R·I·T

Object-Oriented Programming: Inheritance

Object Oriented Programming

Paradigm:

Represent programs as a set of objects that encapsulate data and methods (state and behaviour) and pass messages between one another.

Key Object Oriented Concepts:

Class (template for a set of objects)

–Class ('static') variables that belong to a class

–Class ('static') methods that belong to a class

Instances (*objects),* each with state and behavior

–Instance variables that belong to *individual objects*

–Instance methods that are associated with *individual objects* (but defined once!)

Main Elements of a Java Class

1. **Class signature**
   - Name, access modifiers (public, private, etc.), relationships with other classes, etc.

2. **Class ('static') properties**
   - Data members (variables, constants)
   - Methods: accessors, mutators, other methods
     - *cannot* reference (use) instance variables

3. **Instance properties**
   - Data members (variables, constants)
   - Methods: accessors, mutators, other methods
     - can reference (use) static and instance variables

4. **(Instance) Constructors**
   - Used by the 'new' operator to initialize constructed instances

```
public class MyClass { // CLASS SIGNATURE
        private static int numberObjects = 0;    // CLASS DATA
        private int instanceVariable;            // INSTANCE DATA

        public MyClass(int value){               // CONSTRUCTOR
               instanceVariable = value;
               numberOfObjects++;
        }

        public int getInstanceVariable() { // INSTANCE METHOD
              return instanceVariable;
        }

        public static int getNumberObjects() {   // CLASS METHOD
               return numberObjects;
               // CANNOT refer to instanceVariable here
        }

        public static void main(String[] args) { // CLASS METHOD
               MyClass instance = new MyClass(5);
               MyClass instance2 = new MyClass(6);
               System.out.println(numberObjects + ": " +
                      instance.getInstanceVariable() +
                      instance2.getInstanceVariable() );
        }
}
```

What is Inheritance?

Definition

A new class taking the definition of an existing class as the starting point for it's own definition

Superclass

The existing *("parent")* class providing the initial definition for the new *"derived"* or *"child"* class

Subclass

A class derived from an existing class (*"child class"*)

In Java

Only accessible (e.g. non-private) data members and methods are inherited by a subclass *definition*. Constructors are also not inherited.

NOTE that objects of subclasses still have properties of the superclass.

Inheritance is a formalized type of 'code-reuse'

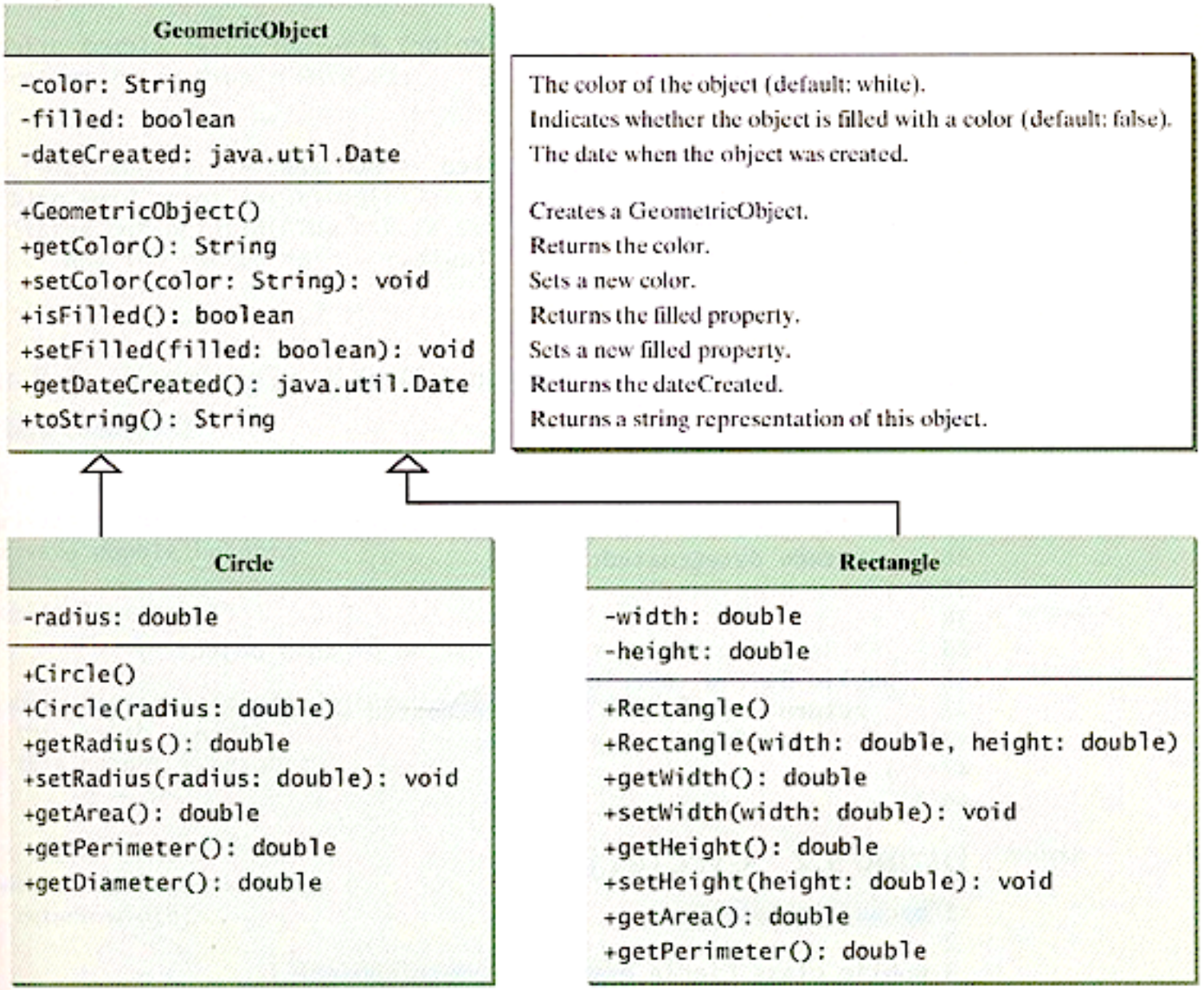


FIGURE 9.1 The `GeometricObject` class is the superclass for `Circle` and `Rectangle`.

The Inheritance Hierarchy: What happens if Class A inherits from Class B?

Effect on Object Properties

Objects from a class possess:

- Instance data & methods of the class
- Instance data & methods of the superclass
- Instance data & methods of the superclass' superclass
- ... and so on, up to the Object class in the class inheritance hierarchy.

Invoking Instance Methods

May invoke accessible methods of an object for the *reference variable* class, and any preceding classes in the inheritance hierarchy

```
String x = "Hi there.";  // String and Object methods usable on x
Object a = x;            // Only Object methods may be invoked on a.
```

Inheritance in Java

Syntax

Use “extends” keyword

e.g. class NewClass **extends** AnotherClass { ... }

‘Object’ as the “Parent of them all”

All classes in Java extend (inherit from) the object class.

```
public class NewClass{} =
     public class NewClass extends Object{}
```

Multiple Inheritance

A class inheriting from more than one parent class

- Not permitted in Java
- Is permitted in other languages such as C/C++

getClass() example

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

produces

```
Class is: java.lang.String
```

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 Object References

Upcasting References

- Converting an object reference type to a superclass ("up" the inheritance/type hierarchy). Does not need to be explicit.
- e.g. Object o = new Student(); // a Student referenced as an Object
- e.g. reference parameters of type Object may accept objects of any other type (implicitly cast to an Object reference)

Downcasting References

- Converting an object reference type to a subclass ("down" the inheritance hierarchy). Requires explicit casting & using instanceof.
- e.g. if (o instanceof Student) Student s = (Student) o;
- e.g. TestPolymorphismCasting.java

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

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

What is '`this`' ?

Definition

- A reference to 'myself' for an object
- Used within instance methods for object invoking the method
- All instance variable references and method invocations implicitly refer to 'this'
  - Within an instance method: x = 2 *same as* this.x = 2; toString() *same as* this.toString())

Some Uses

1. Prevent masking of variables, e.g. formal params. and instance variables in a constructor:

```
public MyClass(int x){ this.x = x; }
```

2. Invoke other constructors within a class
   - Note: this(arg-list) must be first statement in constructor definition

```
public MyClass(int x){ this(); this.x = x; }
```

3. Have object pass itself as a method argument

```
someClass.printFancy(this);
```

The 'super' keyword

Purpose

Provides a reference to the superclass of the class in which it appears

Uses

1. Invoke a superclass constructor
   - Similar to using 'this,' the call to 'super(arg1, arg2, ...)' must be the first statement in a constructor if present.

2. Invoke a superclass method that has been overridden
   - e.g. we can use super.toString() to invoke the toString() method of the superclass rather than that in the current class
   - Similar to 'this,' it is possible but not necessary to use super to invoke all inherited methods from the superclass (implicit)
   - **Warning:** we cannot 'chain' super, as in super.super.p()

The Inheritance Hierarchy and *Constructor Chaining*

Calling a constructor

Normally invokes default constructors for each class from root of the inheritance hierarchy, starting with *Object*

- This is necessary to ensure that all inherited data is properly initialized according to the class definitions.

e.g. public A() { } = public A() { super(); }

Example

Faculty class

A Warning About Constructor Chaining in Java...

Default Constructor ("no-arg constructor")

Is automatically defined if no constructor is given by the programmer, otherwise it must be explicitly defined to exist

This Means...

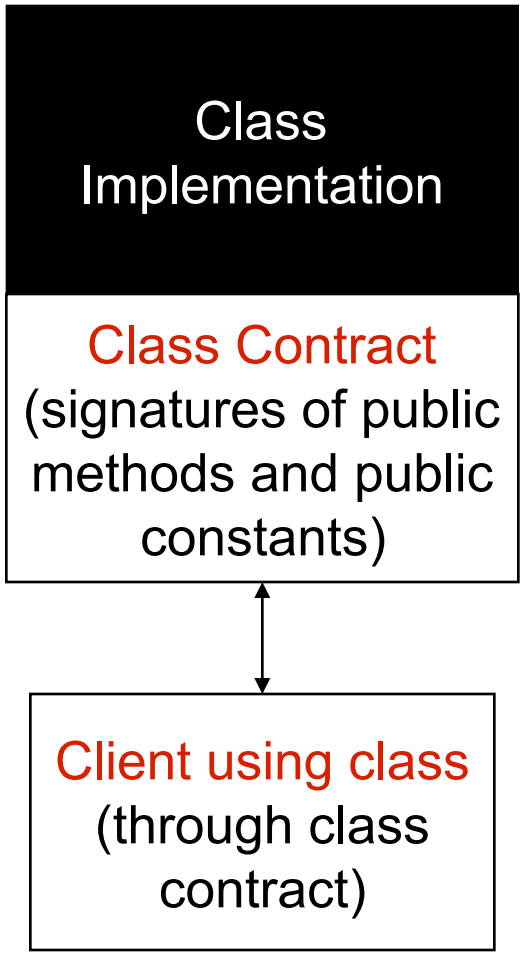
That an error occurs if we go to construct an object and one of its ancestor classes in the inheritence hierarchy does not have a *default* constructor defined.

Fix:

If a class may be extended, explicitly define a default constructor to avoid this situation.

More naïve approach: always define a default constructor.

Class Contract



- Collection of methods and data members accessible outside of a class
- Includes description of data members and method signatures

Method Signature

Name, return type, and parameter types for a method

e.g. boolean isDaytime(int seconds)

Visibility Modifiers

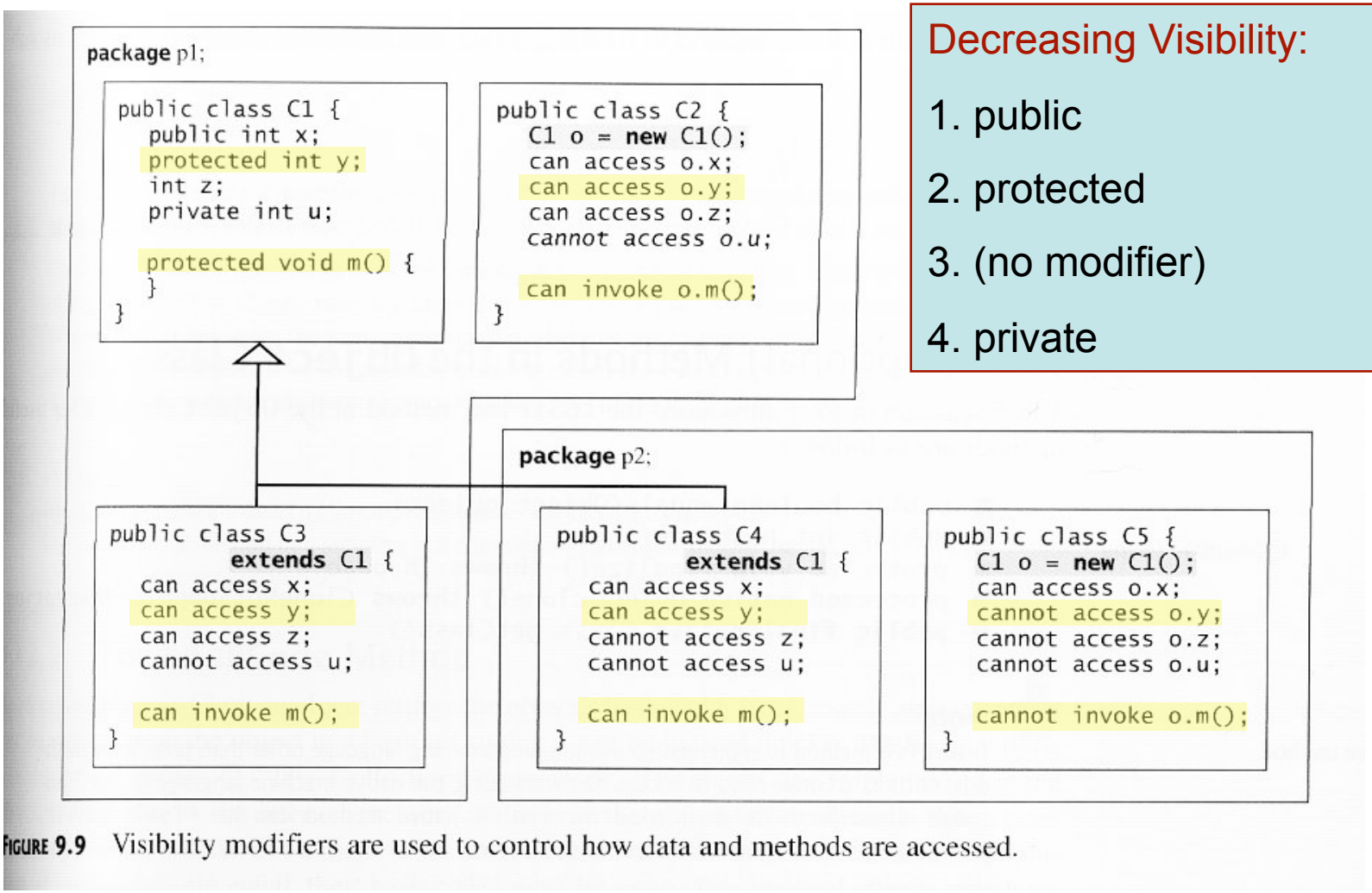


FIGURE 9.9 Visibility modifiers are used to control how data and methods are accessed.

Decreasing Visibility:

1. public
2. protected
3. (no modifier)
4. private