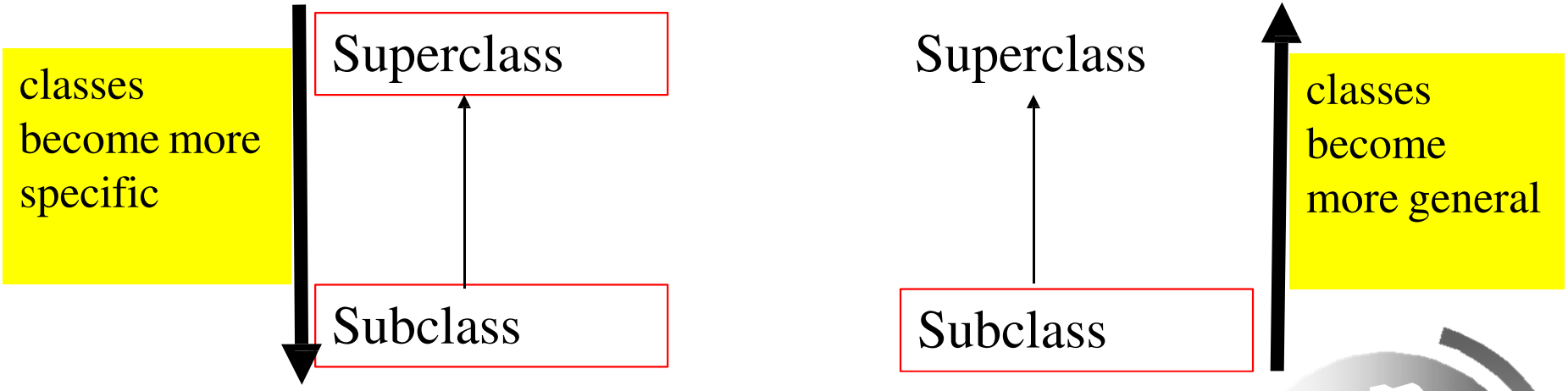
Chapter 13 Abstract Classes and Interfaces

Dr. Asem Kitana
Dr. Abdallah Karakra

Abstract Classes

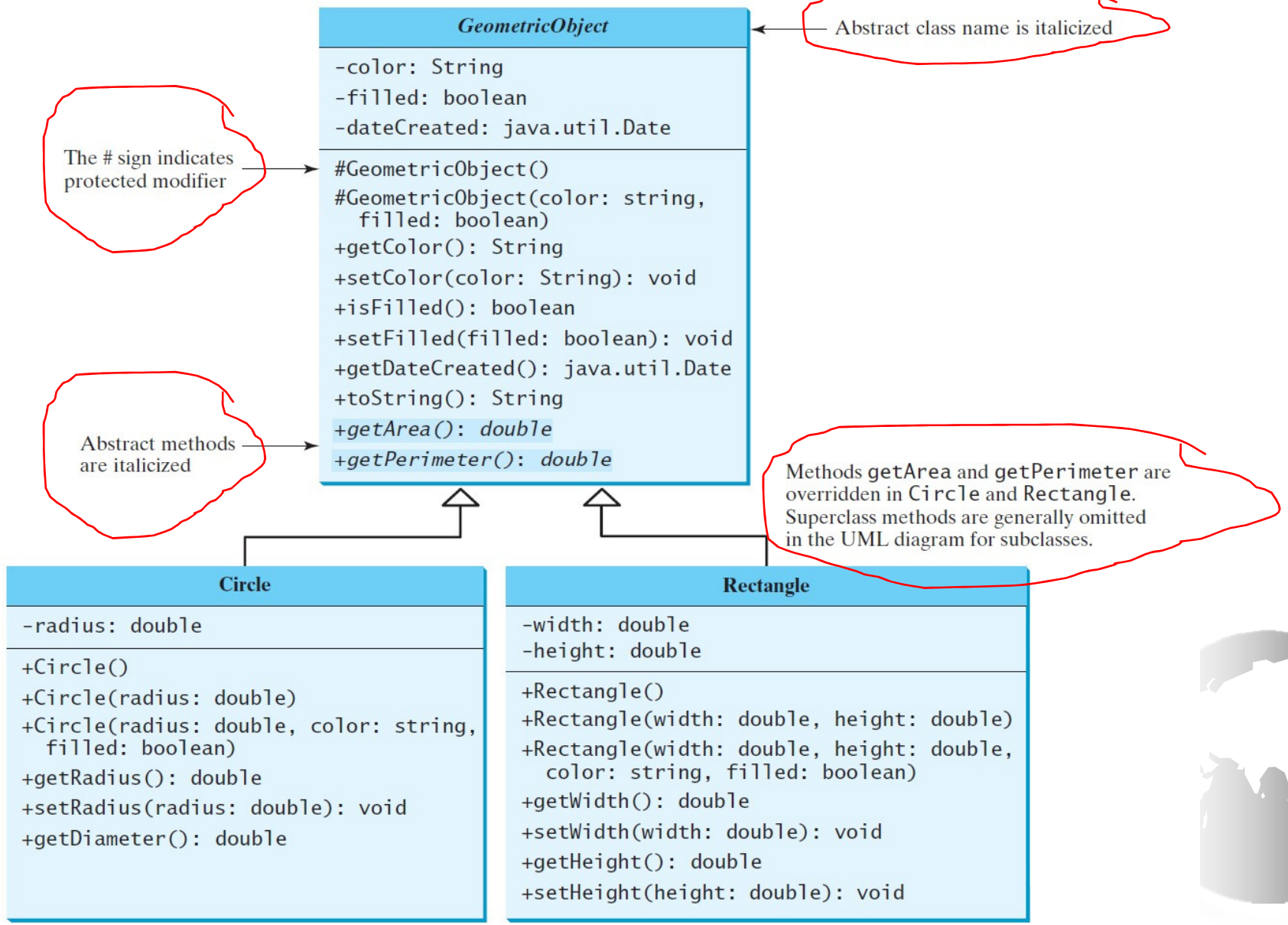
An **abstract class** cannot be used to create objects. An abstract class can contain abstract methods, which are implemented in **concrete** subclasses.



Class design should ensure that a superclass contains common features of its subclasses. Sometimes a superclass is so abstract that it cannot have any specific instances. Such a class is referred to as an abstract class.

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

  /** Construct a default geometric object */
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }

  /** Construct a geometric object with color and filled value */
  protected GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
  }

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

  /** Return filled. Since filled is boolean,
   *  the get method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** Get dateCreated */
  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();
}
```

Abstract Classes and Abstract Methods



abstract method in abstract class

- An abstract method ***cannot be contained*** in a non-abstract class.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract. In other words, in a non-abstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.

object cannot be created from abstract class

An abstract class cannot be instantiated using the **new operator**, but you can still define its constructors, which are invoked in the constructors of its subclasses. For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

object cannot be created from abstract class

```
GeometricObject geoObject1 = new Circle(5);
GeometricObject geoObject2 = new Rectangle(5, 3);
```

abstract class without abstract method

- A class that contains abstract methods must be abstract.
- ***It is possible*** to define an abstract class that contains no abstract methods. In this case, you cannot create instances of the class using **the new operator**. This class is used as a base class for defining a new subclass.

superclass of abstract class may be concrete

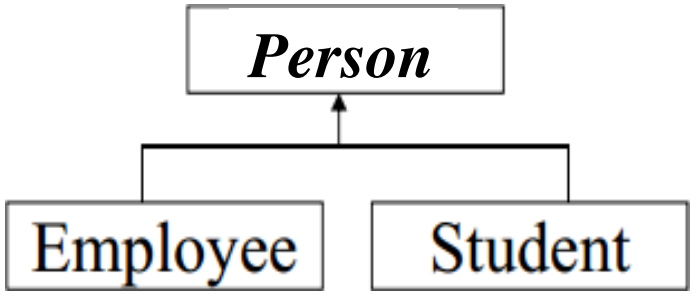
A subclass can be abstract even if its superclass is concrete. For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

A concrete class is any normal class in a Java program. This class will not have any abstract methods.

concrete method overridden to be abstract

A subclass can override a method from its superclass to define it abstract. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined abstract.

```
abstract class Person {
  protected String name;
  . . .
  public abstract String getDescription() ;
  . . .
}
Class Student extends Person {
  private String major;
  . . .
  public String getDescription() {
        return name + " a student major in " + major;
  }
}
Class Employee extends Person {
  private float salary;
  . . .
  public String getDescription() {
        return name + " an employee with a salary of $ " + salary;
  }
}
```



Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?

What is an interface? Why is an interface useful?

An interface is a classlike construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects. For example, you can specify that the objects are **comparable, edible, cloneable** using appropriate interfaces.

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {
  constant declarations;
  abstract method signatures;
}
```

Example:

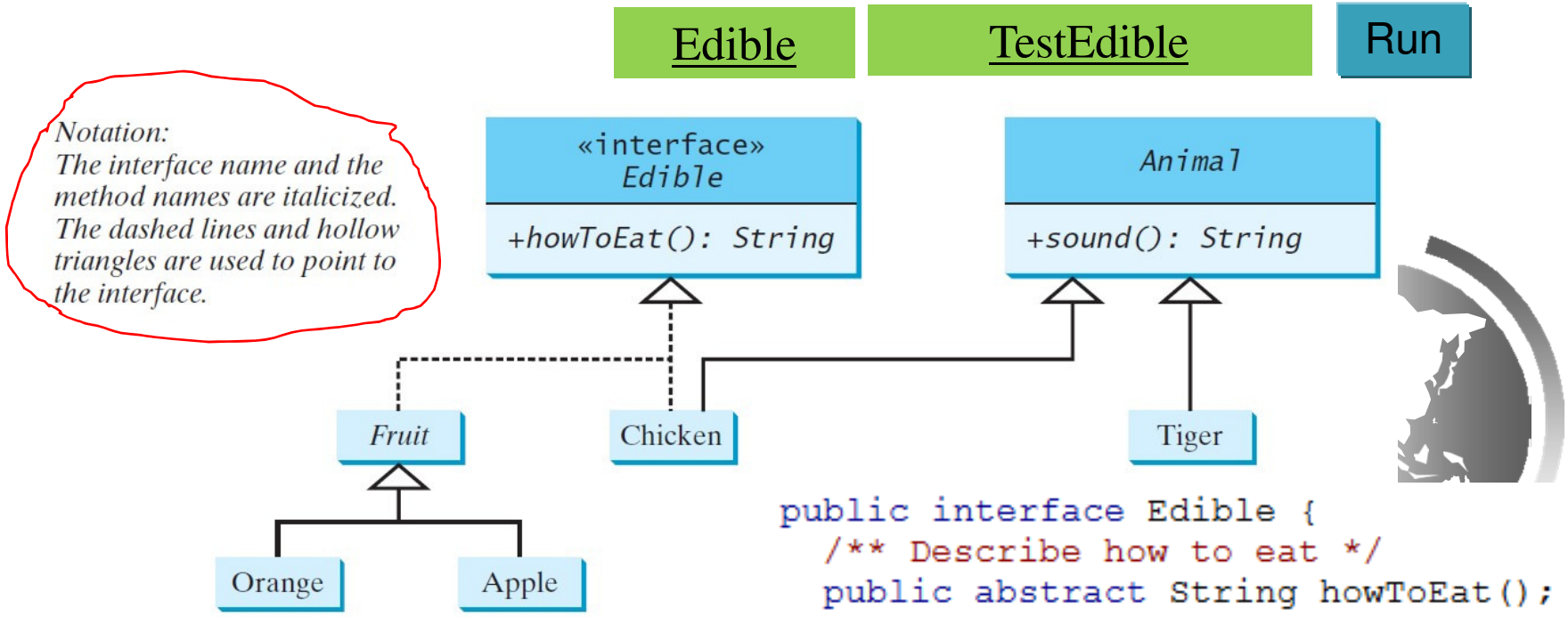
```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Interface is a Special Class

An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you **cannot create an instance from an interface using the new operator**, but in most cases you can use an interface more or less the same way you use an abstract class. For example, **you can use an interface as *a data type* f**or a variable, as the result of casting, and so on.

Example

You can now use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface (See TestEdible).



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

```java
abstract class Animal {
  private double weight;

  public double getWeight() {
    return weight;
  }

  public void setWeight(double weight) {
    this.weight = weight;
  }

  /** Return animal sound */
  public abstract String sound();
}
```

```java
class Chicken extends Animal implements Edible {
  @Override
  public String howToEat() {
    return "Chicken: Fry it";
  }

  @Override
  public String sound() {
    return "Chicken: cock-a-doodle-doo";
  }
}
```

```java
class Tiger extends Animal {
  @Override
  public String sound() {
    return "Tiger: RROOAARR";
  }
}
```

```
abstract class Fruit implements Edible {
  // Data fields, constructors, and methods omitted here
}
```

```
class Orange extends Fruit {
  @Override
  public String howToEat() {
    return "Orange: Make orange juice";
  }
}
```

```
class Apple extends Fruit {
  @Override
  public String howToEat() {
    return "Apple: Make apple cider";
  }
}
```

```
Tiger: RROOAARR
Chicken: Fry it
Chicken: cock-a-doodle-doo
Apple: Make apple cider
```

Omitting Modifiers in Interfaces

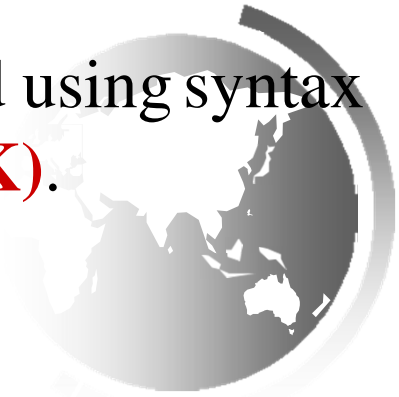
All data fields are ***public final static*** and all methods are ***public abstract*** **in an interface**. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax **InterfaceName.CONSTANT_NAME (e.g., T1.K)**.



Example: The Comparable Interface

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable<E> {
  public int compareTo(E o);
}
```

The compareTo Method

Since all the numeric wrapper classes and the Character class implement the Comparable interface, **the compareTo method is implemented in these classes**.

These classes are defined as follows in the Java API:

Integer and BigInteger Classes

```
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

String and Date Classes

```
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

The compareTo Method

Thus, numbers are comparable, strings are comparable, and so are dates. You can use the **compareTo** method to compare two numbers, two strings, and two dates.

The **compareTo** method determines the order of this object with the specified object **o** and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than **o**.

Example

```
1  System.out.println(new Integer(3).compareTo(new Integer(5)));
   // -1
2  System.out.println("ABC".compareTo("ABE")); // -2
3  java.util.Date date1 = new java.util.Date(2013, 1, 1);
4  java.util.Date date2 = new java.util.Date(2012, 1, 1);
5  System.out.println(date1.compareTo(date2)); // 1
```

Example

Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object. All the following expressions are **true**.

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

Generic `sort` Method

Since all **Comparable** objects have the **compareTo** method, the **java.util.Arrays .sort(Object[])** method in the Java API uses the **compareTo** method to compare and sorts the objects in an array, provided that the objects are instances of the **Comparable** interface.

Generic `sort` Method

```
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 (int i=0; i<cities.length;i++)
8  { System.out.print(cities[i] + " "); }
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 (int i=0; i<hugeNumbers.length;i++)
16 { System.out.print(hugeNumbers[i] + " "); }
17 }
18 }
```

Problems @ Javadoc Declaration Console

<terminated> SortComparableObjects [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (Jan 14, 2023, 9:09:15 PM

```
Atlanta Boston Savannah Tampa
54623239292 432232323239292 2323231092923992
```

Comparable Rectangle

You cannot use the **sort** method to sort an array of **Rectangle** objects, because **Rectangle** does not implement **Comparable**. However, you can define a new rectangle class that implements **Comparable**. The instances of this new class are comparable.

Comparable Rectangle

```java
package comparableRectangleObject;
public class Rectangle {
    double width;
    double height;

    public Rectangle() {

    }

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

    public double getArea() {
            return width * height;
    }
}
```

```java
package comparableRectangleObject;
public class ComparableRectangle extends Rectangle
implements Comparable<ComparableRectangle> {
/** Construct a ComparableRectangle with specified properties */
 public ComparableRectangle(double width, double height) {
 super(width, height);
 }

 @Override // Implement the compareTo method defined in Comparable
 public int compareTo(ComparableRectangle o) {
 if (getArea() > o.getArea())
 return 1;
else if (getArea() < o.getArea())
 return -1;
 else
 return 0;
}
 public String toString() {
 return "Width " + width + " Height " + height + " Area: " + getArea();
 }
}
```

```java
package comparableRectangleObject;
public class SortRectangles {
 public static void main(String[] args) {
ComparableRectangle[] rectangles = {
 new ComparableRectangle(3.4, 5.4),
 new ComparableRectangle(13.24, 55.4),
 new ComparableRectangle(7.4, 35.4),
 new ComparableRectangle(1.4, 25.4)};
 java.util.Arrays.sort(rectangles);
 for (int i=0; i<rectangles.length;i++)
 { System.out.println(rectangles[i]); }
 System.out.println();
}
}

```

Problems Javadoc Declaration Console

<terminated> SortRectangles [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.

```
Width 3.4 Height 5.4 Area: 18.36
Width 1.4 Height 25.4 Area: 35.559999999999995
Width 7.4 Height 35.4 Area: 261.96
Width 13.24 Height 55.4 Area: 733.496
```

Comparable Rectangle

ComparableRectangle is also an instance of **Rectangle**, **GeometricObject**, **Object**, and **Comparable**.

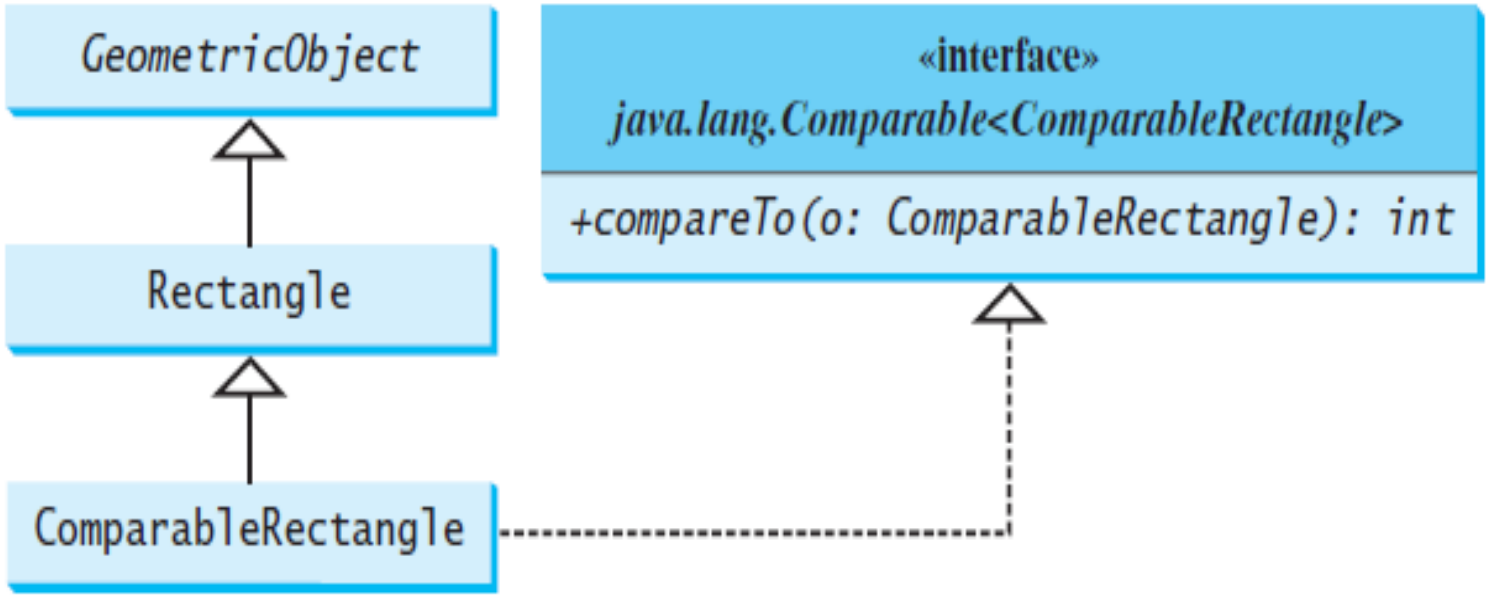


FIGURE 13.5 ComparableRectangle extends Rectangle and implements Comparable.

The `Cloneable` Interface

The **Cloneable** *interface specifies that an object can be cloned. (*used to create a copy of an object)

The `Cloneable` Interface

Marker Interface: An empty interface.

A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable properties. A class that implements the Cloneable interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

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

Example

Many classes (e.g., Date and ArrayList) in the Java library implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code

```
1  ArrayList<Double> list1 = new ArrayList<>();
2  list1.add(1.5);
3  list1.add(2.5);
4  list1.add(3.5);
5  ArrayList<Double> list2 = (ArrayList<Double>)list1.clone();
6  ArrayList<Double> list3 = list1;
7  list2.add(4.5);
8  list3.remove(1.5);
9  System.out.println("list1 is " + list1);
10 System.out.println("list2 is " + list2);
11 System.out.println("list3 is " + list3);
```

displays

```
list1 is [2.5, 3.5]
list2 is [1.5, 2.5, 3.5, 4.5]
list3 is [2.5, 3.5]
```

Example

```
1  int[] list1 = {1, 2};
2  int[] list2 = list1.clone();
3  list1[0] = 7;
4  list2[1] = 8;
5  System.out.println("list1 is " + list1[0] + ", " + list1[1]);
6  System.out.println("list2 is " + list2[0] + ", " + list2[1]);
```

displays

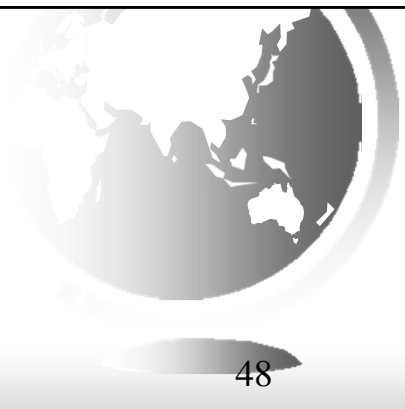
```
list1 is 7, 2
list2 is 1, 8
```

Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

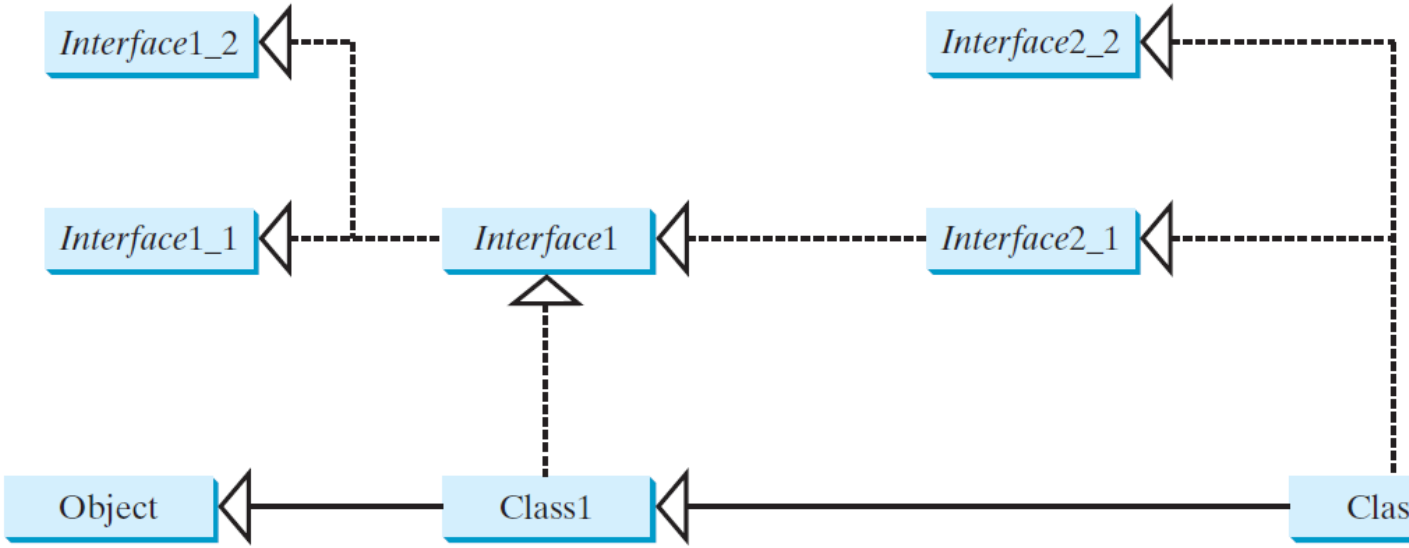
Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

	Variables	*Constructors*	*Methods*
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 `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods



Interfaces vs. Abstract Classes, cont.

All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.



Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Whether to use an interface or a class?

- Abstract classes and interfaces can both be used to model common features.

How do you decide whether to use an interface or a class?

- In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes.

  For example, a staff member is a person.

Whether to use an interface or a class?

- A weak is-a relationship, also known as an is-kind-of 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.
- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired.
- In the case of multiple inheritance, you have to design one as a superclass, and others as interface.