libraries hold methods that are not associated with a particular object
    - Instead of operating on the data that is held in the object, we pass a parameter to them to operate on
    - Note that you can pass objects as parameters – not all methods that use objects need to be object-oriented!
Reference types

- Variables with a class type hold a reference to the memory location of an object of that type
  - The value of the variable is just the memory address where the object is stored
  - With a primitive type, the value of the variable is the actual value
    - This leads to some non-intuitive behavior for variables with reference types
Reference types vs. Primitive Types

What do you expect this segment of code to do?

```java
int i = 12;
int j = i;
j = j + 1;
System.out.println(i);
System.out.println(j);
```
What do you expect this segment of code to do?

```java
int i = 12;
int j = i;
j = j + 1;
System.out.println(i);
    // reports 12
System.out.println(j);
    // reports 13
```
Reference types

- Now what do you expect this segment of code to do?

```java
ArrayList<Integer> l1 = new ArrayList<>();
l1.add(3);
ArrayList<Integer> l2 = l1;
l2.add(2);
System.out.println(l1.size());
System.out.println(l2.size());
```

36
Reference types

Now what do you expect this segment of code to do?

```java
ArrayList<Integer> l1 = new ArrayList<>();
l1.add(3);
ArrayList<Integer> l2 = l1;
l2.add(0, 2);
System.out.println(l1.get(0)); // reports 2
System.out.println(l2.get(0)); // reports 2
```
Reference types

Why the “strange” behavior on the ArrayList example?

- Because ArrayList is a reference type

```java
int j = i;
```
- This line copies the value stored in `i` into the variable `j`

```java
ArrayList<Integer> l2 = l1;
```
- This line does NOT make a copy of the object `l1` into the object `l2`
- This line instead copies the memory location stored in the variable `l1` into the variable `l2`
  - Both of these variables now reference (or “point to”) the same memory location
  - We call this aliasing or making an alias. Both `l1` and `l2` are now two different names for the same object.
    - Just like “Superman” and “Clark Kent” are both names for the same person – they’re both aliases to the same object.
Reference Types

ArrayList<Integer> l1 = new ArrayList<>();

Note that this triangle is what we’ll use to denote a reference type.

Constructor creates a new, empty ArrayList object. Assigns the Reference l1 to refer to it.
ArrayList<Integer> l1 = new ArrayList<>();
l1.add(3);

Add method on l1 puts the value 3 at the end of the list referred to by l1.
Reference Types

ArrayList<Integer> l1 = new ArrayList<>();
l1.add(3);
ArrayList<Integer> l2 = l1;

**NOTE:** The new variable l2 does NOT contain a copy of l1. Instead both references now refer to (point at) the same object.
Reference Types

ArrayList<Integer> l1 = new ArrayList<>();
l1.add(3);
ArrayList<Integer> l2 = l1;
l2.add(0, 2);

The value 2 is added to the end of the object referred to by the variable l2. Note that this is the same object that l1 points to, and so the change will be reflected through both variables.
Reference Types

ArrayList<Integer> l1 = new ArrayList<>();
l1.add(3);
ArrayList<Integer> l2 = l1;
l2.add(0,2);
Reference Types

- Upshot: Be careful when dealing with *reference types*
  - Variables hold memory locations – programs may not run as intended
  - Example: equality tests (==)
    - If l1 and l2 are both of type ArrayList, what is this boolean test really doing?
      \[(l1 == l2)\]
    - What if l1 and l2 were of type String?
      - This is why we can’t use == to test for String equality in our programs! Strings are a *reference type*.
        - We need to use .equals() for Strings
        - We do the same to test equality of other objects (including ArrayLists)
Classes vs. Objects - Summary

- A class is a definition of a data type
  - Similar to a blueprint for a house
- An object is the data value instantiated (i.e. created) from a class
  - Similar to a physical house built from a blueprint
- In your code, you can declare multiple objects that use the same class
  - Think of how you can build multiple houses using the same blueprint
  - Each object has its own copies of the data values and methods defined by the class
Why Object-Oriented Programming?

Why would we want to use an object-oriented programming framework?

- **Code re-use**
  - OOP makes it very easy to re-use code from project to project.
  - With a focus on making everything a class, and breaking models up into data and behavior, code reuse comes naturally

- **Testing**
  - OOP makes it much easier to test code to make sure that it works properly
  - Test behaviors of individual objects in isolation, rather than trying to figure out how to test all of the code at once.
Why Object-Oriented Programming?

Why would we want to use an object-oriented programming framework?

- **Code re-use**
  - OOP makes it very easy to re-use code from project to project.
  - With a focus on making everything a class, and breaking models up into data and behavior, code reuse comes naturally.

- **Testing**
  - OOP makes it much easier to test code to make sure that it works properly.
  - Test behaviors of individual objects in isolation, rather than trying to figure out how to test all of the code at once.
Why Object-Oriented Programming?

- **Maintainability**
  - OOP makes it easy to find and fix errors in code
  - Error will show up as a problem with the behavior of an object, which narrows down the places to look to find why the object is mibehaving.

- **Problem Solving**
  - OOP models match up to our models of the world very well
    - Everything is an object
  - This makes coming up with solutions to problems in code somewhat easier
    - At least as a starting point.