Inheritance

Chapter 5

Copyright © 1998-2002 Delroy A. Brinkerhoff. All Rights Reserved.

Inheritance: Assorted Terminology

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

Least Derived Class

Base Class

Shape

Superclass

Parent Class

Generalization

Ancestor

UML inheritance diagram

Derived Class

Circle

Subclass

Child Class

Specialization

Descendant

Shape Object

This is a "subobject" embedded in the Circle object.

Circle component. Contains members unique to Circle.

Circle Object

Inheritance Concepts & Terminology

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
- Overriding—subclass defines a method with the same name

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

Shape

Square

Triangle

draw

draw

draw

draw

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, an other in a derived class
  - Have the same signature and the same return type
  - Subclass method cannot be more restrictive 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 overrides or hides the draw method in the Shape class
- In Java™, overridden methods are automatically polymorphic; in C++, virtual methods are polymorphic
- The move method is overloaded in the Circle class
Inheritance, Constructors, & super

Referring to an object’s parent

- Instantiating a subclass instantiates superclasses subobjects
  - Subobjects instantiated from superclasses must be initialized
  - The subclass constructor 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

- Subclasses may override superclass methods
  - Same signature (name, argument list, and return value type)
  - Hides superclass method

  Subclass method is accessed with the super object:

  super.overridenMethodName();

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);
  }
  public void printData()
  {
    super.printData();
    System.out.println(data);
  }
}
// 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();
  }
}

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 Class Example

How do you draw a “Shape?”

- Abstract class

  - Concrete classes
    - Inherit Shape position and color
    - Inherit Shape move
    - Must override Shape draw and erase or become abstract

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 and an abstract class must have at least one abstract method
  - 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

Chapter 5 Slide 11 of 34

Chapter 5 Slide 12 of 34
Abstraction Example

Java™ abstract classes and methods

- An abstract class has at least one abstract method
- A class with one or more abstract methods must be abstract
- An abstract class can have concrete methods

```java
abstract public class Shape {
    private int color;
    public Shape(int c) {
        color = c;
    }
    public abstract void draw();
    public void setColor(int C) {
        color = C;
    }
}
```

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

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 (from a subclass to a superclass) does not require an explicit cast
- Provides a generic data type (a variable of type Shape can reference a Circle, Square, etc. object)
- Down-casting (from a superclass to a subclass) requires an explicit cast
- Should be avoided when possible

```java
Shape C = new Circle(); // up-cast
Circle C0 = (Circle) C; // down-cast
```

Associating Features with Classes

Symbol table example

- Shape S;
  - Compiler finds Shape in it’s symbol table
  - Able to locate features in Shape object
- Circle C;
  - Compiler finds Circle in symbol table
  - Able to locate features in Circle object
- Able to locate features in Shape subobject
- Shape references only “know” about Shape features
- Circle references “know” about both Shape and Circle features

Up vs Down Casting

Accessing instance variables

- Shape C = new Circle();
- Circle S = new Shape(); // NOT Safe

Safe Down Casting with instanceof

Look before casting

- `instanceof` is a binary operator
  - Left hand operand: an object
  - Right hand operand: a class
  - Returns boolean
- Example
  ```java
  Shape S = new Circle(...); // upcast
  if (S instanceof Circle)
      Circle C = (Circle) S;
  else if (S instanceof Triangle)
      Triangle T = (Triangle)S;
  ```

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

Exact shape selected dynamically at runtime, perhaps in response to user input.

Which draw method is called? Cannot determine at compile time—selection deferred until runtime.

Java™ Program Example

Simple polymorphism example

class Poly // polymorphism example
{
  public static void main(String args[])
  {
    Shape[] graphics = new Shape[4]; // instantiate array
    graphics[0] = new Shape(); // instantiate objects; fill array
    graphics[1] = new Circle(); // up-cast
    graphics[2] = new Square(); // up-cast
    graphics[3] = new Triangle(); // up-cast
    for (int i = 0; i < 4; ++i)
      graphics[i].draw(); // polymorphic call to Draw
  }
} // class Poly

Determining Which Method Is Called

Polymorphism vs non-polymorphism

- Shape S = new Circle();
- void render(Shape S);
  <render(new Shape());
  <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: S.draw() // Shape draw method
  <Polymorphic: S.draw() // Circle draw method

Java™ Virtual Machine

Simulates a machine with a simple architecture

Polymorphism Example

Java™

public class Shape
{
  private int color;
  public Shape(int c)
  {
    color = c;
  }
  void draw()
  {
  }
  void setColor(int C)
  {
    color = C;
  }
}

public class Circle extends Shape
{
  private int radius;
  public Circle(int r, int c)
  {
    super(c);
    radius = r;
  }
  void draw()
  {
  }
  void draw()
  {
  }
}

Shape S = new Shape(blue);
Circle C = new Circle(25, yellow);
Shape CP = new Circle(5, pink);

S.draw(); // Shape draw
C.draw(); // Circle draw
CP.draw(); // Circle draw

Polymorphism Revealed

How Java™ implements polymorphism

- C points to a "handle"
- The handle points
  - Circle object (data)
  - Circle method area
- S points at the same handle as C
Preventing Inheritance: `final`

Switching off polymorphism

- Applying the `final` modifier to a method prevents derived classes from overriding that method
- Applying the `final` modifier to a class prevents it from being a base class (i.e., a `final` class cannot be extended)
- All methods in a `final` class are automatically `final`
- There are two reasons to make classes and methods `final`
  - **Efficiency**
    - Dynamic binding requires a table look-up and an indirect function call
    - Dynamic binding precludes expanding small functions inline
  - **Safety**
    - You have no control over the action of an overridden method

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( )` class name and object information
  - `boolean equals(Object)` true if they represent the same address
  - `Object clone( )` duplicates with simple bitwise copy
  - `Class getClass( )` `Class` object with data about object
- Reduces the need for multiple inheritance
  - Would also make multiple inheritance difficult to implement
  - Container classes can hold `Object` values
- Polymorphic method calls access the correct behaviors

Test For Equality Example

By default, objects are equal if they have the same address

```java
public class eqDemo {
    public static void main(String[] args) {
        Demo D1 = new Demo(100);
        Demo D2 = new Demo(200);
        Demo D3 = D1;
        if (D1 == D2) System.out.println("D1 == D2");
        if (D1 == D3) System.out.println("D1 == D3");
        if (D2 == D3) System.out.println("D2 == D3");
        if (D1.equals(D2)) System.out.println("D1.equals(D2)\n        if (D1.equals(D3)) System.out.println("D1.equals(D3)\n        if (D2.equals(D3)) System.out.println("D2.equals(D3)\n    }
}
```

Generic Stack

Can hold any object but cannot hold simple types: upcasting at work

```java
public class Stack {
    private int stackSize;
    private int stackpointer = 0;
    private Object stack[];
    public Stack(int size) {
        stackSize = size;
        stack = new int[stackSize];
    }
    public void push(Object data) {
        if (stackpointer < stackSize) {
            stack[stackpointer++] = data;
        }
    }
    public Object pop() {
        if (stackpointer > 0) {
            return stack[--stackpointer];
        }
    }
    public int size() {
        return stackpointer;
    }
    // Stack ...
}
```

Object Wrappers

Classes for simple data types

- Number, Integer, Long, Float, Double, Byte, Character, Void, and Boolean
- Common services
  - Type conversions
    - `String toString( )`
    - `int i = Integer.parseInt(S);` // new with 1.1
    - `double d = Double.parseDouble(S);` // new with 1.2
  - Common Numeric Constants
    - `MAX_VALUE`
    - `MIN_VALUE`
- Can be stored in container objects (simple types cannot)

The Vector Class

Container class `java.util.Vector`

- Can hold any number of any object
  - Dynamically resizes
  - Stores items as `Object` type (i.e., upcasts)
- Methods
  - `<vector( [ int capacity ] )`
  - `<add( [ int index ] , Object item ) or addElement( Object item )`
  - `<clear()`
  - `<size( )`
  - `<elementAt( int index ) // 0 .. size( ) - 1`
  - `<indexOf( Object item , [ int index ] ) // should be called search`
  - `<insertElementAt( Object item , int index )`
  - `<isEmpty( )`
  - `<remove( Object item )`
  - `remove( int index )`
Vector Example

Inserting and retrieving objects

```java
Vector shapes = new Vector();
shapes.addElement(new Circle(...));
shapes.addElement(new Square(...));
shapes.addElement(new Triangle(...));
for (int i = 0; i < shapes.size(); i++)
{
    Shape S = (Shape)shapes.elementAt(i);
    S.draw();
}
```

Runtime Type Identification

RTTI

- `class Object`
  - `getClass()` // returns a `class` object
  - `toString()`
- `class Class extends Object`
  - `A class object contains information about the class from which an object was instantiated`
  - `getName()`
  - `getSuperclass()`

Big Numbers

New Java™ 1.1 Math classes

- `BigInteger(String)`
- `BigDecimal(String)`
- Arithmetic operations (used for RSA cryptography and signed applets)
  - `add`
  - `subtract`
  - `multiply`
  - `divide`
  - `pow`
  - `max`
  - `min`
  - `negate`
  - `remainder`
  - `mod`
  - `isProbablePrime`
  - `modPow`
  - `divideAndRemainder`
  - `shiftLeft` and `shiftRight`
  - `compareTo`

Date and Calendars

Basic time telling (see chap.03/Cal.java)

- `Date: seconds since Jan 1, 1970, 00:00:00 UCT`
- `Easy to use but not internationalizable`
- `Many "deprecated" methods`
- `Calendar (abstract) and GregorianCalendar`
  - `Harder to use but have support for internationalization`
  ```java
  Date currentTime = new Date();
  int Hours = currentTime.getHours();
  int Minutes = currentTime.getMinutes();
  int Seconds = currentTime.getSeconds();
  ```
  ```java
  GregorianCalendar now = new GregorianCalendar();
  int Hours = now.get(Calendar.HOUR);  
  int Minutes = now.get(Calendar.MINUTE);  
  int Seconds = now.get(Calendar.SECOND);  
  ```