Chapter 5

Object-Oriented Programming
Objectives

- Develop code that implements tight encapsulation, loose coupling, and high cohesion
- Develop code that demonstrates the use of polymorphism
- Develop code that declares and/or invokes overridden or overloaded methods and code
- Develop code that implements “is-a” and/or “has-a” relationships
Understanding Object-Oriented Relationships
Implementing Polymorphism
Conversion of Object Reference Types
Using Method Overriding and Overloading
A class and its members are related to each other: the class has data variables in it.

Classes themselves are related to each other: a class is derived from another class.

Two kinds of relationships: *is-a* and *has-a*, correspond to the two properties of classes: inheritance and data abstraction.
The *is-a* relationship corresponds to inheritance

A class is in an *is-a* relationship with any class up in its hierarchy tree

- Boombox is a subclass of Stereo.
- Stereo is a superclass of Boombox.
- Boombox inherits from Stereo.
- Boombox is derived from Stereo.
- Boombox extends Stereo.
corresponds to an object-oriented characteristic called encapsulation: the data and the methods are combined into a class

```java
class Stereo {
}

class Boombox extends Stereo {
    CDPlayer cdPlayer = new CDPlayer();
}

class CDPlayer {
}
```
Encapsulation facilitates *data abstraction*, the relationship between a class and its data members.

- Tight encapsulation: all data members of the class should be declared private
- Data abstraction (data hiding): the data is hidden from the user
- makes the code more reliable, robust, and reusable
**Listing 5-1.** TestEncapsulateBad.java

```java
1. public class TestEncapsulateBad {
2.     public static void main(String[] args) {
3.         EncapsulateBad eb = new EncapsulateBad();
4.         System.out.println("Do you have a headache? " + eb.headache);
5.     }
6. }
7. class EncapsulateBad {
8.     public boolean headache = true;
9.     public int doses = 0;
10. }
```
Listing 5-2. TestEncapsulateGood.java

1. public class TestEncapsulateGood {
2.     public static void main(String[] args) {
3.         EncapsulateGood eg = new EncapsulateGood();
4.         eg.setHeadache(false);
5.         System.out.println("Do you have a headache? " + eg.getHeadache());
6.     }
7. }
8. class EncapsulateGood {
9.     private boolean headache = true;
10.    private int doses = 0;
11.    public void setHeadache(boolean isHeadache){
12.        this.headache = isHeadache;
13.    }
14.    public boolean getHeadache(){
15.        return headache;
16.    }
17. }
Loose Coupling

- refers to minimizing the dependence of an object on other objects
- can change the implementation of a class without affecting the other classes
- make the code extensible and easy to maintain
- demands that a class keep its members private and that the other class access them through getters and setters
access the public member of another class directly

Listing 5-3. TightlyCoupledClient.java

```java
1. public class TightlyCoupledClient{
2.   public static void main(String[] args) {
3.     TightlyCoupledServer server = new TightlyCoupledServer();
4.     server.x=5;  //should use a setter method
5.     System.out.println("Value of x: " + server.x);
6.     //should use a getter method
7.   }
8. }
9. class TightlyCoupledServer {
10.   public int x = 0;  //should be private
11. }
```
refers to how a class is structured

A cohesive class is a class that performs a set of closely related tasks

If a class is performing a set of unrelated tasks (a non-cohesive class), you should consider writing multiple classes - reshuffling the tasks
Understanding Object-Oriented Relationships

Implementing Polymorphism

Conversion of Object Reference Types

Using Method Overriding and Overloading
Implementing Polymorphism

- The *is-a* relationship between a superclass and a subclass: you can substitute an object of the subclass for an object of the superclass

```java
Animal a = new Animal(); // Variable a points to an object of Animal
a = new Cow(); // Now, a points to an object of Cow.

Cow c = new Animal(); // not valid
```
A superclass can have multiple subclasses, giving different meaning to the same thing

- The capability to convert an object reference from one type to another type

Listing 5-4. TestPoly.java

1. public class TestPoly {
2.   public static void main(String [] args) {
3.     Animal heyAnimal = new Animal();
```java
4.     Cow c = new Cow();
5.     Buffalo b = new Buffalo();
6.     heyAnimal=c;
7.     heyAnimal.saySomething();
8.     heyAnimal=b;
9.     heyAnimal.saySomething();
10. }
11. }
12.class Animal {
13.     public void saySomething() {
14.         System.out.println("Umm...");
15.     }
16. }
17. class Cow extends Animal {
18.     public void saySomething() {
19. //         super.saySomething();
20.         System.out.println("Moo");
21.     }
22. }
23. class Buffalo extends Animal{
24.     public void saySomething() {
25. //         super.saySomething();
26.         System.out.println("Bah");
27.     }
28. }
```
Contents

- Understanding Object-Oriented Relationships
- Implementing Polymorphism
- Conversion of Object Reference Types
- Using Method Overriding and Overloading
Implicit Conversion of Object Reference Types:
- Assignment Conversion
- Method Call Conversion

Explicit Conversion of Object Reference Types - object reference casting
## Assignment Conversion

Table 5-2. *Rules for Implicit Conversion of Object Reference Types*

<table>
<thead>
<tr>
<th>Target Type</th>
<th>Source Type</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Class</td>
<td>Target type must be an Object.</td>
</tr>
<tr>
<td></td>
<td>Source type must be a subclass of target type.</td>
<td>Target type must be an Object.</td>
</tr>
<tr>
<td>Interface</td>
<td>Source type must implement the interface of target type.</td>
<td>Source type must be subinterface of target type.</td>
</tr>
<tr>
<td>Array</td>
<td>Compiler error.</td>
<td>Compiler error.</td>
</tr>
</tbody>
</table>
Figure 5-3. *Hierarchy of classes to illustrate the conversion rules*
Method Call Conversion

- The rules are the same as the rules for assignment conversion.
- The passed-in argument types are converted into the method parameter types when the conversion is valid.
Explicit Conversion of Object Reference Types

- Because object reference conversion may involve reference types and object types:
  - Some of the casting rules relate to the reference type (known at compile time) and therefore can be enforced at compile time.
  - Some of the casting rules relate to the object type (not known at compile time) and therefore can only be enforced at runtime.
Rules at Compile Time

- Classes: one class must be a subclass of the other.
- Arrays: the elements of both the arrays must be object reference types. Also, the object reference type of the source array must be convertible to the object reference type of the target array.
- The casting between an interface and an object that is not final is always allowed.
Rules at Runtime

- If the target type is a class, then the class of the expression being converted must be either the same as the target type or its subclass.
- If the target type is an interface, then the class of the expression being converted must implement the target type.
1. Auditorium a1, a2;
2. ClassRoom c;
3. LectureHall lh;
4. a1 = new Auditorium();
5. c = a1;  // legal implicit conversion.
6. a2 = (Auditorium) c;  // Legal cast.
7. lh = (LectureHall) c;  // Illegal cast.
Contents

- Understanding Object-Oriented Relationships
- Implementing Polymorphism
- Conversion of Object Reference Types
- Using Method Overriding and Overloading
Overriding allows you to modify the behavior of an inherited method to meet the specific needs of a subclass: extensibility.

Overloading allows you to use the same method name to implement different (but related) functionalities: flexibility.
to change the behavior of an inherited method

redefine the method by keeping the same signature but rewriting the body

The rules for overriding a method:

- You cannot override a method that has the final modifier.
- You cannot override a static method to make it non-static.
- The overriding method and the overridden method must have the same return type
The number of parameters and their types: same
You cannot override a method to make it less accessible
If the overriding method has a throws clause:
  • The overridden method must have a throws clause
  • Each exception included in the throws clause of the overriding method must be either one of the exceptions in the throws clause of the overridden method or a subclass of it.
protected int aMethod(String st, int i, double number);

<table>
<thead>
<tr>
<th>Method Signature in a Subclass</th>
<th>Validity</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected int  aMethod(String st, int i, double number)</td>
<td>Valid</td>
<td>Same signature</td>
</tr>
<tr>
<td>protected int  aMethod(String st, int j, double num)</td>
<td>Valid</td>
<td>Same signature</td>
</tr>
<tr>
<td>protected double  aMethod(String st, int i, double number)</td>
<td>Invalid</td>
<td>Different return type</td>
</tr>
<tr>
<td>protected int  aMethod(int i, String st, double number)</td>
<td>Invalid</td>
<td>Argument types are in different order</td>
</tr>
<tr>
<td>protected int  aMethod(String st, int i, double number, int j)</td>
<td>Invalid</td>
<td>Different number of types</td>
</tr>
<tr>
<td>protected int  aMethod(String st, int i)</td>
<td>Invalid</td>
<td>Different number of types</td>
</tr>
<tr>
<td>int  aMethod(String st, int i, double number)</td>
<td>Invalid</td>
<td>Default modifier is less public than protected</td>
</tr>
</tbody>
</table>
The same task is to be performed in slightly different ways under different conditions

- Multiple methods in a class with identical names
- No two overloaded methods could have the same parameter types in the same order
- The return types in overloaded methods may be the same or different
Listing 5-9. TestAreaCalculator.java

1. class TestAreaCalculator {
2.     public static void main(String[] args) {
3.         AreaCalculator ac = new AreaCalculator();
4.         System.out.println("Area of a rectangle with length 2.0, and width 3.0: "+ ac.calculateArea(2.0f, 3.0f));
5.         System.out.println("Area of a triangle with sides 2.0, 3.0, and 4.0: "+ ac.calculateArea(2.0, 3.0, 4.0));
6.         System.out.println("Area of a circle with radius 2.0: "+ ac.calculateArea(2.0));
7.     }
8. }
9. class AreaCalculator {
10.    float calculateArea(float length, float width) {
11.        return length*width;
12.    }
13.    double calculateArea(double radius) {
14.        return ((Math.PI)*radius*radius);
15.    }
16.    double calculateArea(double a, double b, double c) {
17.        double s = (a+b+c)/2.0;
18.        return Math.sqrt(s*(s-a)*(s-b)*(s-c));
19.    }
20. }
Listing 5-10. ConstOverload.java

```java
1. public class ConstOverload{
2.   public static void main(String[] args) {
3.       new A();
4.   }
5. }
6. class A {
7.   int x=0;
8.   A(){
9.     this(5);
10.    System.out.println("A() ");
11. }
11. A(int i){
12.    // this();
13.    System.out.println(i);
14. }
15. }
```