CSE 2123
Object-oriented Programming: Objects & Classes

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”?  

- 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
Objects - Strings

- Java Strings are objects
  - Encapsulate 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'
    // What is the behavior? Methods!
    
    int x = userName.length(); // x=3
    char y = userName.charAt(0); // y='b'
    String z = userName.substring(1,2); // z = "o";
    ```
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 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 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

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
Public interface example - ArrayList

- 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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ArrayList()</code></td>
<td>Constructs an empty list with an initial capacity of ten.</td>
</tr>
<tr>
<td><code>ArrayList(Collection&lt;? extends E&gt; c)</code></td>
<td>Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.</td>
</tr>
<tr>
<td><code>ArrayList(int initialCapacity)</code></td>
<td>Constructs an empty list with the specified initial capacity.</td>
</tr>
</tbody>
</table>

**Method Summary**

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean</code></td>
<td><code>add(E e)</code></td>
<td>Appends the specified element to the end of this list.</td>
</tr>
<tr>
<td><code>void</code></td>
<td><code>add(int index, E element)</code></td>
<td>Inserts the specified element at the specified position in this list.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ArrayList()</code></td>
<td>Constructs an empty list with an initial capacity of ten.</td>
</tr>
<tr>
<td><code>ArrayList(Collection&lt;? extends E&gt; c)</code></td>
<td>Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.</td>
</tr>
<tr>
<td><code>ArrayList(int initialCapacity)</code></td>
<td>Constructs an empty list with the specified initial capacity.</td>
</tr>
</tbody>
</table>
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")) {
        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;
    }
}
```

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

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

- **How do we know which constructor will be called?**
  - Look at the *arguments* to the constructor method
    - Remember! Constructor is a method!
  - This one has no arguments, so the “no argument” 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*
    - Objects can have *private methods* too – we’ll talk more about those later
  - 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;
    }
}
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;
    }
}
```

Accessor?  Mutator?
import java.util.ArrayList;

...
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;
    }
}
```
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;
    }
}
```
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 – a “sneak peak” of some things we’ll be doing later
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);
        }
    }
}
User-defined classes

Think back to those reasons for an object-oriented approach:

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

Every class available in the Java Standard Library has been programmed by a (team of) programmers

- Every class from Scanner to ArrayList to TreeMap
- Upshot: we can also write our own classes and use them just like the ones from the Standard Library
User-defined classes

- We often write our own classes to better model things from the real world
  - Keep all of our data together
  - Keep methods (behavior) bundled with our data
- Consider a datafile full of student information
  - Student names, IDs, courses taken, grades
  - We want to read that file and make use of that information
Sample Data File

Melnitz, Janine
999999999
CSE1223, 4.0
X

Spengler, Egon
899898899
CSE1223, 4.0
CSE2123, 4.0
X

Venckman, Peter
776789898
CSE1223, 3.0
X
User-defined classes

How could we approach this problem?

- We could (CSE 1223 style) create a bunch of different arrays (or ArrayLists)
  - One for first name
  - One for last name
  - One for student ID
  - One for each course list (i.e. one array per student)
  - One for each grade list (again, one per student)

- This would quickly become a GIANT MESS
  - It takes a lot of work to keep it organized while writing the code
  - Maintaining this code is painful when changes need to be made
User-defined classes

Instead, let’s think in terms of encapsulation and objects

- Each Student will be an object with attributes:
  - First Name
  - Last Name
  - Student ID
  - List of Courses Taken
  - List of Grades for Each course

- Consider what kinds of data types each of those elements should be
User-defined Class Implementation

- This should look somewhat familiar:

```java
public class Student {
    private String firstName;
    private String lastName;
    private String id;
    private ArrayList<String> coursesTaken;
    private ArrayList<Double> courseGrades;
}
```

- Class declaration like program declaration
  - Because programs in Java are actually all implemented like classes themselves (more on this later)
Other Examples

- Consider the inventory system of a bookstore or library
  - What data would we want in a “Book” class implementation?

- Consider an on-line “shopping cart” such as Amazon.com
  - What data would we want in a “ShoppingCart” class implementation?
Writing Classes

- Instantiating an object
  - We need to call the constructor to construct a new copy of the Student object:
    ```java
    Student student1 = new Student();
    ```
  - Just like instantiating any other object
    - All objects are equal in Java – whether in the Standard Library or User-defined
  - Notice that we haven’t actually written a constructor method yet
    - All classes automatically include a “dumb” constructor known as the default constructor
Student student1 = new Student();

- Here we use the *default constructor*
  - Numeric types set to 0
  - Boolean variables set to false
  - “Reference” types set to null
    - All classes are “reference” types, as are arrays
    - Basically anything that uses the `new` keyword

- We can (and normally should) write our own replacement for the default constructor
  - In the case of our Student class, how useful is it to us to have null values for our ArrayLists?
public class Student {
    private String firstName;
    private String lastName;
    private String id;
    private ArrayList<String> coursesTaken;
    private ArrayList<Double> courseGrades;

    public Student() {
        firstName="";
        lastName="";
        id="";
        coursesTaken = new ArrayList<String>();
        courseGrades = new ArrayList<Double>();
    }
}
Writing Classes - Constructors

- Classes may have more than one constructor
  - Each constructor must have a different *signature*
- Signature – first line of a method

```java
public Student()
public Student(String firstName, String lastName)
public Student(String firstName, String lastName, String studentId)
```

- Each of these constructors can be implemented as a separate method
Using Constructors

Student student1 = new Student();
Student student2 = new Student("Bob","Marley");
Student student3 = new Student("Bob","Marley","987");

- Parameters placed within parentheses
- The object uses the right constructor method based on the signature
- Again, each signature must be unique
Using Constructors

Why have multiple constructors?

- Different situations sometimes call for different initial object values
- Imagine a system where you are taking in input about a student on-line
  - Might want a default “empty” object to read from the form
- In contrast, think of a system where you load student info from a file
  - Might want the file handle in the constructor, instead of reading and setting elements one at a time
User-defined classes

- Let’s think about other methods for Student
- What kind of methods might we want?

  computeGPA
  checkPrereqs
  readStudent
  writeStudent
public class Student {
    private String firstName;
    private String lastName;
    private String id;
    private ArrayList<String> coursesTaken;
    private ArrayList<Double> courseGrades;

    public Student() { … }
    public void readStudent(Scanner inFile) { … }
    public void writeStudent(PrintWriter outFile) { … }
    public double computeGPA() { … }
}
Public vs. Private

- **private** methods and data can only be accessed *by the object that owns them*
  - A private method can only be called from a method inside the class
  - Private data can only be read by methods inside the class

- **public** methods and data can only be accessed *by any object at all*
  - `main` is always a public method
Why restrict access?

- Helps to ensure *encapsulation* of the class
  - Make sure data can only be changed via ways the class programmer has accounted for
  - Enables class programmer to make promises to the programmers using his/her class
  - Enables the programmer to more easily debug his/her class
    - If there is only one way to set a data field, there are fewer places to check when the code is not behaving properly
Accessory and Mutator Methods

- Also known as “getter” and “setter” methods
- **public** methods to allow a programmer to manipulate the data fields in an object

For **Student**:

- setFirstName
- setLastName
- setStudentId
- addCourse
- getFirstName
- getLastName
- getStudentId
- getCourseGrade
- getCourseName
Accessor and Mutator methods

```java
public class Student {
    private String firstName;
    private String lastName;
    private String id;
    private ArrayList<String> coursesTaken;
    private ArrayList<Double> courseGrades;

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getFirstName() {
        return this.firstName;
    }

    ...}
```
this Object

- Inside the object, we have two types of variables:
  - Member variables
  - Local variables

- Member variables belong to the class
  - Can be accessed by any method

- Local variables belong to a single method
  - Have no value outside that method

- To force the use of a member variable in a method, use the keyword `this`
public class Student {
    private String firstName;
    private String lastName;
    private String[] coursesTaken;
    private double[] courseGrades;

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getFirstName() {
        return this.firstName;
    }

    ...
}

this Object
Invoking class methods

Also familiar:

```java
student1.setFirstName("Bob");
student1.setCourseGrade(1, 3.0);
String lName = student1.getLastName();
```

Methods always invoked with call:

```java
object.methodName( ... )
```
Methods

- Method signatures
  - Like constructors, methods have signatures
  - Two methods with the same name but different signatures are implemented differently:

```java
public void setCourseGrade(int grade);
public void setCourseGrade(double gradePct);
public void setCourseGrade(char letterGrade);
```
Static Methods vs. Class Methods

Note that the class methods DO NOT include the `static` keyword

- Static methods and variables are items that are tied to the whole class
  - Invoked by using the `class name` followed by the method:
    ```java
    Integer.parseInt("42");
    ```

- Non-static (or `instance methods`, or `class methods`) are tied to individual instances
  - Invoked by using the `object name` followed by the method:
    ```java
    String myString = "abc123";
    char myChar = myString.charAt(2);
    ```
Implementation Notes

- Classes should be implemented in a file with the same name as the class
  - Student implemented in “Student.java”
  - Book implemented in “Book.java”
  - etc.
Example – Student
In-class Example - Book

- Using the “Book” example, write an implementation for the Book class
  - Data
  - Constructor
  - Methods
In-class Example - Customer

- Design and implement a class to hold Customer information for an online retailer
  - Data?
  - Constructor?
  - Methods?
We’ve done a bit of “hand-waving” about the differences between *primitive types* and *reference types* (also known as *class types*).

- **Primitive types**: int, char, double, boolean, etc.
- **Reference types**: String, Scanner, ArrayList, Student

Why do we call these “reference types”? 
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

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);
```
Reference types

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

```java
Student s1 = new Student();
s1.setStudentId(“999999999”);
Student s2 = s1;
s2.setStudentId(“888888888”);
System.out.println(s1.getStudentId());
System.out.println(s2.getStudentId());
```
Reference types

- Why the “strange” behavior on the Student example?
  - Because Student is a reference type

```java
int j = i;

- This line copies the value stored in i into the variable j
```

```java
Student s2 = s1;
```

- This line does NOT make a copy of the object s1 into the object s2
- This line instead copies the memory location stored in the variable s1 into the variable s2
  - Both of these variables now reference (or “point to”) the same memory location
Reference Types

s1: MEM0x00000

```
Student:
  firstName="";
  lastName="";
```

```
public void getFirstName
...
```

```
public void setFirstName
...
```
Reference Types

Student s2 = s1;
Reference Types

Student s2 = s1;

```java
MEMORY

s1: MEM0x000000

s2: MEM0x000000

Student:
  firstName="";
  lastName="";

  public void getFirstName ...

  public void setFirstName ...
```
Reference Types

- **Upshot:** Be careful when dealing with *reference types*
  - Variables hold memory locations – programs may not run as intended
  - **Example:** equality tests (==)
    - If `s1` and `s2` are both of type `Student`, what is this boolean test really doing?
      \[(s1 == s2)\]
    - What if `s1` and `s2` were of type `String`?
      - This is why we can’t use `==` to test for String equality in our programs! Strings are a *reference type.*
Classes vs. Objects

- 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
Classes vs. Objects

public class Example

private void method3
...

private void method2
...

private void method1
...

public static void main
...

public class Student

String firstName;
String lastName;

public Student()
...

public void setFirstName
...

public String getFirstName
...
Classes vs. Objects

public class Example

private void method3
...

private void method2
...

private void method1
...

public static void main
...  
  Student sg1 = new ...

public class Student

  String firstName;
  String lastName;

  public Student()
  ...

  public void setFirstName
  ...

  public String getFirstName
  ...
Classes vs. Objects

```java
public class Example {
    private void method1 {
        ...
    }
    private void method2 {
        ...
    }
    private void method3 {
        ...
    }
    public static void main {
        ... Student sg1 = new ... 
    }
}

public class Student {
    String firstName;
    String lastName;

    public Student() {
        ...
    }
    public void setFirstName {
        ...
    }
    public String getFirstName {
        ...
    }
}
```
Classes vs. Objects

```java
public class Example
{
    private void method1
    {
    }
    private void method2
    {
    }
    private void method3
    {
    }
    public static void main
    {
        Student sg1 = new 
    }
}

public class Student
{
    String firstName;
    String lastName;
    public void setFirstName
    {
    }
    public void getFirstName
    {
    }
    public void setFirstName
    {
    }
    public Student()
    {
    }
}
```
Classes vs. Objects

```java
public class Example {
    private void method1 {
        ...
    }
    private void method2 {
        ...
    }
    private void method3 {
        ...
    }
    public static void main {
        ...
        Student sg1 = new ...
        ...
        Student sg2 = new ...
    }
}
```

```java
public class Student {
    String firstName;
    String lastName;
    public void setFirstName {
        firstName="";
    }
    public void getFirstName {
        ...
    }
    public void setFirstName {
        ...
    }
    public Student {
        ...
    }
    public Student {
        ...
    }
}
```
Classes vs. Objects

```
public class Example

private void method1
...

private void method2
...

private void method3
...

public static void main
...
Student sg1 = new ...
...
Student sg2 = new ...

public class Student

String firstName;
String lastName;


public void setFirstName
...

public void getFirstName
...

public void setFirstName
...

public Student()
...

public class Student
String firstName;
String lastName;


public Student()
...

public void setFirstName
...
```
Classes vs. Objects

public class Example

private void method1
...

private void method2
...

private void method3
...

public static void main
...

Student sg1 = new ...
...

Student sg2 = new ...

public class Student

String firstName;
String lastName;

public Student()
...

public void setFirstName
...

public void getFirstName
...

public void setFirstName
...

public void getFirstName
...

public void setFirstName
...

public void setFirstName
...

public void setFirstName
...

public void setFirstName
...