R·I·T

Computer Science II
4003-232-07 (Winter 20072)

Week 2: Inheritance, Polymorphism, and Interfaces

Richard Zanibbi
Rochester Institute of Technology

R·I·T

Polymorphism and Casting

(text 9.7 – 9.8)

Dynamic Binding (or *Polymorphism* of Methods)

Definition
- Selecting the definition of a method to invoke at runtime (i.e. which definition to *bind* to the method call)
- Must match method name, number, order and types of arguments
- Important when methods can be overridden (e.g. toString())

Dynamic Binding In Java
The search for which definition to bind to a method call starts from the actual (constructed) class of an object, or a named class, and proceeds up the inheritance hierarchy towards Object.

Example
PolymorphismDemo.java (Liang pp. 311-312)

GraduateStudent → Student → Person → Object

Dynamic Binding and Arguments

Method Arguments in Java
- May be of any type that is considered a subtype (e.g. subclass) of the parameter type.
- e.g. *public static void m(Object x)* from the previous example will accept any object belonging to a subclass of Object as an argument (i.e. from any class!)
- e.g. *public static void p(double x)* may accept any of the numeric types for x (byte, short, int, long, float, double) and implicitly perform a *widening type conversion* (cast) if necessary: see p.40 in course text.
- For overloaded methods, start with actual operands' type and use the method definition with the *most specific* ('lowest') accepting formal parameter
- Example: OverloadedNumbers.java

Generic Programming
- Takes advantage of dynamic binding, ability to handle many types in the same way (*generically*) and invoke overridden methods

The '`instanceof`' operator

Use
A boolean operator that tests whether an object belongs to a given class.

Examples
Circle myCircle = new Circle(1.0);
- myCircle instanceof Circle // true
- myCircle instanceof Object // true
- myCircle instanceof String // false

Type Casting Objects

Upcasting
- Converting the type of an object to a superclass ("up" the inheritance/type hierarchy). Usually not explicit, as properties of superclasses are inherited by subclasses automatically.
- e.g. Object o = new Student(); // Student referenced as an Object
- Similarly, parameters of type Object may accept objects of any other type, with an implicit cast to class Object.

Downcasting
- Converting the type of an object to a subclass ("down" the inheritance hierarchy). Requires explicit casting, with a check to ensure that the cast will be successful using instanceof.
- e.g. if (o instanceof Student) Student s = (Student) o;
- e.g. TestPolymorphismCasting.java (Liang p. 315)

Why do we need to check types before downcasting?

Precedence of Cast vs. Dot operator

Caution!

The access (dot) operator has higher precedence than type casting.

Fix:

Put results casting operations in brackets when paired with access operators, e.g.

((Circle)object).getArea() vs.
(Circle)object.getArea()

Subtle Point: Matching the Method vs. Selecting the Method Definition

Matching the Method Signature (static)

- For objects, the selection of which method signature to use is determined at compile time based on the type of a reference.
- Put another way, the type of a reference to an object determines which class interface is active for an object
- If the active class interface is a superclass of the class that defines a desired method, it will not be found.
  - e.g. Object o = new Circle(1); o.getRadius() // won't work.
  - Object o = new Circle(1); ((Circle)o).getRadius() // will work.

Selecting the Method Definition (dynamic)

Is done dynamically at runtime (dynamic binding). The actual (constructed) class determines the implementation used.

Methods in the Object Class

boolean equals(Object o)

Test if another object is the same as the current one (test by *reference value*)

int hashCode()

Used to define integer hash codes for hash sets (a type of data structure). In Object, this is the object's address.

Object clone()

Copies the state of an object to produce a copy.
e.g. `int[] targetArray = (int []) sourceArray.clone();`

void finalize()

- Invoked by the garbage collector before an object is destroyed.
- Objects without a reference are "garbage."
- By default, does nothing.
- You should never invoke finalize() in a program!
- Example: FinalizationDemo.java (p. 323)

Class getClass()

- The JVM creates objects to represent classes ("meta-objects"), including the class name, constructors, and methods. There is a class "Class" used to define these.
- It is possible to query a Class object to get information about a class at runtime.
- Every object may be asked to return the Class object (meta-object) with which it is associated using getClass()

getClass() example

```
Object obj = new Object();
Class metaObject = obj.getClass();
System.out.println("Class is: "
    + metaObject.getName());
```

produces

```
Class is: java.lang.Object
```


Hiding Data and Methods

- **Static Methods**
- **Static/Instance Data Members**

cannot be overridden; only hidden. (Avoid this!)

Accessing Hidden Methods and Data

- Using super() in the subclass
- Using a reference variable of the superclass type (i.e. use the superclass type (interface))
- Unlike instance methods, static methods and data members are bound at compile time ("statically")
- *Example: HidingDemo.java* (p. 326)
- ** Static methods and fields can always be accessed directly using the class name (*if* it is visible, using *Class.staticMethod()*)

Exercise: Polymorphism

A. What is wrong with the following code?

```
public class Test {
    public static void main(String[] args) {
        Object fruit = new Fruit();
        Object apple = (Apple) fruit;
    }
}

class Apple extends Fruit {  }
class Fruit { }
```


B. Indicate whether each of the following statements are valid or invalid.

```
1.  Object o1 = new String("test");
2.  if (o1 instanceof String) { };
3.  if (o1 instanceof Circle) { };
4.  Object o2 = new Circle(1); // passed radius
5.  String s1 = o1;
6.  String s2 = (String) o1;
7.  Object o3 = (Object) s2;
8.  String s4 = ((Object)o1).toString();
9.  String s5 = (Circle)o2.toString();
10. Object o4 = (Object) o1;
```

C. Why should instanceof be used before performing a downcast of an Object?

R·I·T

Abstract Classes and Interfaces

Abstract Method

Definition

A method which has a signature, but no body. All abstract methods are instance methods (non-static).

- e.g. public abstract int deviseNumber();

Purpose

– Class design: permits defining a method signature whose definition may be provided in subclasses.

– Through dynamic binding and overriding, this will allow different versions of the method to be invoked at run time for objects that belong to the abstract class, but different actual subclasses (example later).

Abstract Class

Definition

– A class which may not have any instances created from it, used solely to define subclasses of itself. Otherwise, it is a normal class, and is included in the class inheritance hierarchy.

– All classes that contain abstract methods *must* be declared abstract.

– e.g. public abstract class GeometricObject() { }

Class Design

– In general, superclasses should be designed to contain common features of subclasses (to maximize code reuse, e.g. Object class)

– Abstract classes are useful for defining and using common data and behaviours for subclasses that may represent significantly different object types.

– Defines an interface for these (possibly very) different subclasses

Concrete Class

A class which may be used to create instances .
Abstract classes are not concrete classes.

Additional Notes on Abstract Classes

– May still have constructors defined (though protected access more appropriate than public)

– May be used as a data type for reference vars.

- e.g. GeometricObject g = new Circle(1.0);
- This is one of their key uses; as an interface to access common data and methods of different subclasses.

Subclasses of Abstract Classes

– Must implement all abstract methods, or also be declared abstract (no instances).

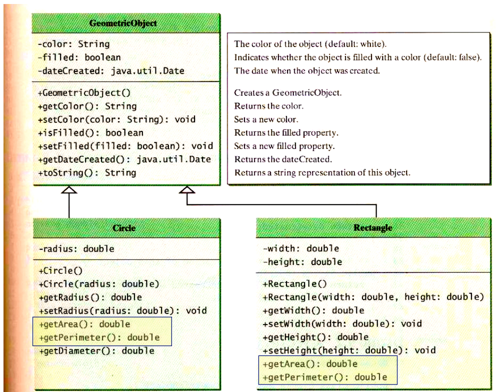

FIGURE 9.1 The GeometricObject class is the superclass for Circle and Rectangle.

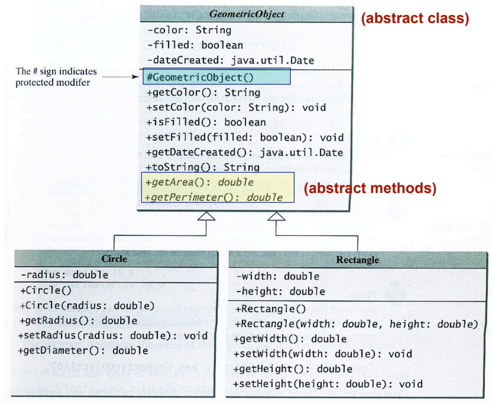

FIGURE 10.1 The new GeometricObject class contains abstract methods.

Examples

TestGeometricObject.java (p. 345)

Uses of Abstract Class:

– Abstract class allows us to get areas and perimeters using a single interface, though the computation of these in the Circle and Rectangle classes are completely different.

Another Example (Text 10.3)

Calendar class (abstract) and Gregorian Calendar (concrete subclass of Calendar)

A Yet More Restricted Class Type: Interfaces in Java

Definition

– A type of class which defines only (*public static final*) constants and (*public*) abstract instance methods.

– Provides a data type for reference variables from which the *interface* may be used to act on objects associated with the interface type.

– NOTE: Interfaces are *not* part of the class hierarchy

Java Syntax

Defined using the “interface” rather than “class” keyword

- e.g. public interface Cloneable { ... }

Motivation for Interfaces in Java

Multiple Inheritance is Prohibited

We cannot inherit from multiple classes; in particular, state is only inherited through a strict linear path in the inheritance tree (hierarchy)

But...

– Still wish to allow very different data types (classes) to have common methods to support generic programming (e.g. the ability to compare objects using a single interface (Comparable))

– A class may ‘implement’ one or more interfaces to support these ‘generic’ types of computation.

Example: Comparable Interface

Purpose

– Allow definition of a method for determining which of a pair of objects of the same class is ‘larger’ or if they are the ‘same size.’

– Can then use compareTo() to compare Strings, Students (e.g. by student id), Geometric objects (e.g. by area), etc. using a single method with different definitions (one per class)

Returns:
-1: this < argument
0: this, argument same
1: this > argument

```
package java.lang

public interface Comparable {
    public abstract int compareTo(Object o);
}
```

Example Classes Using Comparable Interface

public class String extends Object **implements** Comparable { ... }

public class Date extends Object **implements** Comparable { ... }

Interface as a Reference Variable Type

The following are valid for String object s and Date object d:

- s instanceof String,
- s instanceof Object,
- s instanceof Comparable
- d instanceof java.util.Date,
- d instanceof Object,
- d instanceof Comparable

Example of a 'Generic' Comparison Function

```
public class Max {
    public static Comparable max(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2
    }
}
```

Example Usage:

```
String s1 = "a"; String s2 = "b";
String s3 = (String)Max.max(s1,s2);

Date d1 = new Date(); Date d2 = new Date();
Date d3= (Date)Max.max(d1,d2);
```

Any class that implements Comparable can be used with Max.max() (e.g. a revised Rectangle class (see p. 350))

Interfaces and Inheritance

Classes Implementing Interfaces

– Concrete and Abstract classes may inherit from only one parent.

– However, they may implement multiple interfaces.

```
public class A extends B implements InterfaceA,
     InterfaceB, ..., InterfaceN { }
```

Interfaces extending Interfaces

Interfaces may inherit from and extend one or more interfaces.

```
public Interface NewInterface extends IntA, ...., IntN { };
```


UML Representation of Inheritance/Implementation

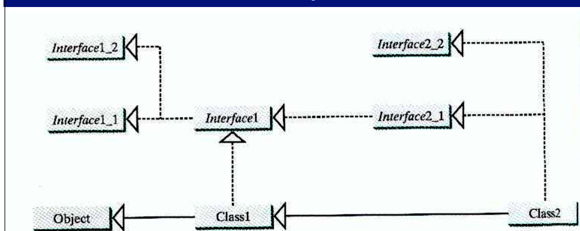

FIGURE 10.5 Abstract class *Class1* implements *Interface1*, *Interface1* extends *Interface1_1* and *Interface1_2*. *Class2* extends *Class1* and implements *Interface2_1* and *Interface2_2*.

Objects of Class2 are instances of *all* the other classes and interfaces shown. This means variables referring to a Class2 object may be any of these types.

Marker Interfaces

Marker Interface

– An interface that contains no constants or methods; 'flags' a class as having certain properties (e.g. to tell Java to permit certain operations)

– e.g. "Cloneable" (designates that objects of a class may have their state copied into another object)

```
public class House implements Cloneable, Comparable { ... }

House h1 = new House(1,1750.50); // id, area
House h2 = (House)house1.clone();
```

.. See text Section 10.4.4 for details.

****Shallow vs. Deep Copies**

– Shallow: object references copied by value (copies reference to a single object) - danger of manipulating the "original" data in this case

– Deep: object data is copied into new objects, and "copied" references point to the new objects and not the original ones (e.g. using clone())