Objects and Classes
(Part 2)

Introduction to Programming and Computational Problem Solving - 2
CSE 8B
Lecture 8
Announcements

• Assignment 3 is due today, 11:59 PM
• Quiz 3 is Oct 21
• Assignment 4 will be released today
  – Due Oct 26, 11:59 PM
• Educational research study
  – Oct 21, weekly survey
• Reading
  – Liang
    • Chapter 9
Object-oriented programming

• Object-oriented programming (OOP) involves programming using objects
• This is the focus of CSE 8B
Objects and classes

• An object represents an entity in the real world that can be distinctly identified
  – For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects
  – An object has a unique identity, state, and behaviors

• Classes are constructs that define objects of the same type
Objects and Java classes

• The state of an object consists of a set of data fields (also known as properties) with their current values
• The behavior of an object is defined by a set of methods
• A Java class uses variables to define data fields and methods to define behaviors
Instance data fields and methods vs static data fields methods

• **Instance** data fields and methods **can only be accessed using an object** (i.e., an instance of a class)
  – The syntax to access an **instance data field** is `objectReferenceVariable.variableName`
  – The syntax to invoke an **instance method** is `objectReferenceVariable.methodName(arguments)`

• **Static** data fields and methods (i.e., non-instance data fields and methods) can be accessed **without using an object** (i.e., they are not tied to a specific instance of a class)
  – The syntax to access a **static data field** is `ClassName.variableName`
  – The syntax to invoke a **static method** is `ClassName.methodName(arguments)`
Instance variables vs static variables

• An instance variable belongs to a specific instance of a class

• A static variable is shared by all objects of the class
  – Static variables are shared by all the instances of the class
  – Static constants are final variables shared by all the instances of the class
Static members

• In code using a class, the best practice is to *make invocations of static methods and access of static data fields obvious*

• Use
  
  ClassName.methodName(arguments)
  ClassName.variableName

• Do **not** use
  
  objectReferenceVariable.methodName(arguments)
  objectReferenceVariable.variableName
The static modifier

- To declare static variables, constants, and methods, use the static modifier
- static is a Java keyword
The static modifier

public class Circle {
    double radius; // The radius of the circle
    static int numberOfObjects = 0; // The number of objects created

    // Construct a circle of radius 1
    Circle() {
        radius = 1;
        numberOfObjects++;  
    }

    // Construct a circle with a specified radius
    Circle(double newRadius) {
        radius = newRadius;
        numberOfObjects++;  
    }

    // Return numberOfObjects
    static int getNumberOfObjects() {
        return numberOfObjects;
    }
}
The static modifier

Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
Limitations of static methods

• An instance method can
  – Invoke an instance or static method
  – Access an instance or static data field

• A static method can
  – Invoke a static method
  – Access a static data field

• A static method cannot
  – Invoke an instance method
  – Access an instance members
Static methods

• If a member method or data field is independent of any specific instance, then make it static

• Do not require those using your class to create instance unless it is absolutely necessary
Visibility modifiers

- Visibility modifiers can be used to specify the visibility of a class and its members
- By default, the class, variable, or method can be accessed by any class in the same package
- Packages can be used to organize classes
  - For example, classes C1 and C2 are placed in package p1, and class C3 is placed in package p2

```java
package p1;
class C1 {
    public class C2 {
        public class C3 {
```
Visibility modifiers

• There is **no restriction** on accessing data fields and methods from **inside** the class

• A **visibility modifier** specifies how data fields and methods in a class can be accessed from **outside** the class
Visibility modifiers

public
  – The class, data, or method is visible to any class in any package

private
  – Modifier cannot be applied to a class, only its members
  – The data or methods can be accessed only by the declaring class

protected
  – Used in inheritance (covered in two weeks)
Packages and classes

- The **default** modifier (i.e., no modifier) on a class restricts access to **within a package**
- The **public** modifier enables **unrestricted** access

These are three different files (each class is in its own file)

```java
package p1;

class C1 {
    ...
}

package p1;

public class C2 {
    can access C1
}

package p1

public class C3 {
    cannot access C1;
    can access C2;
}
```

Compile multiple .java files in the same directory using javac *.java
Packages, classes, and members

- The **private** modifier restricts access to **within a class**
- The **default** modifier (i.e., no modifier) restricts access to **within a package**
- The **public** modifier enables **unrestricted** access

```java
package p1;

public class C1 {
    public int x;
    int y;
    private int z;

    public void m1() {
    }
    void m2() {
    }
    private void m3() {
    }
}

package p1;

public class C2 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        can access o.y;
        cannot access o.z;

        can invoke o.m1();
        can invoke o.m2();
        cannot invoke o.m3();
    }
}

package p2;

public class C3 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        cannot access o.y;
        cannot access o.z;

        can invoke o.m1();
        cannot invoke o.m2();
        cannot invoke o.m3();
    }
}
```
Visibility of own members

- There is **no restriction** on accessing data fields and methods from **inside** the class.
- However, an object cannot access its **private** members **outside** the class.

```java
public class C {
    private boolean x;

    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.x);
        System.out.println(c.convert());
    }

    private int convert() {
        return x ? 1 : -1;
    }
}
```

(a) This is okay because object `c` is used inside the class `C`.

```java
public class Test {
    public static void main(String[] args) {
        C c = new C();
        System.out.println(c.x);
        System.out.println(c.convert());
    }
}
```

(b) This is wrong because `x` and `convert` are private in class `C`.
Constructors

• Use public constructors in most cases
• Use a private constructor if you want to prohibit users from creating an instance of a class
  – For example, in java.lang.Math, the constructor Math() is private
# Methods and data fields visibility

<table>
<thead>
<tr>
<th>Modifiers on Members in a Class</th>
<th>Accessed from the Same Class</th>
<th>Accessed from the Same Package</th>
<th>Accessed from a Subclass in a Different Package</th>
<th>Accessed from a Different Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Default (no modifier)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Covered in 2 weeks
Data field encapsulation

• It is a best practice to **declare all data fields private**

• Protects data
  – From being set to an arbitrary value mistakenly (i.e., tampering) outside of the class

• Makes class easier to maintain
  – Modify the implementation inside the class without modifying all existing code currently using the class outside of the class
Accessor and mutator

- **Accessor**
  - Provide a *getter* method to read a private data field
  - Use syntax
    ```java
    public returnType getPropertyName()
    public boolean isPropertyName()
    ```

- **Mutator**
  - Provide a *setter* method to modify a private data field
  - Use syntax
    ```java
    public void setPropertyName(datatype propertyValue)
    ```
## Data encapsulation

### Circle

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-radius: double</td>
<td>The radius of this circle (default: 1.0).</td>
</tr>
<tr>
<td>-numberOfObjects: int</td>
<td>The number of circle objects created.</td>
</tr>
<tr>
<td>+Circle()</td>
<td>Constructs a default circle object.</td>
</tr>
<tr>
<td>+Circle(radius: double)</td>
<td>Constructs a circle object with the specified radius.</td>
</tr>
<tr>
<td>+getRadius(): double</td>
<td>Returns the radius of this circle.</td>
</tr>
<tr>
<td>+setRadius(radius: double): void</td>
<td>Sets a new radius for this circle.</td>
</tr>
<tr>
<td>+getNumberOfObjects(): int</td>
<td>Returns the number of circle objects created.</td>
</tr>
<tr>
<td>+getArea(): double</td>
<td>Returns the area of this circle.</td>
</tr>
</tbody>
</table>

The - sign indicates private modifier.
Pass by value

• Remember, Java uses **pass by value** to pass arguments to a method

• For a parameter of a **primitive type**, the **actual value** is passed
  – Changing the value of the local parameter inside the method **does not affect** the value of the variable outside the method

• For a parameter of an **array or object type**, the **reference value** is passed
  – Any changes to the array that occur inside the method body **will affect** the original array or object that was passed as the argument
public static void main(String[] args) {
    Circle myCircle = new Circle(1);
    int n = 5;
    printAreas(myCircle, n);
}

public static void printAreas(Circle c, int times) {
    System.out.println("Radius \t Area");
    while (times >= 1) {
        System.out.println(c.getRadius() + "\t\t" + c.getArea());
        c.setRadius(c.getRadius() + 1);
        times--;
    }
}
Arrays of objects

• An array can hold objects as well as primitive type values
• An array of objects is actually an array of reference variables
Arrays of objects

• Create an array **and** each object in it
• When creating an array using **new**, each element in the array is a reference variable with a default value of **null**

```java
Circle[] circleArray = new Circle[10];
for (int i = 0; i < circleArray.length; i++)
{
    circleArray[i] = new Circle();
}
```
Arrays of objects

• Invoking `circleArray[1].getArea()` involves two levels of referencing
  `circleArray` references to the entire array
  `circleArray[1]` references to a Circle object
Immutable objects and classes

• Occasionally, it is desirable to create an object whose contents cannot be changed once the object has been created.

• Such an object is called an immutable object and its class is called an immutable class.
Immutable objects and classes

• For example, deleting the setRadius method in the Circle class would make it an immutable class because radius is private and cannot be changed without a mutator (i.e., set) method.

```java
Circle

-radius: double
-numberOfObjects: int

+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double

The radius of this circle (default: 1.0).
The number of circle objects created.
Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.
```

The - sign indicates private modifier.
Immutable objects and classes

```java
public class Student {
    private int id;
    private BirthDate birthDate;

    public Student(int ssn, int year, int month, int day) {
        id = ssn;
        birthDate = new BirthDate(year, month, day);
    }

    public int getId() {
        return id;
    }

    public BirthDate getBirthDate() {
        return birthDate;
    }
}

public class BirthDate {
    private int year;
    private int month;
    private int day;

    public BirthDate(int newYear, int newMonth, int newDay) {
        year = newYear;
        month = newMonth;
        day = newDay;
    }

    public void setYear(int newYear) {
        year = newYear;
    }
}

public class Test {
    public static void main(String[] args) {
        Student student = new Student(111233333, 1970, 5, 3);
        BirthDate date = student.getBirthDate();
        date.setYear(2010); // Now the student birth year is changed!
    }
}

Warning: a class with all private data fields and without mutators is not necessarily immutable
```
Immutable class

- Requirements of an immutable class
  - All data fields must be private
  - There cannot be any mutator methods for data fields
  - No accessor methods can return a reference to a data field that is mutable
Scope of variables revisited

• The scope of **class variables** (instance and static data fields) is the entire class
  – They can be declared anywhere inside a class
    • Best practice is to **declare them at the beginning of the class**
  – They have default values

• The scope of a **local variable** starts from its declaration and continues to the end of the block that contains the variable
  – Java assigns no default value to a local variable inside a method
  – A local variable must be initialized explicitly before it can be used
Scope of variables revisited

- If a local variable has the same name as a class variable, then the local variable takes precedence (i.e., the class variable is hidden)

```java
public class F {
    private int x = 0; // Class variable
    private int y = 0;

    public F() {
    }

    public void p() {
        int x = 1; // Local variable
        System.out.println("x = " + x); // Uses local variable
        System.out.println("y = " + y);
    }
}
```
this reference

• The this keyword is the name of a reference that refers to an object itself

• One common use of the this keyword is to reference a hidden class variable

```java
public void p() {
    int x = 1; // Local variable
    System.out.println("x = " + this.x);
    System.out.println("y = " + y);
}
```

Class variable
Use this to reference data fields

• For a hidden **static variable**, use `ClassName.staticVariable`

• Best practice is to **use the data field name as the parameter name in the setter method or a constructor**

```java
public class F {
    private int i = 5;
    private static double k = 0;

    void setI(int i) {
        this.i = i;
    }

    static void setK(double k) {
        F.k = k;
    }
}
```

Suppose that `f1` and `f2` are two objects of `F`.

```java
F f1 = new F();
F f2 = new F();
```

Invoking `f1.setI(10)` is to execute `this.i = 10`, where `this` refers `f1`.

Invoking `f2.setI(45)` is to execute `this.i = 45`, where `this` refers `f2`.
public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public Circle() {
        this(1.0);
    }

    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
}

• The this keyword is the name of a reference that refers to an object itself
• We just used the this keyword is to reference a hidden class variable
• It can also be used inside a constructor to invoke another constructor of the same class

public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public Circle() {
        this(1.0);
    }

    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
}

• The this keyword is the name of a reference that refers to an object itself
• We just used the this keyword is to reference a hidden class variable
• It can also be used inside a constructor to invoke another constructor of the same class

public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public Circle() {
        this(1.0);
    }

    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
}

• The this keyword is the name of a reference that refers to an object itself
• We just used the this keyword is to reference a hidden class variable
• It can also be used inside a constructor to invoke another constructor of the same class

public class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public Circle() {
        this(1.0);
    }

    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
}
Next Lecture

• Object-oriented thinking

• Reading
  – Liang
    • Chapter 10