


# **Chapter 5**

# **Inheritance and**

# **polymorphism**



# **Inheritance and polymorphism**

# 1 Introduction

Object-oriented programming allows you to define new classes from existing classes. This is called inheritance.

# 2 Superclasses and Subclasses

Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).

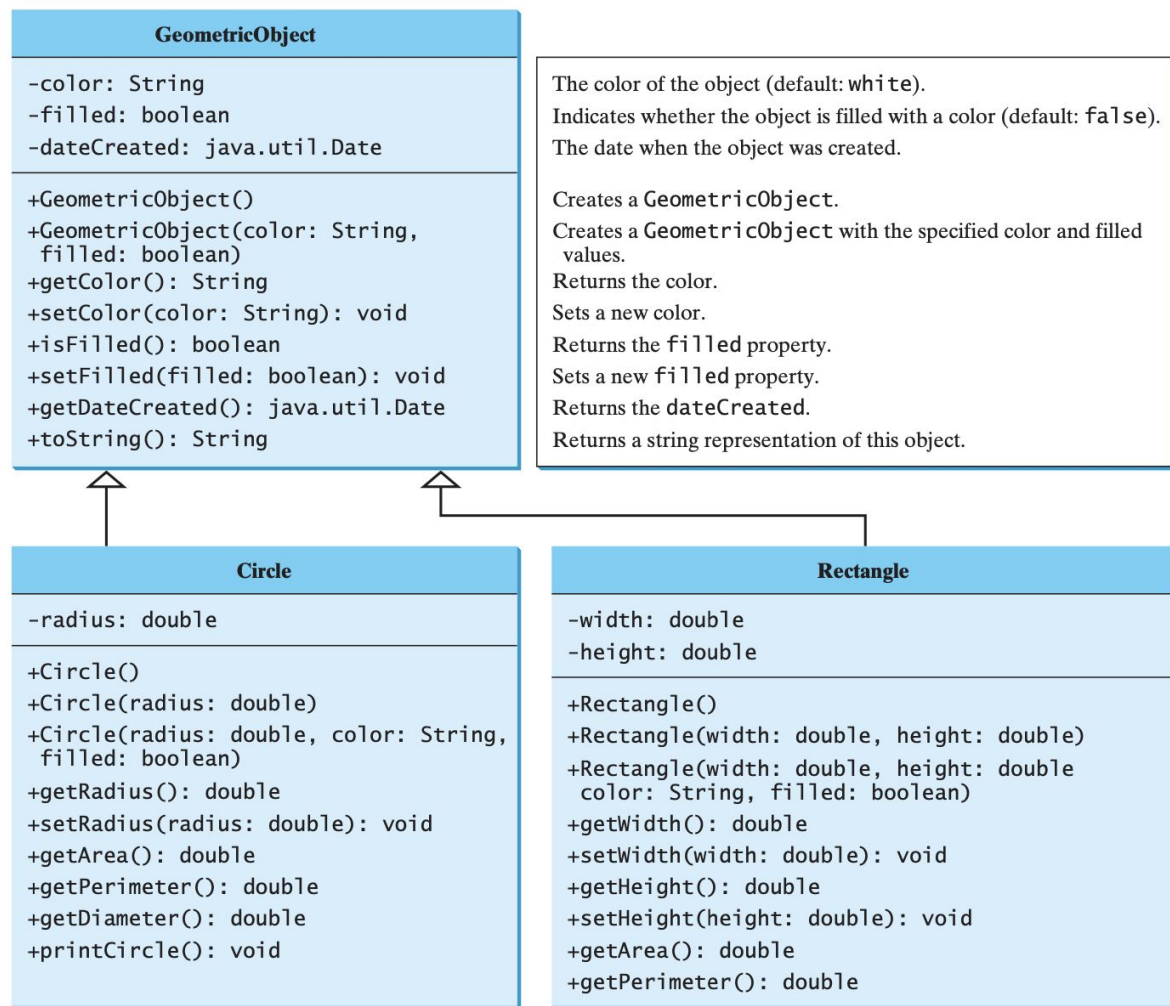
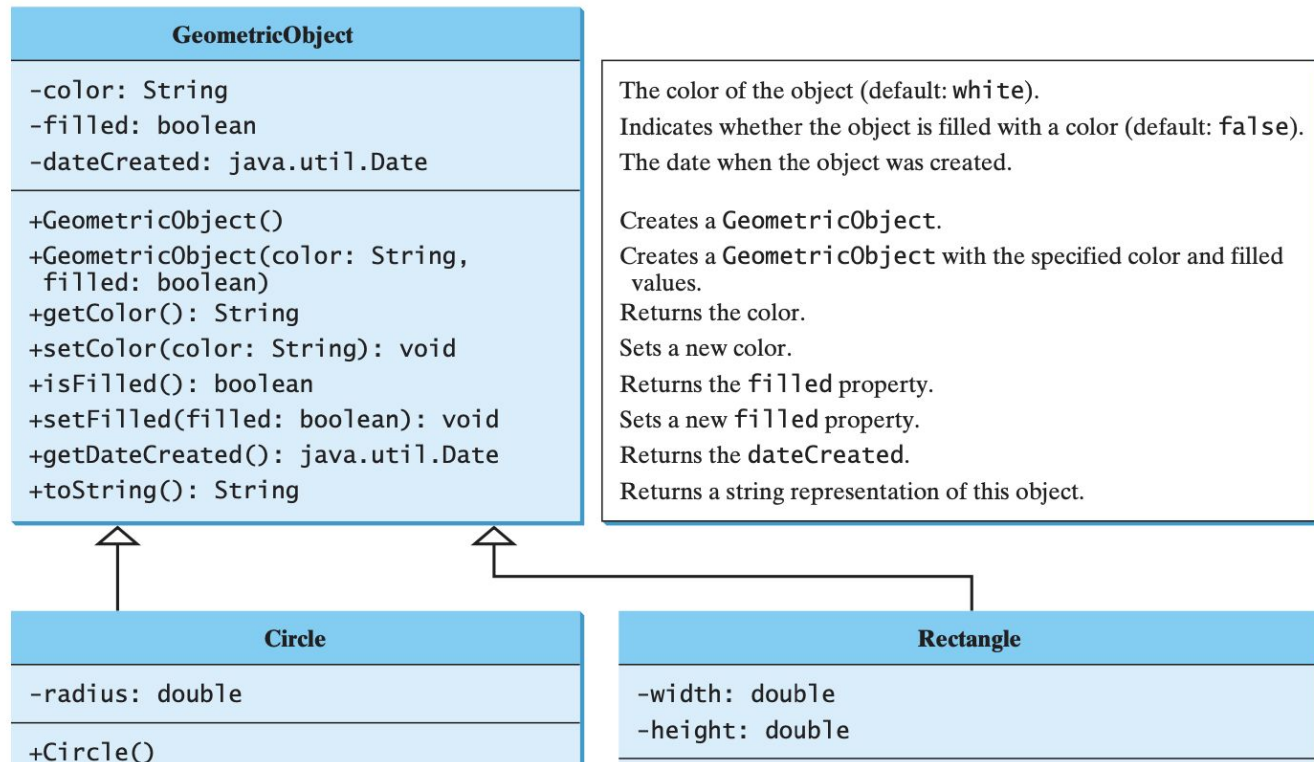


FIGURE 11.1 The **GeometricObject** class is the superclass for **Circle** and **Rectangle**.

## 2 Superclasses and Subclasses

Subclass  
Superclass

`public class Circle extends GeometricObject`



## 2 Superclasses and Subclasses

- A subclass is not a subset of its superclass. A subclass usually contains more information and methods than its superclass.
- Inheritance is used to model the is-a relationship. Do not blindly extend a class just for the sake of reusing methods.
- A Java class may inherit directly from only one superclass: single inheritance.

# 3 Using the *super* Keyword

- The keyword *super* refers to the superclass and can be used to:
  - To call a superclass constructor.
  - To call a superclass method.

```
public class SimpleGeometricObject {  
    private String color = "white";  
    private boolean filled;  
    private java.util.Date dateCreated;  
    public SimpleGeometricObject(String color, boolean filled) {  
        dateCreated = new java.util.Date();  
        this.color = color;  
        this.filled = filled;  
    }  
}  
  
public class CircleFromSimpleGeometricObject extends SimpleGeometricObject {  
    private double radius;  
    public CircleFromSimpleGeometricObject(double radius, String color,  
        boolean filled) {  
        super(color, filled);  
        this.radius = radius;  
    }  
}
```

# 3 Using the *super* Keyword

## Constructor Chaining

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically calls `super()`

```
public ClassName() {  
    // some statements  
}
```

Equivalent

```
public ClassName() {  
    super();  
    // some statements  
}
```

```
public ClassName(double d) {  
    // some statements  
}
```

Equivalent

```
public ClassName(double d) {  
    super();  
    // some statements  
}
```



# 3 Using the *super* Keyword

- Constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain.
- If possible, you should provide a no-arg constructor for every class to make the class easy to extend and to avoid errors.

```
1 public class Faculty extends Employee {
2     public static void main(String[] args) {
3         new Faculty();
4     }
5
6     public Faculty() {
7         System.out.println("(4) Performs Faculty's tasks");
8     }
9 }
10
11 class Employee extends Person {
12     public Employee() {
13         this("(2) Invoke Employee's overloaded constructor");
14         System.out.println("(3) Performs Employee's tasks ");
15     }
16
17     public Employee(String s) {
18         System.out.println(s);
19     }
20 }
21
22 class Person {
23     public Person() {
24         System.out.println("(1) Performs Person's tasks");
25     }
26 }
```

# 3 Using the super Keyword

## Calling Superclass Methods

```
super.method(parameters);
```

## 4 Overriding Methods

To override a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass.

```
1 public class CircleFromSimpleGeometricObject
2     extends SimpleGeometricObject {
3     // Other methods are omitted
4
5     // Override the toString method defined in the superclass
6     public String toString() {
7         return super.toString() + "\nradius is " + radius;
8     }
9 }
```

# 5 Overriding vs. Overloading

- Overloading means to define multiple methods with the same name but different signatures.
- Overriding means to provide a new implementation for a method in the subclass.

# 5 Overriding vs. Overloading

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

(a)

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

(b)

## 6 The Object Class and Its toString() Method

- Every class in Java is a subclass of the java.lang.Object class.

```
public class  ClassName {  
    ...  
}
```

Equivalent

```
public class  ClassName  extends  Object {  
    ...  
}
```

# 7 Polymorphism

- A class defines a type.
- A type defined by a subclass is called a **subtype**.
- A type defined by its superclass is called a **supertype**.
- Polymorphism means that **a variable of a supertype can refer to a subtype object**.
- Every **instance of a subclass** is also an **instance of its superclass**, but not vice versa.  
=> Pass an instance of a subclass to a parameter of its superclass type

# 7 Polymorphism

## LISTING 11.5 PolymorphismDemo.java

```
1  public class PolymorphismDemo {
2      /** Main method */
3      public static void main(String[] args) {
4          // Display circle and rectangle properties
5          displayObject(new CircleFromSimpleGeometricObject
6                        (1, "red", false));
7          displayObject(new RectangleFromSimpleGeometricObject
8                        (1, 1, "black", true));
9      }
10
11     /** Display geometric object properties */
12     public static void displayObject(SimpleGeometricObject object) {
13         System.out.println("Created on " + object.getDateCreated() +
14                            ". Color is " + object.getColor());
15     }
16 }
```



## 8 Dynamic Binding

- A method can be defined in a superclass and overridden in its subclass.
- The JVM decides which method is invoked at runtime (depending on the actual type of the object calling the method).
- Two terms: declared type and actual type.
- The type that declares a variable is called the declared type of the variable.
- The actual type of the variable is the actual class for the object referenced by the variable.

```
Object o = new GeometricObject();  
System.out.println(o.toString());
```

declared type: Object

actual type: GeometricObject

toString() of the actual type is invoked.

# 8 Dynamic Binding

## LISTING 11.6 DynamicBindingDemo.java

```
1  public class DynamicBindingDemo {
2      public static void main(String[] args) {
3          m(new GraduateStudent());
4          m(new Student());
5          m(new Person());
6          m(new Object());
7      }
8
9      public static void m(Object x) {
10         System.out.println(x.toString());
11     }
12 }
13
14 class GraduateStudent extends Student {
15 }
16
17 class Student extends Person {
18     @Override
19     public String toString() {
20         return "Student" ;
21     }
22 }
23
24 class Person extends Object {
25     @Override
26     public String toString() {
27         return "Person" ;
28     }
29 }
```

## 9 Casting Objects and the *instanceof* Operator

- One object reference can be typecast into another object reference.
- This is called casting object.

`m(new Student());`

assigns the object `new Student()` to a parameter of the `Object` type.

### LISTING 11.6 `DynamicBindingDemo.java`

```
1 public class DynamicBindingDemo {
2     public static void main(String[] args) {
3         m(new GraduateStudent());
4         m(new Student());
5         m(new Person());
6         m(new Object());
7     }
8
9     public static void m(Object x) {
10        System.out.println(x.toString());
11    }
12 }
13
14 class GraduateStudent extends Student {
15 }
16
17 class Student extends Person {
18     @Override
19     public String toString() {
20         return "Student" ;
21     }
22 }
23
24 class Person extends Object {
25     @Override
26     public String toString() {
27         return "Person" ;
28     }
29 }
```

# 9 Casting Objects and the *instanceof* Operator

```
Object o = new Student();  
Student b = o; //a compile error
```

- Even though `o` is really a `Student` object, the compiler is not clever enough to know it.
- To tell the compiler that `o` is a `Student` object, use explicit casting.

```
Student b = (Student)o; // Explicit casting
```

- To ensure that the object is an instance of another object before attempting a casting, use the `instanceof` operator

```
if (o instanceof Student){  
    Student b = (Student)o;  
}
```

# 10 The Object's equals Method

- Like the `toString()` method, the `equals(Object)` method is another useful method defined in the `Object` class.

//check the same reference, should override to check the same content

```
public boolean equals(Object o){  
    return (this == obj);  
}
```

- Override equals in the `Circle` class.

```
public boolean equals(Object o) {  
    if (o instanceof Circle)  
        return radius == ((Circle)o).radius; // check radius  
    else  
        return this == o;  
}
```

# 11 The ArrayList Class

- An ArrayList object can be used to store an unlimited number of objects.
- ArrayList is known as a generic class with a generic type E.
- You can specify a concrete type to replace E when creating an ArrayList.

```
ArrayList<String> cities = new ArrayList<String>();
```

# 11 The ArrayList Class

**java.util.ArrayList<E>**

```
+ArrayList()  
+add(o: E): void  
+add(index: int, o: E): void  
+clear(): void  
+contains(o: Object): boolean  
+get(index: int): E  
+indexOf(o: Object): int  
+isEmpty(): boolean  
+lastIndexOf(o: Object): int  
+remove(o: Object): boolean  
  
+size(): int  
+remove(index: int): boolean  
  
+set(index: int, o: E): E
```

Creates an empty list.

Appends a new element *o* at the end of this list.

Adds a new element *o* at the specified index in this list.

Removes all the elements from this list.

Returns true if this list contains the element *o*.

Returns the element from this list at the specified index.

Returns the index of the first matching element in this list.

Returns true if this list contains no elements.

Returns the index of the last matching element in this list.

Removes the first element *o* from this list. Returns true if an element is removed.

Returns the number of elements in this list.

Removes the element at the specified index. Returns true if an element is removed.

Sets the element at the specified index.

**FIGURE 11.3** An **ArrayList** stores an unlimited number of objects.

# 11 The ArrayList Class

**TABLE 11.1** Differences and Similarities between Arrays and **ArrayList**

<i>Operation</i>	<i>Array</i>	<i>ArrayList</i>
Creating an array/ArrayList	<code>String[] a = new String[10]</code>	<code>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;();</code>
Accessing an element	<code>a[index]</code>	<code>list.get(index);</code>
Updating an element	<code>a[index] = "London";</code>	<code>list.set(index, "London");</code>
Returning size	<code>a.length</code>	<code>list.size();</code>
Adding a new element		<code>list.add("London");</code>
Inserting a new element		<code>list.add(index, "London");</code>
Removing an element		<code>list.remove(index);</code>
Removing an element		<code>list.remove(Object);</code>
Removing all elements		<code>list.clear();</code>



# 12 Useful Methods for Lists

- Create an arraylist from an array

```
String[] array = {"red", "green", "blue"};  
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

- Create an array of objects from an arraylist

```
String[] array1 = new String[list.size()];  
list.toArray(array1);
```

- Sort the elements

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};  
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));  
java.util.Collections.sort(list);  
System.out.println(list);
```

## 12 Useful Methods for Lists

- max and min methods

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};  
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));  
System.out.println(java.util.Collections.max(list));  
System.out.println(java.util.Collections.min(list));
```

- perform a random shuffle for the elements in a list

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};  
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));  
java.util.Collections.shuffle(list);  
System.out.println(list);
```

# 13 The protected Data and Methods

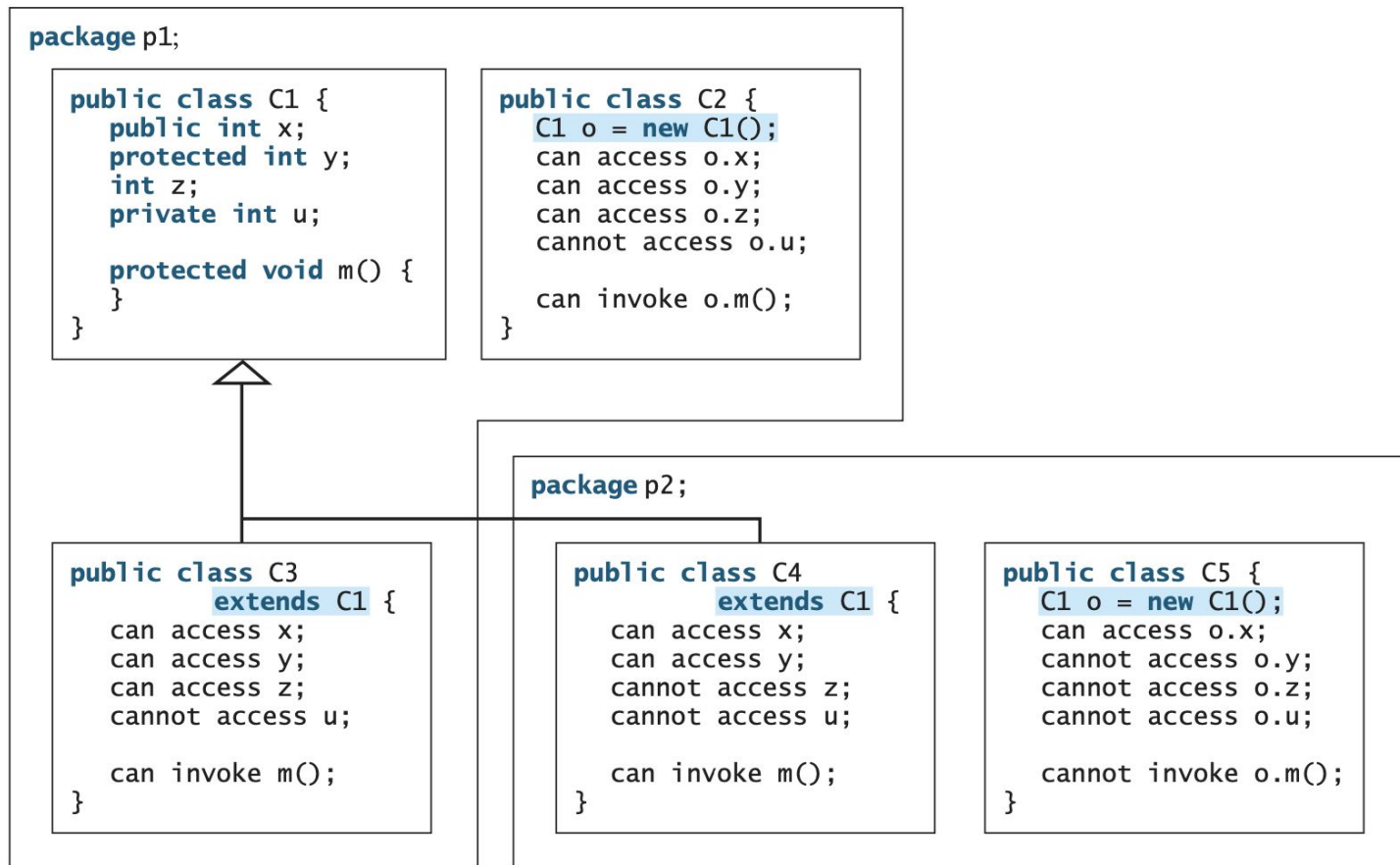
A protected member of a class can be accessed from a subclass.

Visibility increases  
→  
private, default (no modifier), protected, public

**TABLE 11.2** Data and Methods Visibility

<i>Modifier on members in a class</i>	<i>Accessed from the same class</i>	<i>Accessed from the same package</i>	<i>Accessed from a subclass in a different package</i>	<i>Accessed from a different package</i>
public	✓	✓	✓	✓
protected	✓	✓	✓	—
default (no modifier)	✓	✓	—	—
private	✓	—	—	—

# 13 The protected Data and Methods



**FIGURE 11.5** Visibility modifiers are used to control how data and methods are accessed.

# 14 Preventing Extending and Overriding

Neither a `final class` nor a `final method` can be extended. A `final data field` is a constant.

```
public final class A {  
    // Data fields, constructors, and methods omitted  
}
```

```
public class Test {  
    // Data fields, constructors, and methods omitted  
    public final void m() { // Do something  
    }  
}
```

# **ABSTRACT CLASSES AND INTERFACES**

# 1 Introduction

A superclass defines common behavior for related subclasses.

An interface can be used to define common behavior for classes (including unrelated classes).

## 2 Abstract Classes

- In the inheritance hierarchy, classes become more specific and concrete with each new subclass.
- Class design should ensure that a superclass contains common features of its subclasses.
- Sometimes a superclass is so abstract that it cannot be used to create any specific instances. Such a class is referred to as an abstract class.



## 2 Abstract Classes

Common methods:

getArea()

getPerimeter()

their implementation depends on  
the specific type of geometric object

=> abstract methods

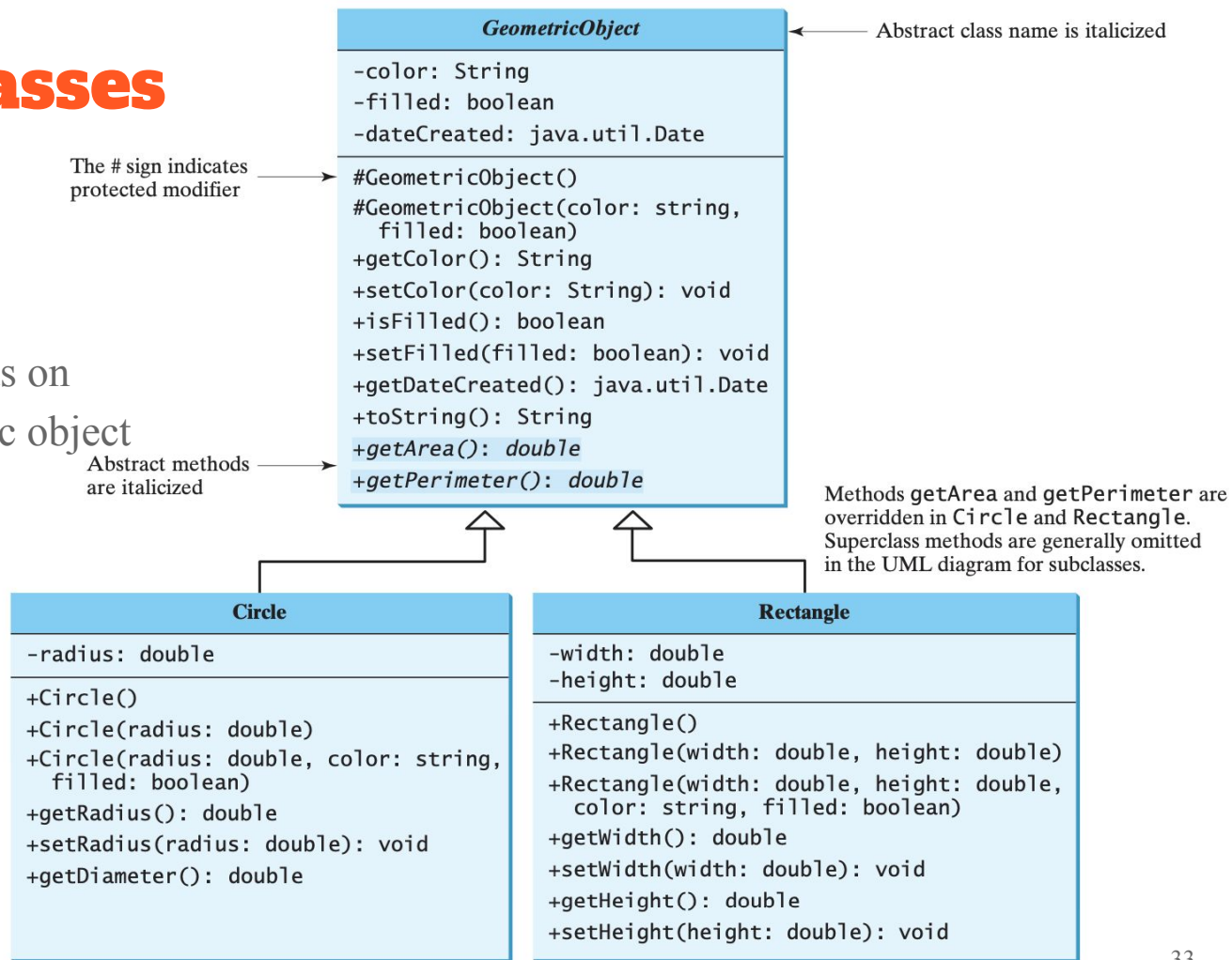


FIGURE 13.1 The new **GeometricObject** class contains abstract methods.

```

public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    protected GeometricObject() {
        dateCreated = new java.util.Date();
    }

    protected GeometricObject(String color, boolean
filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

```

```

    public boolean isFilled() {
        return filled;
    }

    public void setFilled(boolean filled) {
        this.filled = filled;
    }

    public java.util.Date getDateCreated() {
        return dateCreated;
    }

    @Override
    public String toString() {
        return "created on " + dateCreated + "\ncolor: " +
color +
        " and filled: " + filled;
    }

    /** Abstract method getArea */
    public abstract double getArea();

    /** Abstract method getPerimeter */
    public abstract double getPerimeter();
}

```

```
public class Circle extends GeometricObject {  
    private double radius;
```

```
  
    public Circle() {  
    }  

```

```
    public Circle(double radius) {  
        this.radius = radius;  
    }  

```

```
    @Override /** Return area */  
    public double getArea() {  
        return radius * radius * Math.PI;  
    }  

```

```
    @Override /** Return perimeter */  
    public double getPerimeter() {  
        return 2 * radius * Math.PI;  
    }  
    //.....  
}
```

```
public class Rectangle extends GeometricObject {  
    private double width;  
    private double height;
```

```
  
    public Rectangle() {  
    }  

```

```
    public Rectangle(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  

```

```
    @Override /** Return area */  
    public double getArea() {  
        return width * height;  
    }  

```

```
    @Override /** Return perimeter */  
    public double getPerimeter() {  
        return 2 * (width + height);  
    }  
}
```

## 2 Abstract Classes

- In a non-abstract subclass extended from an abstract class, all the abstract methods must be implemented.
- Object cannot be created from abstract class, but you can still define its constructors, which are invoked in the constructors of its subclasses.
- It is possible to define an abstract class that doesn't contain any abstract methods. This class is used as a base class for defining subclasses.
- An abstract class can be used as a data type.

`GeometricObject[] objects = new GeometricObject[10];`

# 4 Interfaces

Interfaces are used to specify common behavior for objects of related classes or unrelated classes.

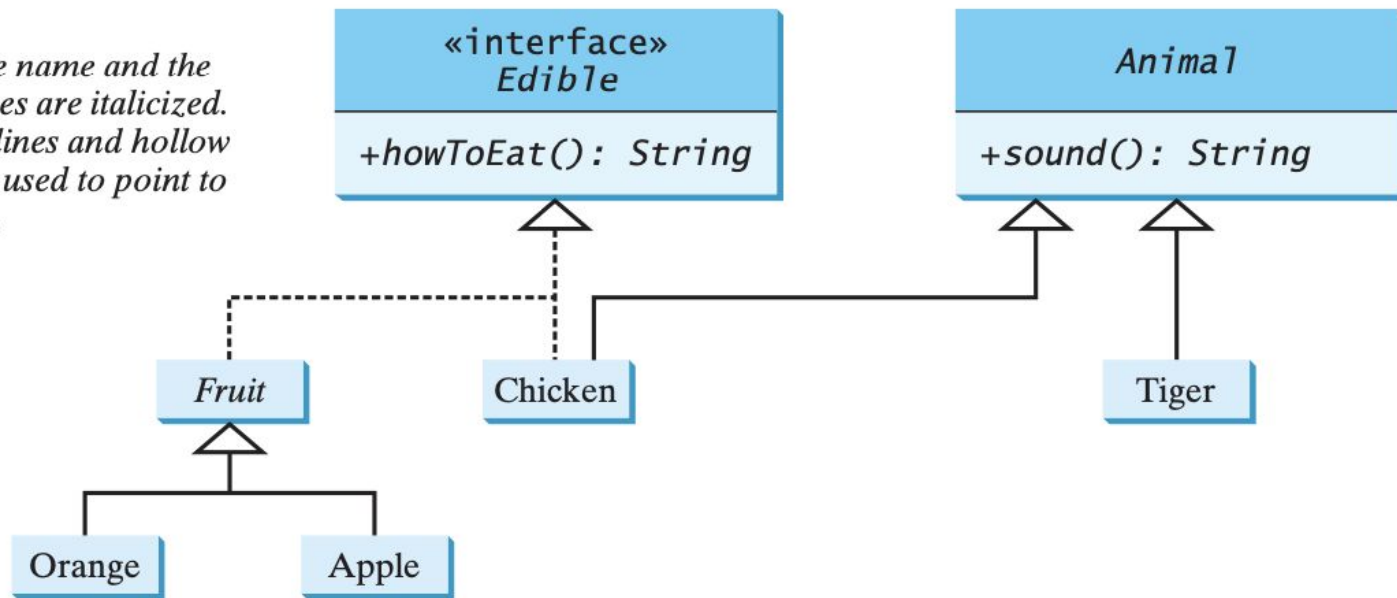
```
modifier interface InterfaceName {  
    /** Constant declarations */  
    /** Abstract method signatures */  
}
```

```
public interface Edible {  
    /** Describe how to eat */  
    public abstract String howToEat();  
}
```

# 4 Interfaces

*Notation:*

*The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.*



**FIGURE 13.4** **Edible** is a supertype for **Chicken** and **Fruit**. **Animal** is a supertype for **Chicken** and **Tiger**. **Fruit** is a supertype for **Orange** and **Apple**.

# 4 Interfaces

```
15 abstract class Animal {
16     /** Return animal sound */
17     public abstract String sound();
18 }
19
20 class Chicken extends Animal implements Edible {
21     @Override
22     public String howToEat() {
23         return "Chicken: Fry it";
24     }
25
26     @Override
27     public String sound() {
28         return "Chicken: cock-a-doodle-doo";
29     }
30 }
31
32 class Tiger extends Animal {
33     @Override
34     public String sound() {
35         return "Tiger: RROOARRR";
36     }
37 }
```

```
38
39 abstract class Fruit implements Edible {
40     // Data fields, constructors, and methods omitted here
41 }
42
43 class Apple extends Fruit {
44     @Override
45     public String howToEat() {
46         return "Apple: Make apple cider";
47     }
48 }
49
50 class Orange extends Fruit {
51     @Override
52     public String howToEat() {
53         return "Orange: Make orange juice";
54     }
55 }
```

# 4 Interfaces

## LISTING 13.7 TestEdible.java

```
1 public class TestEdible {
2     public static void main(String[] args) {
3         Object[] objects = {new Tiger(), new Chicken(), new Apple()};
4         for (int i = 0; i < objects.length; i++) {
5             if (objects[i] instanceof Edible)
6                 System.out.println(((Edible)objects[i]).howToEat());
7
8             if (objects[i] instanceof Animal) {
9                 System.out.println(((Animal)objects[i]).sound());
10            }
11        }
12    }
13 }
```



# 4 Interfaces

Since all data fields are **public static final** and all methods are **public abstract** in an interface, Java allows these modifiers to be omitted.

```
public interface T {  
    public static final int K = 1;  
    public abstract void p();  
}
```

Equivalent

```
public interface T {  
    int K = 1;  
    void p();  
}
```

# 5 The Comparable Interface

The `Comparable` interface defines the `compareTo` method for comparing objects.

// Interface for comparing objects, defined in java.lang

```
package java.lang;
```

```
public interface Comparable<E> {
```

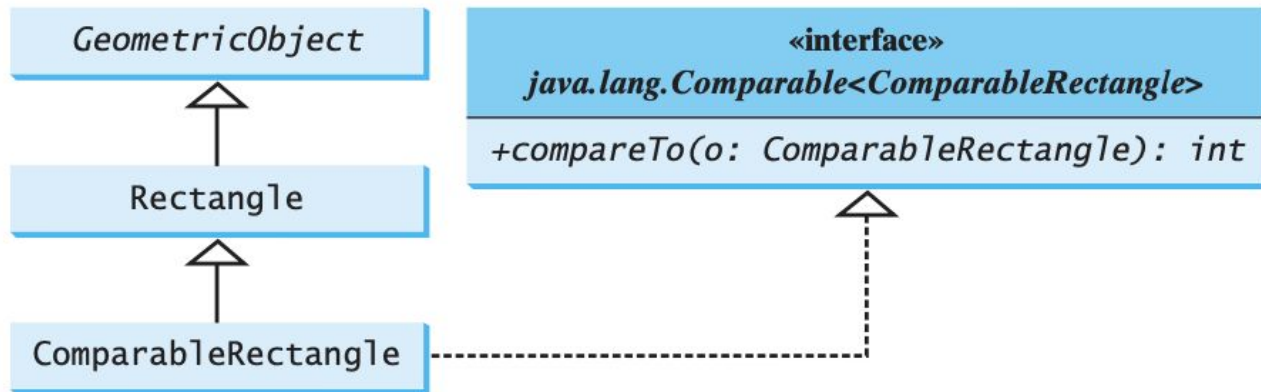
```
    public int compareTo(E o);
```

```
}
```

The `Comparable` interface is a generic interface.

The generic type `E` is replaced by a concrete type when implementing this interface.

# 5 The Comparable Interface



**FIGURE 13.5** *ComparableRectangle* extends *Rectangle* and implements *Comparable*.

# 5 The Comparable Interface

## LISTING 13.8 SortComparableObjects.java

```
1  import java.math.*;
2
3  public class SortComparableObjects {
4      public static void main(String[] args) {
5          String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
6          java.util.Arrays.sort(cities);
7          for (String city: cities)
8              System.out.print(city + " ");
9          System.out.println();
10
11         BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
12             new BigInteger("432232323239292"),
13             new BigInteger("54623239292")};
14         java.util.Arrays.sort(hugeNumbers);
15         for (BigInteger number: hugeNumbers)
16             System.out.print(number + " ");
17     }
18 }
```

# 5 The Comparable Interface

## LISTING 13.9 ComparableRectangle.java

```
1 public class ComparableRectangle extends Rectangle
2     implements Comparable<ComparableRectangle> {
3     /** Construct a ComparableRectangle with specified properties */
4     public ComparableRectangle(double width, double height) {
5         super(width, height);
6     }
7
8     @Override // Implement the compareTo method defined in Comparable
9     public int compareTo(ComparableRectangle o) {
10         if (getArea() > o.getArea())
11             return 1;
12         else if (getArea() < o.getArea())
13             return -1;
14         else
15             return 0;
16     }
17
18     @Override // Implement the toString method in GeometricObject
19     public String toString() {
20         return super.toString() + " Area: " + getArea();
21     }
22 }
```

# 5 The Comparable Interface

## LISTING 13.10 SortRectangles.java

```
1 public class SortRectangles {  
2     public static void main(String[] args) {  
3         ComparableRectangle[] rectangles = {  
4             new ComparableRectangle(3.4, 5.4),  
5             new ComparableRectangle(13.24, 55.4),  
6             new ComparableRectangle(7.4, 35.4),  
7             new ComparableRectangle(1.4, 25.4)};  
8         java.util.Arrays.sort(rectangles);  
9         for (Rectangle rectangle: rectangles) {  
10             System.out.print(rectangle + " ");  
11             System.out.println();  
12         }  
13     }  
14 }
```

# 6 The Comparator Interface

```
class LengthComparator implements Comparator<String> {  
    public int compare(String first, String second) {  
        return first.length() - second.length();  
    }  
}
```

```
public class TestMain {  
    public static void main(String[] args) {  
        String[] friends = { "Peter", "Paul", "Mary" };  
        Arrays.sort(friends); // normal comparison  
        Comparator<String> tmp = new LengthComparator();  
        Arrays.sort(friends, tmp); // length comparison  
        for (String i: friends) System.out.println(i);  
        System.out.println(friends);  
    }  
}
```

# 7 The Cloneable Interface

The Cloneable interface specifies that an object can be cloned.

```
package java.lang;  
public interface Cloneable { //marker interface  
}
```

An interface with an empty body is referred to as a marker interface.

The objects of `Cloneable` can be cloned using the `clone()` method defined in the `Object` class.



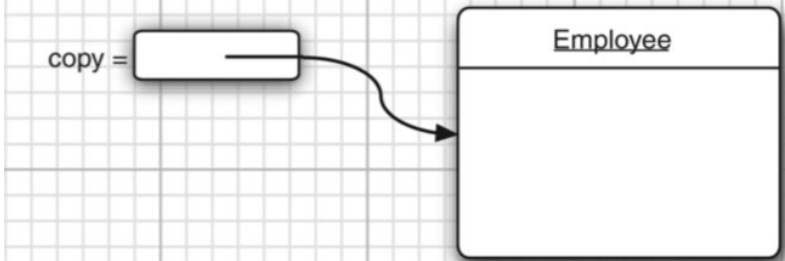
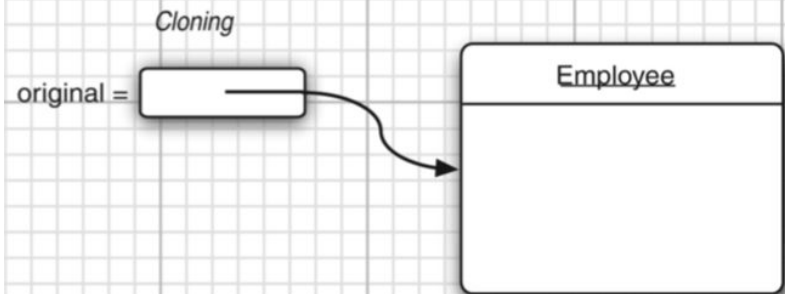
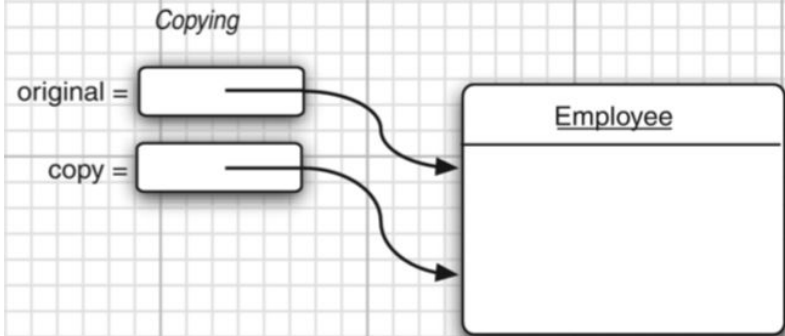
# 7 The Cloneable Interface

## Object Cloning

```
Employee original = new Employee("John Public",  
50000);
```

```
Employee copy = original;  
copy.raiseSalary(10);
```

```
Employee copy = original.clone();  
copy.raiseSalary(10); // OK--original unchanged
```

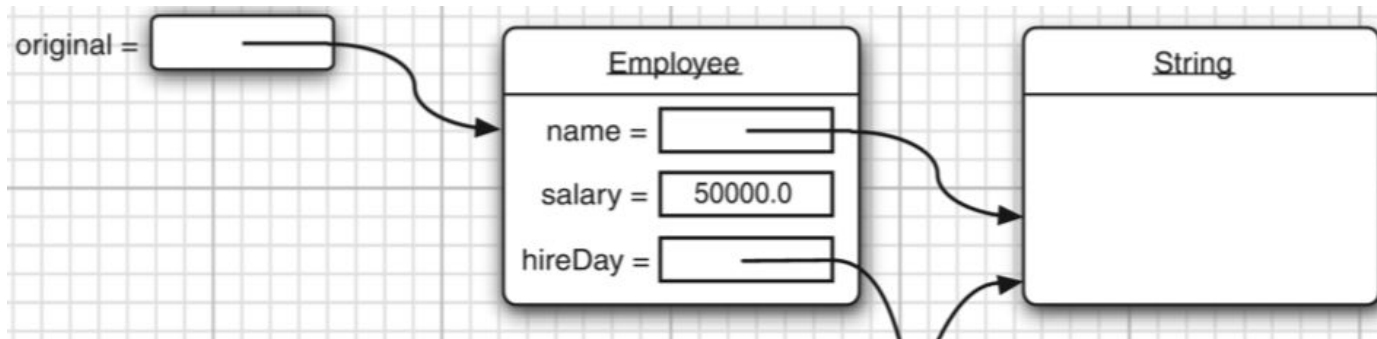


# 7 The Cloneable Interface

## Object Cloning

Make only a field-by-field copy:

- A **shallow copy**: original and cloned variables refer to the same object.
- A deep copy: **redefine clone()** to make a deep copy that clones the sub objects as well.

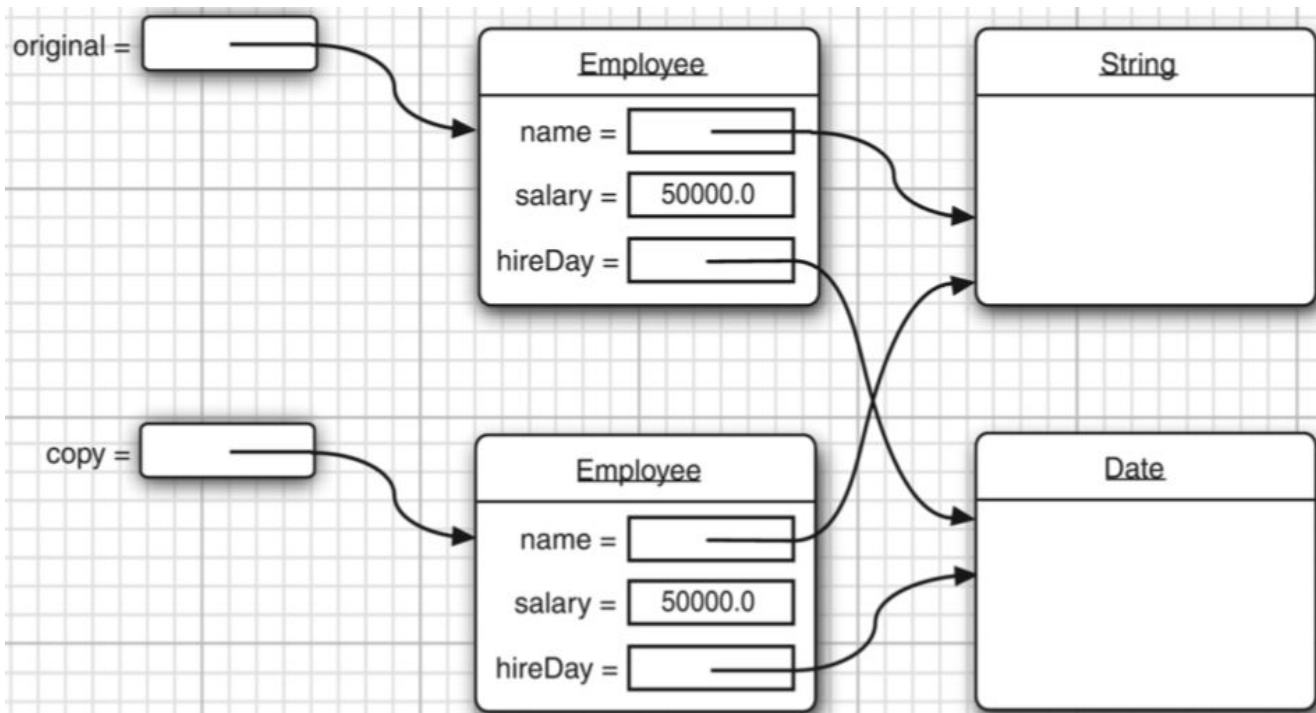


# 7 The Cloneable Interface

## Object Cloning

Make only a field-by-field copy:

- A **shallow copy**: original and cloned variables refer to the same object.
- A **deep copy**: **redefine clone()** to make a deep copy that clones the sub objects as well.



## 7 The Cloneable Interface

```
class Employee implements Cloneable {  
    ...  
    public Employee clone() throws CloneNotSupportedException {  
        // call Object.clone()  
        Employee cloned = (Employee) super.clone();  
  
        // clone mutable fields  
        cloned.hireDay = (Date) hireDay.clone();  
  
        return cloned;  
    }  
}
```

Full code: v1ch06.clone

# 8 Interfaces vs. Abstract Classes

**TABLE 13.2** Interfaces vs. Abstract Classes

	<i>Variables</i>	<i>Constructors</i>	<i>Methods</i>
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be <b>public static final</b> .	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

# 8 Interfaces vs. Abstract Classes

Java allows only single inheritance for class extension but allows multiple extensions for interfaces.

```
public class NewClass extends BaseClass implements Interface1, ..., InterfaceN {  
    ...  
}
```

An interface can inherit other interfaces using the `extends` keyword.

```
public interface NewInterface extends Interface1, ... , InterfaceN {  
    // constants and abstract methods  
}
```

# 9 Class Design Guidelines

## Cohesion

A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.

You can use a class for students but you should not combine students and staff in the same class, because students and staff are different entities.

# 9 Class Design Guidelines

## Consistency

- Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods.
- Make the names consistent, choose the same names for similar operations.
  - For example, the `length()` method returns the size of a `String`, a `StringBuilder`, and a `StringBuffer`.
- Consistently provide a public no-arg constructor for constructing a default instance.



# 9 Class Design Guidelines

## Encapsulation

- A class should use the *private* modifier to hide its data from direct access by clients.
- Provide a getter method only if you want the data field to be readable
- Provide a setter method only if you want the data field to be updateable.

# 9 Class Design Guidelines

## Clarity

- A class should have a clear contract that is easy to explain and easy to understand.
  - independent methods that function independently of their order of occurrence.
  - independent properties: you should not declare a data field that can be derived from other data fields

# 9 Class Design Guidelines

## Completeness

- Classes are designed for use by many different customers.
- A class should provide a variety of ways for customization through properties and methods.
- E.g. String class contains more than 40 methods that are useful for a variety of applications.

# 9 Class Design Guidelines

## Instance vs. Static

- A variable or method that is dependent on a specific instance of the class must be an instance variable or method.
- A variable that is shared by all the instances of a class should be declared static.
- Always reference static variables and methods from a class name to improve readability and avoid errors.

# 9 Class Design Guidelines

## Instance vs. Static

- Do not pass a parameter from a constructor to initialize a static data field. It is better to use a setter method to change the static data field.

```
public class Something {  
    private int t1;  
    private static int t2;  
  
    public Something(int t1, int t2) {  
        ...  
    }  
}
```

(a)

```
public class Something {  
    private int t1;  
    private static int t2;  
  
    public Something(int t1) {  
        ...  
    }  
  
    public static void setT2(int t2) {  
        Something.t2 = t2;  
    }  
}
```

(b)

# 9 Class Design Guidelines

## Inheritance vs. Aggregation

The difference between inheritance and aggregation is the difference between an is-a and a has-a relationship.

# 9 Class Design Guidelines

## Interfaces vs. Abstract Classes

- Both interfaces and abstract classes can be used to specify common behavior for objects.
- A strong is-a relationship that clearly describes a parent–child relationship should be modeled using classes.
  - E.g., since an orange is a fruit, their relationship should be modeled using class inheritance.
- A weak is-a relationship indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces.
  - For example, all strings are comparable, so the String class implements the Comparable interface.

# **Lambda Expressions**

## **Inner class**



# 1 Lambda Expressions

## Why Lambdas?

- A lambda expression is a block of code was passed to someone—a **event**, or a **sort method**. That code block was called at some later time.
- Lambda expressions can be used to simplify coding for event handling.

```
class LengthComparator implements Comparator<String> {  
    public int compare(String first, String second) {  
        return first.length() - second.length();  
    }  
}
```

```
class TimePrinter implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        // do some work  
    }  
}
```

# 1 Lambda Expressions

## The Syntax of Lambda Expressions

**parameters, ->, expression**

```
class LengthComparator implements Comparator<String> {  
    public int compare(String first, String second) {  
        return first.length() - second.length();  
    }  
}
```

```
(String first, String second) -> first.length() - second.length()
```

```
(String first, String second) -> {  
    if (first.length() < second.length())  
        return -1;  
    else if (first.length() > second.length())  
        return 1;  
    else  
        return 0;  
}
```

# 1 Lambda Expressions

## The Syntax of Lambda Expressions

parameters, ->, expression

```
class TimePrinter implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        System.out.println("At the tone, the time is " + new Date());  
        Toolkit.getDefaultToolkit().beep();  
    }  
}  
  
event -> {  
    System.out.println("At the tone, the time is " + new Date());  
    Toolkit.getDefaultToolkit().beep();  
}
```

# 1 Lambda Expressions

## The Syntax of Lambda Expressions

parameters, ->, expression

- No parameters

```
() -> { for (int i = 100; i >= 0; i--) System.out.println(i); }
```

# 1 Lambda Expressions

## Functional Interfaces

Many interfaces in Java encapsulate blocks of code, lambdas are compatible with them.

E.g.,

```
ArrayList: public boolean removeIf(Predicate<? super E> filter){...}
```

```
interface Predicate<T> {  
    boolean test(T t);  
    //....  
}
```

```
public static void main(String[] args) {  
    ArrayList<Integer> list = new ArrayList<Integer>();  
    list.add(4); list.add(5); list.add(null);  
    list.removeIf(e -> e == null);  
    for (Integer i: list){  
        System.out.println(i);  
    }  
}
```

# 1 Lambda Expressions

## Method References

```
Timer t = new Timer(1000, event -> System.out.println(event));
```

```
Timer t = new Timer(1000, System.out::println);
```

```
(x, y) -> Math.pow(x, y)
```

```
Math::pow
```

# 1 Lambda Expressions

## Variable Scope

The body of a lambda expression has the **same scope as a nested block**.

```
Path first = Paths.get("/usr/bin");
```

```
Comparator<String> comp = (first, second) -> first.length() - second.length();
```

**A lambda expression has three ingredients:**

1. Parameters
2. A block of code
3. Free variables (captured variables):
  - effectively **final**: their values are never changed after their initialization;
  - lambda expression cannot change their values as well.

# 1 Lambda Expressions

```
public static void countDown(int start, int delay) {  
    ActionListener listener = event -> {  
        start--;  
        System.out.println(start);  
    };  
    new Timer(delay, listener).start();  
}
```

```
public static void repeat(String text, int count) {  
    for (int i = 1; i <= count; i++) {  
        ActionListener listener = event -> {  
            int j=10;  
            System.out.println(i + ": " + text);  
        };  
        new Timer(1000, listener).start();  
    }  
}
```

1. Parameters
2. A block of code
3. Free variables



# 1 Lambda Expressions

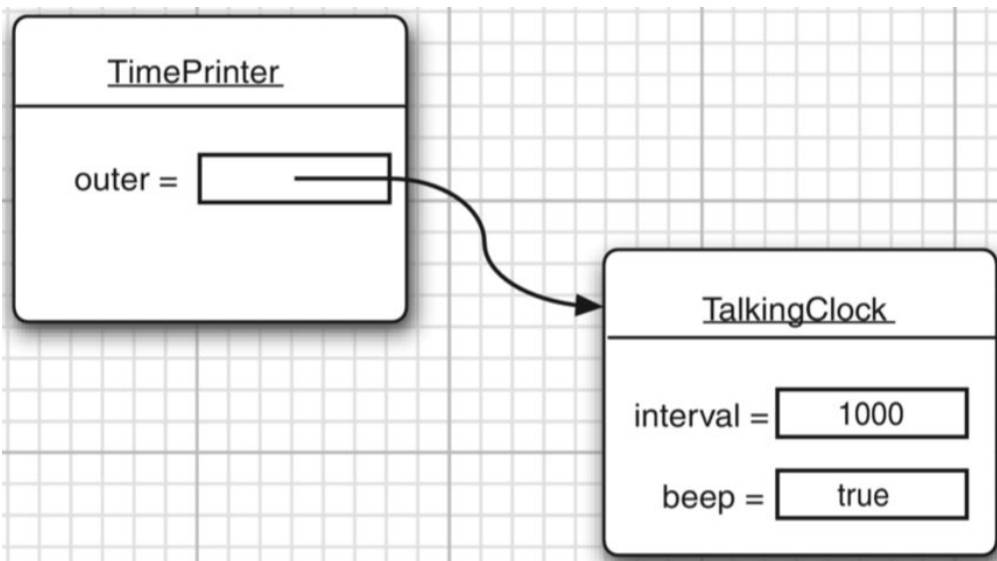
Your own interface along with lambda expressions.

```
// SumCalculator.java
interface SumCalculator {
    int sum(int a, int b);
}
```

```
// Main.java
public class Main {
    public static void main(String[] args) {
        SumCalculator sumCalculator = (x, y) -> x + y;
        int result = sumCalculator.sum(7, 6);
        System.out.println("Sum 7,6): " + result);
        result = sumCalculator.sum(15, -35);
        System.out.println("Sum 15, -35): " + result);
    }
}
```

## 2 Inner Classes

- An inner class is a class that is defined inside another class.
  - An inner class method **access** both its own data fields and those of the outer object creating it.
  - Inner classes can be **hidden** from other classes in the same package.



## 2 Inner Classes

### Use of an Inner Class to Access Object State

```
class TalkingClock{
    private int interval;
    private boolean beep;

    public TalkingClock(int interval, boolean beep){
        this.interval = interval;
        this.beep = beep;
    }

    public void start(){
        ActionListener listener = new TimePrinter();
        // this.new TimePrinter();
        Timer t = new Timer(interval, listener);
        t.start();
    }

    class TimePrinter implements ActionListener{
        public void actionPerformed(ActionEvent event){
            System.out.println("At the tone, the time is " + new Date());
            if (beep) Toolkit.getDefaultToolkit().beep();
            // TalkingClock.this.beep
        }
    }
}
```

## 2 Inner Classes

### Use of an Inner Class to Access Object State

- Refer to an inner class:

```
TalkingClock jabberer = new TalkingClock(1000, true);  
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

# 2 Inner Classes

## Are Inner Classes Useful?

- Inner classes are **a phenomenon of the compiler**, virtual machine knows nothing about them.
- Inner classes have more access privileges (access private field).
- Inner classes are hidden from other class in the same package.

# 2 Inner Classes

## Local Inner Classes

- The class is defined **locally in a single method**.
- Their scope is always restricted to the block in which they are declared.

# 2 Inner Classes

## Accessing Variables from Outer Methods

- Local inner classes can access
  - the fields of their outer classes
  - local variables of outer methods

# 2 Inner Classes

## Local Inner Classes

```
class TalkingClockLocal {  
    private int interval;  
    private boolean beep;  
    public TalkingClockLocal(int interval, boolean beep) {  
        this.interval = interval;  
        this.beep = beep;  
    }  
    public void start() {  
        class TimePrinterLocal implements ActionListener {  
            public void actionPerformed(ActionEvent event) {  
                System.out.println("At the tone, the time is " + new Date());  
                if (beep)  
                    Toolkit.getDefaultToolkit().beep();  
            }  
        }  
        ActionListener listener = new TimePrinterLocal();  
        Timer t = new Timer(interval, listener);  
        t.start();  
    }  
}
```

Code: InnerClassTestLocal



# 2 Inner Classes

## Accessing Variables from Outer Methods

```
class TalkingClockLocalVar {  
    public TalkingClockLocalVar() {  
    }  
    public void start(int interval, boolean beep) {  
        class TimePrinterLocalVar implements ActionListener {  
            public void actionPerformed(ActionEvent event) {  
                System.out.println("At the tone, the time is " + new Date());  
                if (beep) // local variable of outer method  
                    Toolkit.getDefaultToolkit().beep();  
            }  
        }  
        ActionListener listener = new TimePrinterLocalVar();  
        Timer t = new Timer(interval, listener);  
        t.start();  
    }  
}
```

# 2 Inner Classes

## Anonymous Inner Classes

```
new SuperType(construction parameters) {  
    inner class methods and data  
}
```

SuperType: class, interface

# 2 Inner Classes

## Anonymous Inner Classes

```
class TalkingClockLocalAnony {  
    public TalkingClockLocalAnony() {  
  
        public void start(int interval, boolean beep) {  
            ActionListener listener = new ActionListener() {  
                public void actionPerformed(ActionEvent event) {  
                    System.out.println("At the tone, the time is " + new Date());  
                    if (beep)  
                        Toolkit.getDefaultToolkit().beep();  
                }  
            };  
            Timer t = new Timer(interval, listener);  
            t.start();  
        }  
    }  
}
```

# 2 Inner Classes

## Anonymous Inner Classes

- Anonymous inner class
- Lambda expression

```
class TalkingClockLocalAnony {  
    public TalkingClockLocalAnony() {  
        }  
        // start: anonymous inner class  
        public void start(int interval, boolean beep) {  
            ActionListener listener = new ActionListener() {  
                public void actionPerformed(ActionEvent event) {  
                    System.out.println("At the tone, the time is " + new Date());  
                    if (beep)  
                        Toolkit.getDefaultToolkit().beep();  
                }  
            };  
            Timer t = new Timer(interval, listener);  
            t.start();  
        }  
        // start: lambda expression  
        public void start(int interval, boolean beep) {  
            Timer t = new Timer(interval, event -> {  
                System.out.println("At the tone, the time is " + new Date());  
                if (beep)  
                    Toolkit.getDefaultToolkit().beep();  
            });  
            t.start();  
        }  
    }  
}
```