Inheritance and Interfaces

Chapter 9

Inheritance

Generalization, gen-spec, is-a, is-a-kind-of, is-like-a, similar

Least Derived

Person

Base Class

Superclass

Parent Class

Generalization

Ancestor

Most Derived

Employee

Derived Class

Subclass

Child Class

Specialization

Descendant

Contains all members of the base class, which is called a "subobject".

Inheritance Concepts

A fundamental dimension of the object-oriented model

- Relationship between a class and a more refined version
- Abstraction for sharing similarities among classes while preserving their differences
  - Mechanism for code reuse
  - Conceptual simplification by reducing the number of unique features
- Each subclass inherits all data & methods of the superclass
- An instance of a subclass is simultaneously an instance of all its ancestor classes
- The second defining feature of the object model
  - Object-Based systems support objects but not inheritance

Inheritance and extends

Java inheritance notation

public class Shape ()

public class Circle extends Shape ()

public class Square extends Shape ()

public class Triangle extends Shape ()

- Circle, Square, and Triangle are all derived from Shape
- Java uses the keyword extends to denote inheritance

Aggregation

Whole-part, "has a," assembly, or "a part of"

- Form of association
  - Parts exist independently of whole
  - Parts and whole have independent lifetimes
  - Parts may belong to multiple wholes
  - Parts may change during execution
  - Antisymmetric (i.e., one way)
    - If A is part of B, B cannot be part of A
    - Motor and Transmission do not reference Automobile

- Example

  public class Automobile
  {
    private Motor engine;
    private Transmission driveTrain;
  }

Inheritance, Constructors, & super

Referring to an object’s parent

- Subclass constructors call super class constructors
  - Automatically (and transparently) calls the default superclass constructor
  - The super method calls the appropriate superclass constructor when the default constructor is inappropriate or doesn’t exist
  - First statement in the superclass constructor
  - The signature determines the correct constructor call
    - super(x, y);
- Subclasses may override superclass methods
  - Same signature (name, argument list, and return value type)
  - Hides superclass method
  - Superclass method is accessed with the super object:
    super.overriddenMethodName();
Methods With The Same Name
Overloading vs overriding
- Overloading methods (defining methods with the same name)
  - Defined in the same class
  - Must have different signatures (i.e., different argument lists)
  - May have different return types but cannot overload on return type
- Overriding methods (defining methods with the same name)
  - Requires two or more classes related through inheritance
  - One method defined in a base class, another in a derived class
  - Have the same signatures and the same return type
  - Subclass method cannot be more restrictive than superclass method
  - Subclass method cannot throw more checked exceptions than superclass method
- Methods with the same name and the same signature but with different return types are not allowed

Overloading/Overriding Illustrated
Reusing method names
- The draw method in the Circle class 
- The move method is overloaded in the Circle class

Inheritance and Aggregation
"is-a" and "has-a"

```
public class Person
{
    private String name;
    private Address addr;  // has-a
    public Person(String name, String street, String city)
    {
        this.name = name;
        addr = new Address(street, city);
    }
    public String toString()
    {
        return name + " " + addr.toString();
    }
}
```

```
public class Student extends Person  // is-a
{
    private double gpa;
    public Student(String name, String city, String street, double gpa)
    {
        super(name, city, street);
        this.gpa = gpa;
    }
    public String toString()
    {
        return super.toString() + " " + gpa;
    }
}
```

super and this Example
Two uses for each keyword

```
public class Bar
{
    protected int data;
    public Bar(int data)
    {
        this.data = data;
    }
    public Bar()
    {
        this(100);
    }
    public void printData()
    {
        System.out.println(data);
    }
    // class Bar
}
```

```
public class Foo extends Bar
{
    private int data;
    public Foo(int data)
    {
        super(data);
        this.data = data-50;
        super.data = 200;
    }
    public Foo()
    {
        this(10); // this(0);
    }
    public void printData()
    {
        super.printData();
        super.printData(0);
    }
    // class Foo
}
```

super Example
Alternative constructors: private vs protected

```
public class Shape
{
    private int color;
    Shape(int C) { color = C; }
    void draw() { ... }
}
```

```
public class Circle extends Shape
{
    int radius;
    public Circle(int r, int C)
    {
        super(C);
        radius = r;
    }
    public void draw()
    {
        super.draw();
    }
}
```

Object: The Mother of All Classes
The ultimate ancestor
- All classes are derived from Object
  - Explicitly extending Object or by recursively extending a class that extends Object
  - Implicitly by not extending any class
- Common services (i.e., methods inherited from Object)
  - String toString()
  - boolean equals(Object) class name and object information true if they represent the same address
  - Object clone() duplicates with simple bitwise copy
  - Class getClass() object with data about object
Abstract Classes

Classes that cannot be instantiated

- Concrete classes may be instantiated
- Abstract classes cannot be instantiated
- They only make sense in the context of a generalization
  - Organize features common to many classes
  - Declare an operation (protocol, interface) that each subclass must provide
  - The origin class is the topmost defining class; it defines the protocol
- Abstract operations must be overridden in concrete classes
- Some abstract classes appear naturally in the problem domain; others are abstractions artificially introduced for code reuse or from implementation requirements

abstract Classes and Methods

Must be extended or overridden

- abstract classes cannot be instantiated
- abstract methods must be overridden in subclasses
  - Have no body (i.e., no code) in super class (i.e., the abstract class)
  - Cannot be static or private
  - Describes a common protocol for all derived classes
- abstract classes can be superclasses
  - Classes with one or more abstract methods must be abstract
  - Contain data common to all subclasses
  - Maintain concrete methods common to all subclasses
  - Subclasses must override all abstract methods
  - Provide common ancestor for casting and polymorphism

Inheritance and Casting

Converting from a base class object to a derived class object

- Casting is the conversion of one object to another and is only possible within an inheritance hierarchy
- Up-casting (subclass to superclass)
  - Does not require an explicit cast
  - Provides a generic data type
  - Substitutability (p. 527)
- Down-casting (from a superclass to a subclass) requires an explicit cast
  - Should be avoided when possible

Shape

Circle

Polymorphism

Many shapes: late, run-time, or dynamic binding; also dynamic dispatch

- Selection of the correct method is deferred until run-time when the selection is based on the current object
- Objects respond differently to the same message
- Java methods are polymorphic by default
- Requires inheritance (uses upcasting & method overriding)
  - Variable is of superclass type; superclass may be abstract
  - Reference object is of subclass type
  - Polymorphic methods “expect to be overridden”
  - Non-polymorphic methods are unaffected
  - Called method is from the subclass
- Third defining feature of the object-oriented model
Polymorphism Example

Dynamic binding

```
Shape S => S.draw(); // polymorphism example
```

Which Method Is Called

Polymorphism vs non-polymorphism

- **Shape** \( \text{S = new Circle();} \)
- \text{void render(Shape S);} \)
  - \text{render(new Shape());}
  - \text{render(new Circle());}

- **Non-polymorphic call (default in C++)**
  - Method/function belongs to the class named on the left hand side of the assignment operator
- **Polymorphic call (default in Java™)**
  - Method/function belongs to the instantiated class on the right hand side of the assignment operator

Example

- Non-polymorphic: \text{S.draw();} // Shape draw method
- Polymorphic: \text{S.draw();} // Circle draw method

Implementing Interfaces

Using interfaces

- **An interface is a contract**
  - Compiler verifies that the implementing class overrides all interface methods (it is a compile time error if it doesn’t)
- **An interface is a data type**
  - Variables point to objects instantiated from implementing classes

Example

```java
public class Bar implements ActionListener 
{ 
  ActionListener foo = new Bar();
  public void actionPerformed(ActionEvent event) 
  { 
    
  }
}
```
interface vs Abstract Class

Comparing similar constructs

- **Similarities**
  - Specify *abstract* methods, which must be overridden elsewhere
  - Specify constants (data that is *public, static and final*)
  - Can be used as a generic type specifier that can reference any object instantiated from a class that implements that interface, which is useful in polymorphism
  - Can be the right hand operand of instanceof
  - Cannot instantiate either an abstract class or an interface

- **Differences**
  - Interfaces do not specify concrete methods
  - Interfaces do not specify instance variables
  - Interfaces do not contain anything that would form a subobject

---

Interface Summary

Key concepts

- **Interface**
  - Methods are abstract
  - The abstract keyword may be used but is superfluous (i.e., not required)
  - They do not have bodies
  - Data are public, static, final
  - The keywords may be used but are superfluous (i.e., not required)
  - They are constant and must be initialized

- **public interface name and file name must agree**
  - Non-public interfaces should also follow this naming convention
  - public interfaces can be implemented outside of the package
  - friendly interfaces can only be implemented within the package

- **A class can implement multiple interfaces**
  - State implements once
  - Specify the interfaces as a comma separated list of interface names