Lecture 20: Objects: Scope, Encapsulation, Inheritance

- CS 170, Section 000
- 5 November 2009
Lecture Plan

- Review (sort of): Arrays of Objects
- Logistics
- HW6 questions (due tomorrow)

- Basic Objects (wrap-up): Ch 9
  - Mutability, Scope, \texttt{this} keyword, Encapsulation
  - if time: StackOfIntegers example (9.8)

- Inheritance: Ch 10.1-5
  - Extending classes, \texttt{super} keyword, method \texttt{overriding}
Logistics

• Homework 6: Due Wed Nov 11th
  – Poker simulation

• Midterm 2: Tuesday, Nov 17th

➢ Review Session: Monday Nov 16, 4:15-6pm, W301
Array of Objects

Circle[] circleArray = new Circle[10];

• An array of objects is actually an array of reference variables
• Invoking circleArray[1].getArea() involves two levels of referencing
• circleArray references to the entire array
• circleArray[1] references to a Circle object.
Array of Objects, cont.

```java
Circle[] circleArray = new Circle[10];
```

![Diagram showing an array of Circle objects]
Array of Objects, cont.

- Compute total area covered by all the circles
- TotalArea.java
Practice problems

• Basic concept of objects and classes
  – 7.5
• Static variables and methods
  – 7.10, 7.12
• Visibility modifiers, accessor and mutator methods
  – 7.15
• Using Objects in Arrays and methods
  – 7.17, 7.20
Lecture Plan

- **Basic Objects (wrap-up): Ch 9**
  - Mutability, Scope, **this** keyword, Encapsulation
  - if time: StackOfIntegers example (9.8)

- **Inheritance: Ch 10.1-5**
  - Extending classes, super keyword, method overriding
Visibility Modifiers and Accessor/Mutator Methods

- By default, the class, variable, or method can be accessed by any class in the same package.
  - `public`
    The class, data, or method is visible to any class in any package.
  - `private`
    The data or methods can be accessed only by the declaring class.
Example of Data Field Encapsulation

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>-radius: double</td>
</tr>
<tr>
<td>-numberOfObjects: int</td>
</tr>
</tbody>
</table>

The radius of this circle (default: 1.0).
The number of circle objects created.

-Construits a default circle object.

-Construits 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.

**Circle3.java**

**TestCircle3.java**
Immutable Classes

class Student{
    private StringBuffer privateName;
    Student(String n){ privateName = new StringBuffer(n); }
    public StringBuffer getName(){return privateName};
    //no “setName” method!
}

- Trick question: can privateName still be changed by caller method (e.g., in main):

  main(...){
    Student s = new Student (“Bob”);
    System.out.println(s.getName);
    //... change Bob to Jane – how???
    System.out.println(s.getName); //print “Jane”
}
Passing Objects to Methods

- Passing by value for primitive type value (the value is passed to the parameter)
- Passing by value for reference type value (the value is the reference to the object)
Returning objects from Methods

- Methods can **return** an object
  
  ```java
  static Circle3 makeCircle(double r) {
    Circle3 c = new Circle3(r);
    return c;
  }
  ```

- So, if returned object is mutable and reference is returned, data field **can be changed**
Immutable Objects and Classes

If the contents of an object cannot be changed once the object is created, the object is called an *immutable object* and its class is called an *immutable class*. If you delete the set method in the `Circle` class in the preceding example, the class would be immutable because radius is private and cannot be changed without a set method.

A class with all private data fields and without mutators is not necessarily immutable. For example, the following class `Student` has all private data fields and no mutators, but it is mutable.
Example

```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(111223333, 1970, 5, 3);
        BirthDate date = student.getBirthDate();
        date.setYear(2010); // Now the student birth year is changed!
    }
}
```
Scope of Variables

- The scope of instance and static variables is the entire class. They can be declared anywhere inside a class.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be initialized explicitly before it can be used.
The this Keyword

• The *this* keyword is the name of a reference that refers to an object itself. One common use of the *this* keyword is reference a class’s *hidden data fields*.

• Another common use of the *this* keyword to enable a constructor to invoke another constructor of the same class.
public class Foo {
    int i = 5;
    static double k = 0;

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

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

Suppose that f1 and f2 are two objects of Foo.
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 ThisTest {
    private int number;
    public void setNumber(int number) {
        number = number;
    }
}
The meaning of `this`

- `this` is a keyword
- `this` serves as a proxy for the object

```java
public class ThisTest {
    private int number;
    public void setNumber(int number) {
        number = number;
    }
}
```
public class ThisTest1 {
    private int number;

    ThisTest1(int n) {
        number = n;
    }

    ThisTest1() {
        this(1);
    }
}
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;
    }
}

describing the usage of the `this` keyword:
- This must be explicitly used to reference the data field `radius` of the object being constructed.
- This is used to invoke another constructor.
- Every instance variable belongs to an instance represented by this, which is normally omitted.
Class Abstraction and Encapsulation
(If time may skip)

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.

Class implementation is like a black box hidden from the clients

Class

Class Contract (Signatures of public methods and public constants)

Clients use the class through the contract of the class
**Example: The StackOfIntegers Class**

<table>
<thead>
<tr>
<th>StackOfIntegers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-elements: int[]</td>
<td>An array to store integers in the stack.</td>
</tr>
<tr>
<td>-size: int</td>
<td>The number of integers in the stack.</td>
</tr>
<tr>
<td>+StackOfIntegers()</td>
<td>Constructs an empty stack with a default capacity of 16.</td>
</tr>
<tr>
<td>+StackOfIntegers(capacity: int)</td>
<td>Constructs an empty stack with a specified capacity.</td>
</tr>
<tr>
<td>+empty(): boolean</td>
<td>Returns true if the stack is empty.</td>
</tr>
<tr>
<td>+peek(): int</td>
<td>Returns the integer at the top of the stack without removing it from the stack.</td>
</tr>
<tr>
<td>+push(value: int): int</td>
<td>Stores an integer into the top of the stack.</td>
</tr>
<tr>
<td>+pop(): int</td>
<td>Removes the integer at the top of the stack and returns it.</td>
</tr>
<tr>
<td>+getSize(): int</td>
<td>Returns the number of elements in the stack.</td>
</tr>
</tbody>
</table>

[TestStackOfIntegers]
Designing the StackOfIntegers Class

Data1 → Data2 → Data3 → Data3

Data3 ← Data2 ← Data1
Implementing **StackOfIntegers** Class

- `elements[capacity - 1]`
- `elements[size-1]`
- `size`
- `bottom`
- `top`
- `capacity`

**StackOfIntegers**
StackOfInteger (cont’d)
Lecture Plan

- **Basic Objects (wrap-up): Ch 9**
  - Mutability, Scope, **this** keyword, Encapsulation
  - if time: StackOfIntegers example (9.8)

- **Inheritance: Ch 10.1-5**
  - Extending classes, super keyword, method overriding
Motivation

- What features are common for all the shapes?
- What features are specific to:
  - Triangle?
  - Circle?
  - Rectangle
Inheritance - idea

GeometricObject

- color
- isFilled

Triangle

- sideLength

Circle

- radius

Rectangle

- width
- height
extends keyword

• Use extends keyword to tell that one class inherits from other class

```java
public class GeometricObject {
    public Color color;
    public boolean isFilled;
}
```

```java
public class Circle extends GeometricObject {
    public double radius;
}
```

▷ What are properties of instances of class Circle?
### Superclasses and Subclasses

<table>
<thead>
<tr>
<th>Superclasses</th>
<th>Subclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeometricObject</td>
<td>GeometricObject1.java</td>
</tr>
<tr>
<td>Circle</td>
<td>Circle4.java</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Rectangle1.java</td>
</tr>
</tbody>
</table>

**GeometricObject**
- color: String
- filled: boolean
- dateCreated: java.util.Date

**GeometricObject1**
- +GeometricObject(): GeometricObject
- +getColor(): String
- +setColor(color: String): void
- +isFilled(): boolean
- +setFilled(filled: boolean): void
- +getDateCreated(): java.util.Date
- +toString(): String

**Circle**
- radius: double

<table>
<thead>
<tr>
<th>Circle Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Circle()</td>
<td>Constructor</td>
</tr>
<tr>
<td>+getRadius(): double</td>
<td>Radius getter</td>
</tr>
<tr>
<td>+setRadius(radius: double): void</td>
<td>Radius setter</td>
</tr>
<tr>
<td>+getArea(): double</td>
<td>Area getter</td>
</tr>
<tr>
<td>+getPerimeter(): double</td>
<td>Perimeter getter</td>
</tr>
<tr>
<td>+getDiameter(): double</td>
<td>Diameter getter</td>
</tr>
</tbody>
</table>

**Rectangle**
- width: double
- height: double

<table>
<thead>
<tr>
<th>Rectangle Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Rectangle()</td>
<td>Constructor</td>
</tr>
<tr>
<td>+getWidth(): double</td>
<td>Width getter</td>
</tr>
<tr>
<td>+setWidth(width: double): void</td>
<td>Width setter</td>
</tr>
<tr>
<td>+getHeight(): double</td>
<td>Height getter</td>
</tr>
<tr>
<td>+setHeight(height: double): void</td>
<td>Height setter</td>
</tr>
<tr>
<td>+getArea(): double</td>
<td>Area getter</td>
</tr>
<tr>
<td>+getPerimeter(): double</td>
<td>Perimeter getter</td>
</tr>
</tbody>
</table>
Inheritance

- A subclass inherits all fields and methods from the superclass
- A subclass can also:
  - Add new fields
  - Add new methods
  - Override the methods of the superclass

- Superclass’s constructor are not inherited
  - Unlike fields and methods
- They are invoked explicitly or implicitly
Using the Keyword `super`

- `super` refers to the superclass of the class in which `super` appears
- This keyword can be used in few ways:
  - To call a superclass constructor
  - To call a superclass method
  - To access a superclass public data field
Invoking Superclass Constructor

• Superclasses’ constructors can (only) be invoked from subclasses' constructors explicitly
• Use the keyword super to call the superclass constructor
• Java requires that the statement that invokes superclass’ constructor using the keyword super appear first in the constructor
Superclass’s Constructor Is Always Invoked

• If no superclass constructor is explicitly invoked, the compiler puts `super()` as the first statement in the constructor

```java
public A() {
    // some statements
}
```

is equivalent to

```java
public A() {
    super();
    // some statements
}
```

```java
public A(double d) {
    // some statements
}
```

is equivalent to

```java
public A(double d) {
    super();
    // some statements
}
```
public class Faculty extends Employee {
    public static void main(String[] args) {
        Faculty f = new Faculty("Brad Pitt");
    }

    public Faculty(String name) {
        System.out.println("(3) Faculty's constructor");
    }
}

class Employee extends Person {
    public Employee() {
        System.out.println("(2) Employee's no-arg constructor");
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor");
    }
}
public class Faculty extends Employee {
    public static void main(String[] args) {
        Faculty f = new Faculty("Brad Pitt");
    }
    public Faculty(String name) {
        super();
        System.out.println("(3) Faculty's constructor");
    }
}

class Employee extends Person {
    public Employee() {
        super();
        System.out.println("(2) Employee's no-arg constructor");
    }
}

class Person {
    public Person() {
        super();
        System.out.println("(1) Person's no-arg constructor");
    }
}
Calling Superclass Methods

• super can be used to call method from superclass

```java
public void printCircle() {
    System.out.println("The circle is created " +
        super.getDateCreated() + " and the radius is " +
        radius);
}
```

- super can be omitted sometimes… when?

```java
public void printCircle() {
    System.out.println("The circle is created " +
        getDateCreated() + " and the radius is " +
        radius);
}
```
Overriding Methods in the Superclass

• A subclass inherites methods from a superclass
• Subclass can modify the implementation of a method defined in the superclass.

• Method overriding.

```java
public class Circle extends GeometricObject {
    // Other methods are omitted

    /** Override the toString method defined in GeometricObject */
    public String toString() {
        return super.toString() + 
                "\nradius is " + radius;
    }
}
```
NOTE

• An instance method can be overridden only if it is accessible
• A private method cannot be overridden, because it is not accessible outside its own class
Overriding vs. Overloading

```java
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
    }
}

class B {
    public void p(int i) {
    }
}

class A extends B {
    // This method overrides the method in B
    public void p(int i) {
        System.out.println(i);
    }
}
```

```java
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
    }
}

class B {
    public void p(int i) {
    }
}

class A extends B {
    // This method overloads the method in B
    public void p(double i) {
        System.out.println(i);
    }
}
```
Another Example: Person, Student

- Student extends Person

- Code: inclass/nov10/student/
Review questions

• Which of the following statements are true?

A. A subclass is a subset of a superclass.
B. A subclass is usually extended to contain more functions and more detailed information than its superclass.
C. "class A extends B" means A is a subclass of B.
D. "class A extends B" means B is a subclass of A.
Review questions

• Which of the following statements are true?

A. A method can be overloaded in the same class.
B. A method can be overridden in the same class.
C. If a method overloads another method, these two methods must have the same signature.
D. If a method overrides another method, these two methods must have the same signature.
Object: The Cosmic Superclass

- All classes defined without an explicit `extends` clause automatically extend `Object`
Object: The Cosmic Superclass

• Most useful methods:
  – String toString()
  – boolean equals(Object otherObject)

• Good idea to override these methods in your classes
The toString() method in Object

- The toString() method returns a string representation of the object.
- The method is called whenever the object is converted to a string.

```
System.out.println("Bankaccount: " + myaccount);
```

- The default implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object.

```
BankAccount account = new BankAccount();
System.out.println(account); //BankAccount@15037e5
```
Overriding the `toString` Method

- To provide a nicer representation of an object, override `toString()`.
  ```java
  public String toString()
  {
      return "BankAccount [balance=" + balance + "]";
  }
  ```

- This works better:
  ```java
  BankAccount momsSavings = new BankAccount(5000);
  System.out.println(momsSavings);
  //BankAccount [balance=5000]
  ```

- More examples: Card.java
The equals Method

- The equals() method compares two objects.
- The default implementation of the equals method in the Object class is as follows:

```java
public boolean equals(Object obj) {
    return (this == obj);
}
```

- Java classes such as String override equals() method so that it compares the content of two objects.
- It is a good idea to override equals() method for your own classes.
public class Circle {

    private double radius;

    public Circle(double r) {
        radius = r;
    }

    public boolean equals(Object o) {
        Circle other = (Circle)o;
        return other.radius == radius;
    }
}
Summary

- **Basic Objects: Ch 9 (most of, not all)**
  - Mutability, Scope, *this* keyword, Encapsulation
  - if time: StackOfIntegers example (9.8)

- **Inheritance: Ch 10.1-5**
  - Extending classes
  - *super* keyword
  - method *overriding*
Review questions

• What is the output of the following code:

```java
public class Test {
    public static void main(String[] args) {
        Object o1 = new Object();
        Object o2 = new Object();
        System.out.println((o1 == o2) + " " + (o1.equals(o2)));
    }
}
```

A. false false
B. true true
C. false true
D. true false
Review questions

What is the output of the following code:

```java
public class Test {
    public static void main(String[] args) {
        String s1 = "Java";
        String s2 = "Java0".substring(0,4);
        System.out.print((s1 == s2) + " " + (s1.equals(s2)));
    }
}
```

A. false false
B. true true
C. false true
D. true false