School of Science & Engineering
Department of CSE
Canadian University of Bangladesh

Lecture-13: **Abstract Classes & Interfaces**

Prof. Miftahur Rahman, Ph.D.
Semester: Summer 2024

Object-Oriented Problem Solving

Abstract Classes & Interfaces

Based on Chapter 13 of "Introduction to Java Programming" by Y. Daniel Liang.

Abstract Classes

An **abstract class** is a class that cannot be instantiated on its own and is meant to be subclassed. It can have both abstract methods (without a body) and concrete methods (with a body). Abstract classes are used to provide a base for subclasses to build upon.

Key Points:

- Cannot be instantiated.
- Can have both abstract and concrete methods.
- Can have instance variables.
- Can have constructors.

Example of an Abstract Class

```java
// Abstract class
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();

    // Concrete method
    void sleep() {
        System.out.println("This animal sleeps.");
    }
}

// Subclass (inherits from Animal)
class Dog extends Animal {
    // Providing implementation of abstract method
    void sound() {
        System.out.println("Woof!");
    }
}
```

```java
class Cat extends Animal {
    // Providing implementation of abstract method
    void sound() {
        System.out.println("Meow!");
    }
}
public class MainAbstract {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Creating a Dog object
        Animal myCat = new Cat(); // Creating a Cat object

        myDog.sound(); // Output: Woof!
        myDog.sleep(); // Output: This animal sleeps.

        myCat.sound(); // Output: Meow!
        myCat.sleep(); // Output: This animal sleeps.
    }
}
```

Interfaces

An **interface** is a reference type in Java that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields and cannot provide any method implementations (except for default methods).

Key Points:

- Cannot be instantiated.
- Can only have abstract methods (until Java 8, after which default methods were introduced).
- All methods are implicitly public and abstract.
- Supports multiple inheritance (a class can implement multiple interfaces).

```java
// Interface
interface Animal {
    // Abstract method
    void sound();
    void eat();
}
// Dog class implements Animal interface
class Dog implements Animal {
    public void sound() {
        System.out.println("Woof!");
    }

    public void eat() {
        System.out.println("Dog eats food.");
    }
}
```

```
// Cat class implements Animal interface
class Cat implements Animal {
    public void sound() {
        System.out.println("Meow!");
    }

    public void eat() {
        System.out.println("Cat eats food.");
    }
}
public class MainInterface {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Creating a Dog object
        Animal myCat = new Cat(); // Creating a Cat object

        myDog.sound(); // Output: Woof!
        myDog.eat();   // Output: Dog eats food.

        myCat.sound(); // Output: Meow!
        myCat.eat();   // Output: Cat eats food.
    }
}
```

Outline

- Abstract Classes (13.2)
- Case Study: The Abstract Number Class (13.3)
- Interfaces (13.5)
- The Comparable Interface (13.6)
- Interfaces vs. Abstract Classes (13.8)
- Class Design Guidelines (13.10)

Introduction

- In the inheritance hierarchy, classes become more specific and concrete with each new subclass.
- If you move from a subclass back up to a superclass, the classes become more general and less specific.
- Class design should ensure that a superclass contains common features of its subclasses.

What are Abstract Methods?

- In the example of the previous section, *GeometricObject* was defined as the superclass for Circle and Rectangle.
- Bot Circle and Rectangle contain *getArea()* and *getPerimeter()* methods.
- It is better to define the *getArea()* and *getPerimeter()* methods in the *GeometricObject* class.
- However, these methods cannot be implemented in the *GeometricObject* class, because their implementation depends on the specific type of a geometric object.

What are Abstract Methods? (Cont.)

- Such methods can be defined in the superclass as *abstract methods*.
  - An abstract method is defined without an implementation in the superclass.
    - Its implementation is provided by the subclasses.
  - Abstract methods are denoted using the *abstract* modifier in the method header.

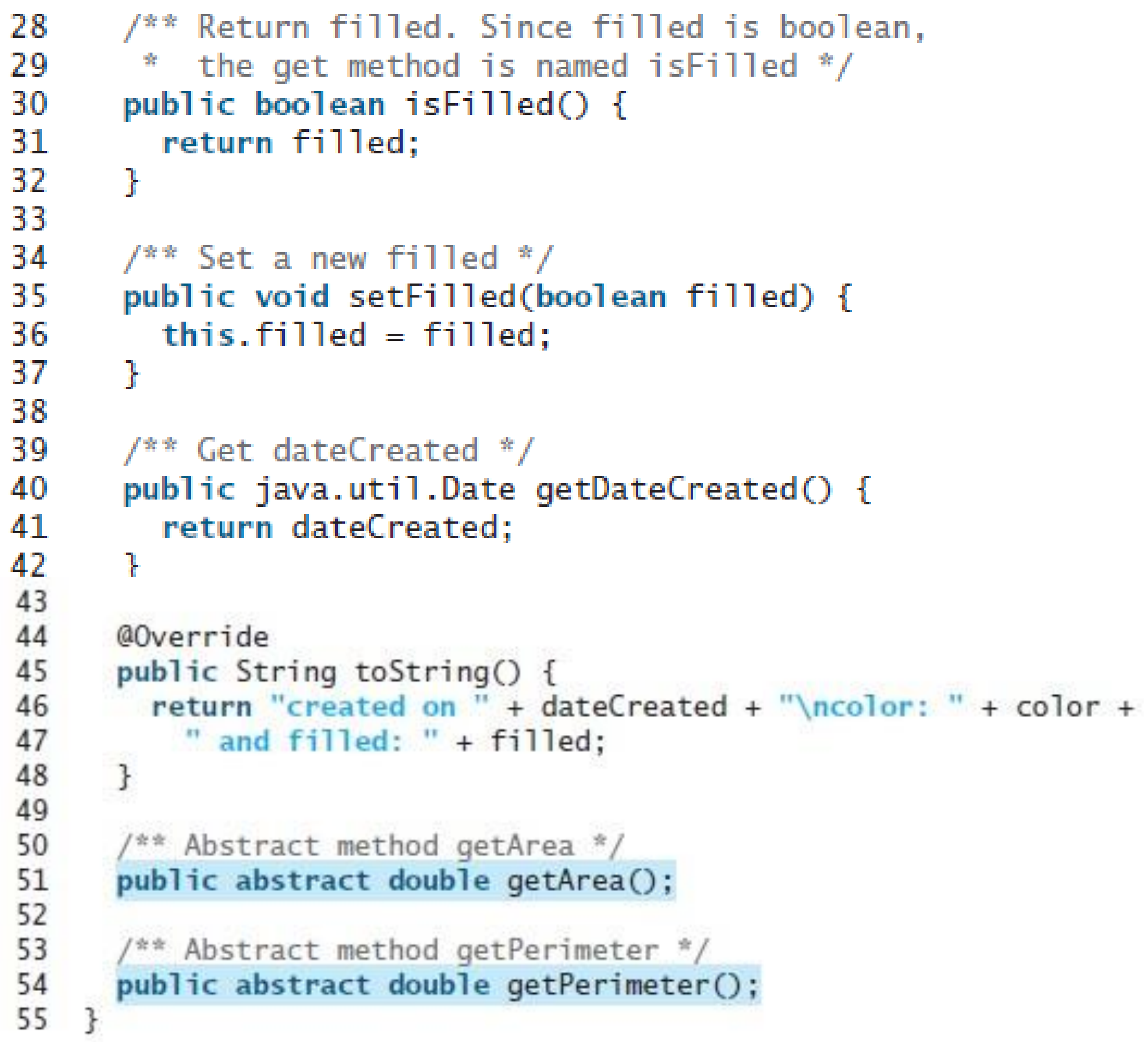
What are Abstract Classes?

- A class that contains at least one abstract method must be defined as an *abstract class*.
  - Abstract classes are denoted using the *abstract* modifier in the class header.
- Abstract classes are like regular classes, but you cannot create instances of abstract classes using the *new* operator.
  - Constructors of an abstract class are defined as protected, because they are used only by subclasses.

Abstract Classes & Methods: Example

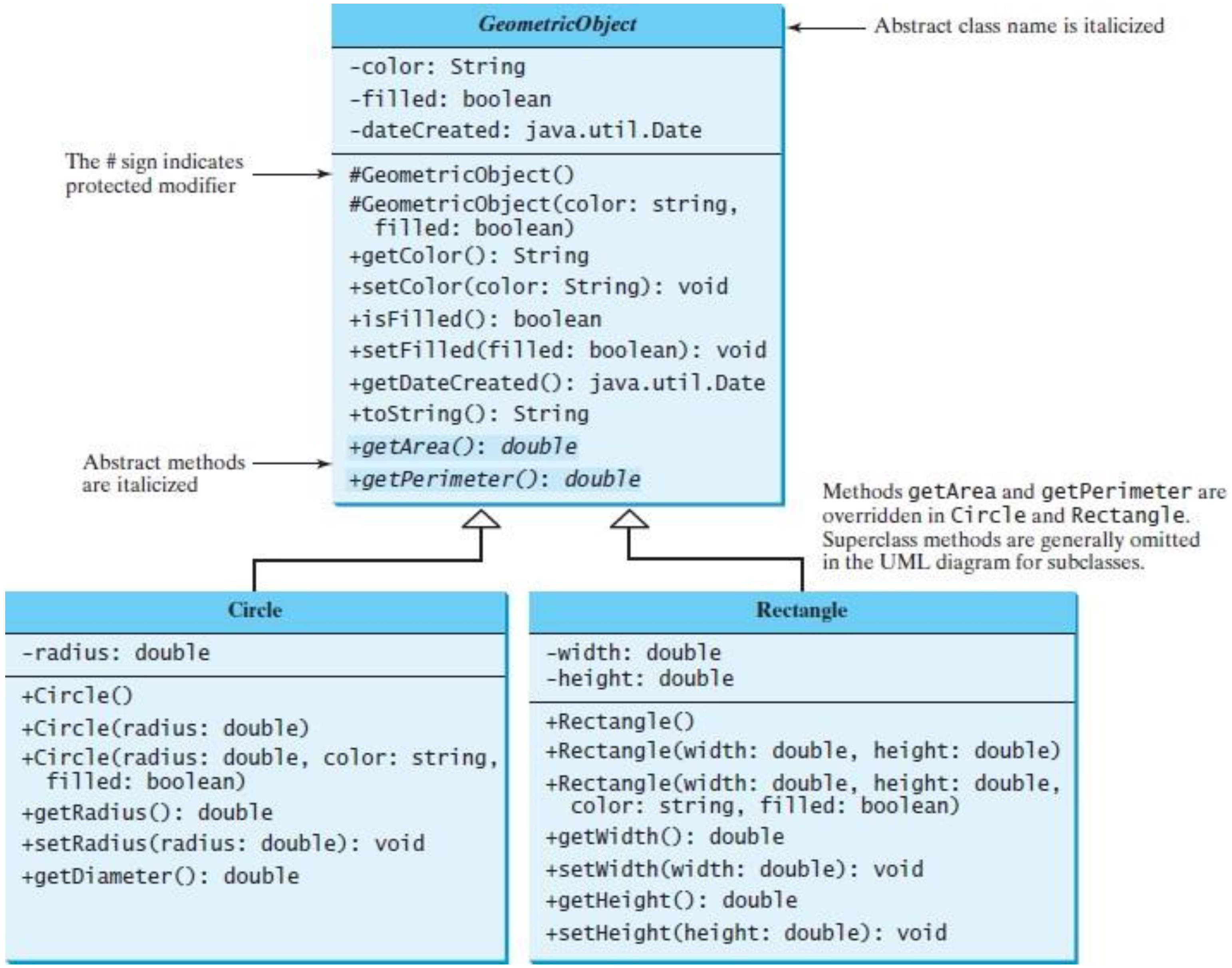
```
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** 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;
  }

```

Abstract Classes & Methods: Example (Cont.)

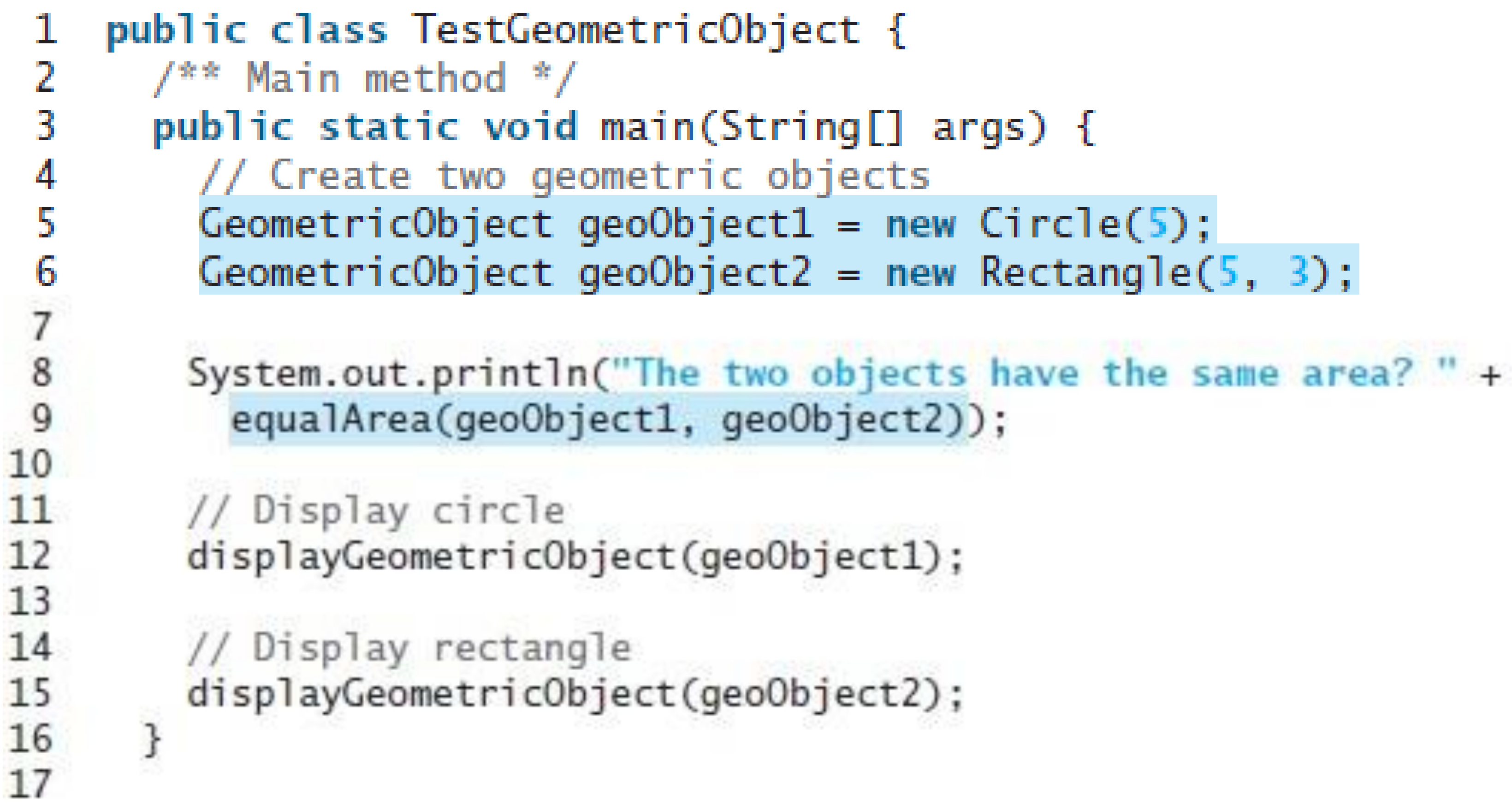


```java
  /** Return filled. Since filled is boolean,
   *  the get method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** 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 Methods: UML Diagram

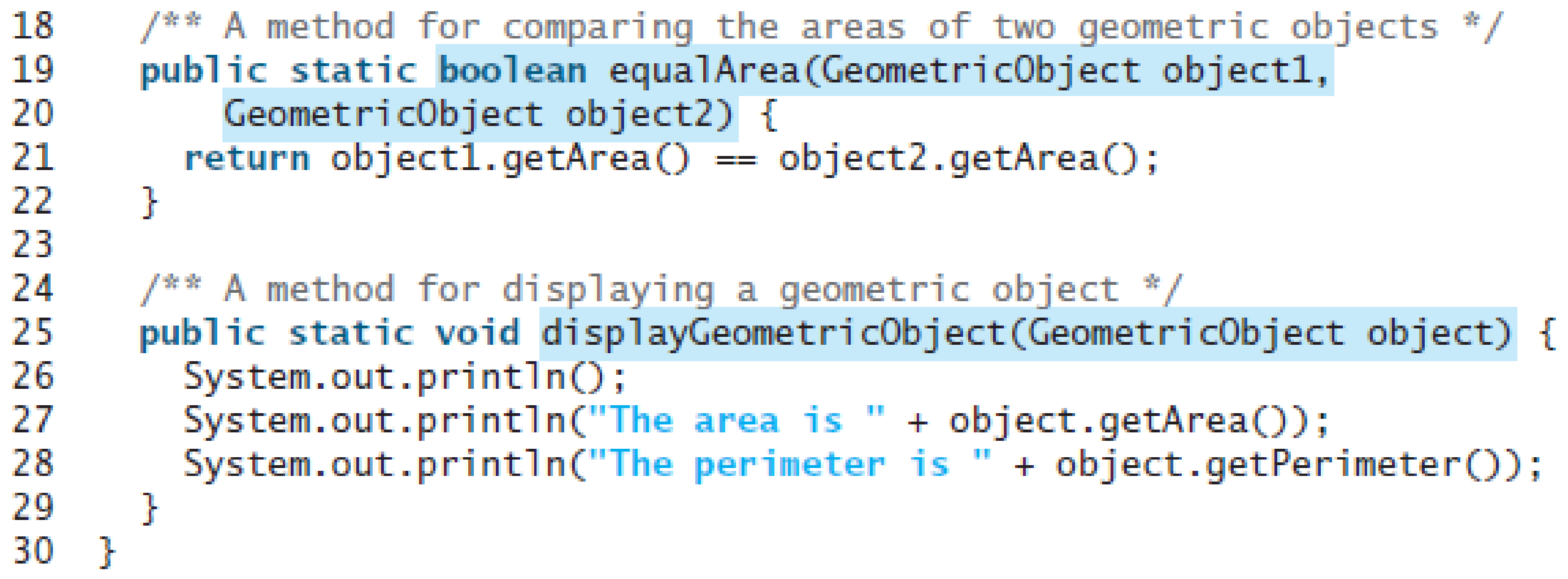

GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
#GeometricObject()
#GeometricObject(color: string, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double
Abstract class name is italicized
The # sign indicates protected modifier
Abstract methods are italicized
Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double, color: string, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

Why Abstract Methods? Example



```java
public class TestGeometricObject {
  /** Main method */
  public static void main(String[] args) {
    // Create two geometric objects
    GeometricObject geoObject1 = new Circle(5);
    GeometricObject geoObject2 = new Rectangle(5, 3);

    System.out.println("The two objects have the same area? " +
      equalArea(geoObject1, geoObject2));

    // Display circle
    displayGeometricObject(geoObject1);

    // Display rectangle
    displayGeometricObject(geoObject2);
  }

```

Why Abstract Methods? Example



```java
  /** A method for comparing the areas of two geometric objects */
  public static boolean equalArea(GeometricObject object1,
      GeometricObject object2) {
    return object1.getArea() == object2.getArea();
  }

  /** A method for displaying a geometric object */
  public static void displayGeometricObject(GeometricObject object) {
    System.out.println();
    System.out.println("The area is " + object.getArea());
    System.out.println("The perimeter is " + object.getPerimeter());
  }
}
```

Important Notes Regarding Abstract Classes and Methods (1)

- An abstract method cannot be contained in a non-abstract class.
  - If a subclass of an abstract superclass does not implement all abstract methods, the subclass must be defined as abstract.
  - Abstract methods are non-static.
- An abstract class cannot be instantiated using the new operator, but:
  - You still can define its constructors which are invoked in the constructors of its subclasses.
  - An abstract class can be used as a data type.
    - Therefore, the following statement, which creates an array whose elements are of the GeometricObject type is correct:

GeometricObject [] Objects = new GeometricObject[10];

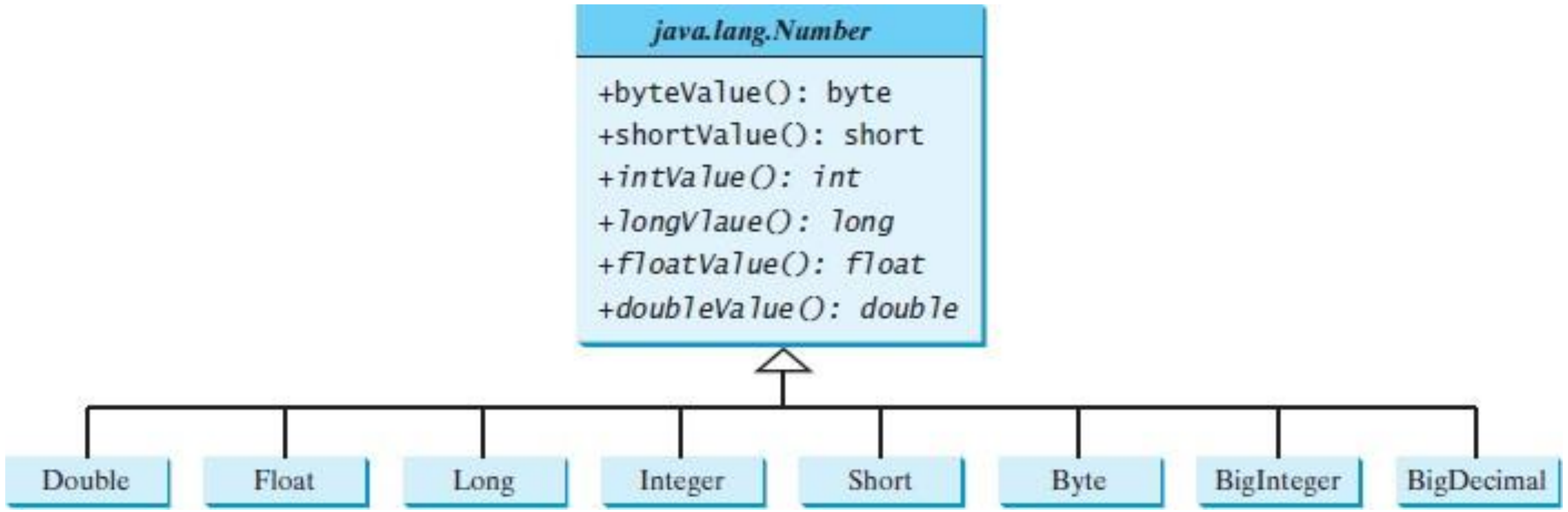
Important Notes Regarding Abstract Classes and Methods (2)

- A class that contains abstract methods must be abstract.
  - However, it is possible to define an abstract class that does not contain any 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 subclasses.
- A subclass can be abstract even if it’s superclass is concrete.
  - For example, the *Object* class is concrete, but its subclasses may be abstract, such as *GeometricObject* in the previous example.

Important Notes Regarding Abstract Classes and Methods (3)

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

Case Study: The Abstract Number Class



Case Study: The Abstract Number Class

LISTING 13.5 LargestNumbers.java

```java
import java.util.ArrayList;
import java.math.*;

public class LargestNumbers {
  public static void main(String[] args) {
    ArrayList<Number> list = new ArrayList<>();
    list.add(45); // Add an integer
    list.add(3445.53); // Add a double
    // Add a BigInteger
    list.add(new BigInteger("3432323234344343101"));
    // Add a BigDecimal
    list.add(new BigDecimal("2.0909090989091343433344343"));

    System.out.println("The largest number is " +
      getLargestNumber(list));
  }

  public static Number getLargestNumber(ArrayList<Number> list) {
    if (list == null || list.size() == 0)
      return null;

    Number number = list.get(0);
    for (int i = 1; i < list.size(); i++)
      if (number.doubleValue() < list.get(i).doubleValue())
        number = list.get(i);

    return number;
  }
}
```

Interfaces

- An *interface* is a class-like construct that contains only constants and abstract methods.
- The intent of interfaces is to define common behavior for related or unrelated classes.
- An interface is treated like a special class in Java.
  - Each interface is compiled into a separate bytecode file.
  - You can use an interface more or less the same way you use an abstract class.
    - An interface can be used as a data type for a reference variable.
    - You cannot create an instance from an interface with the new operator.

Interfaces (Cont.)

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

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

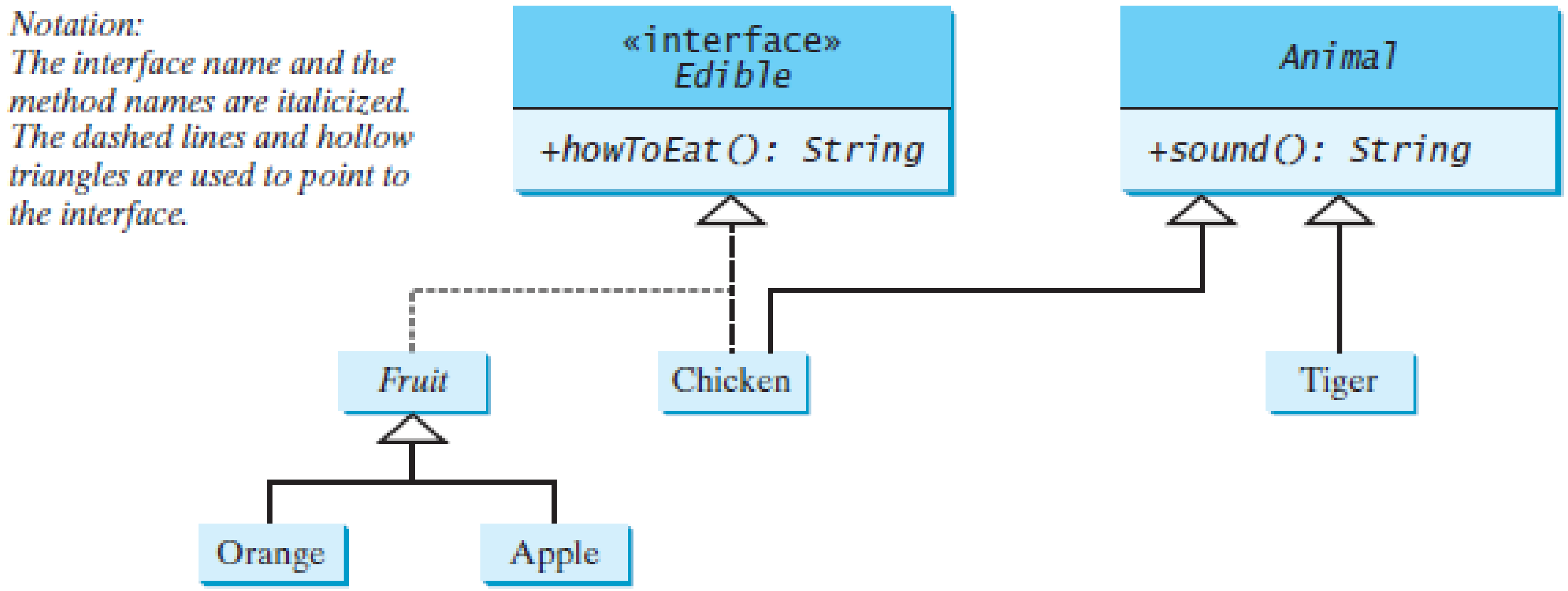
- Example:

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

Implementing an Interface

- You can use an interface to specify the behavior of an object, by letting the class for the object *implement* this interface using the *implements* keyword.
  - When a class implements an interface, it implements all the methods defined in the interface with the exact signature and return type.
- The relationship between an interface and a class that implements it is known as *interface inheritance*.
  - Since *interface inheritance* and *class inheritance* are essentially the same, both are referred to as *inheritance*.

Implementing an Interface (Example)

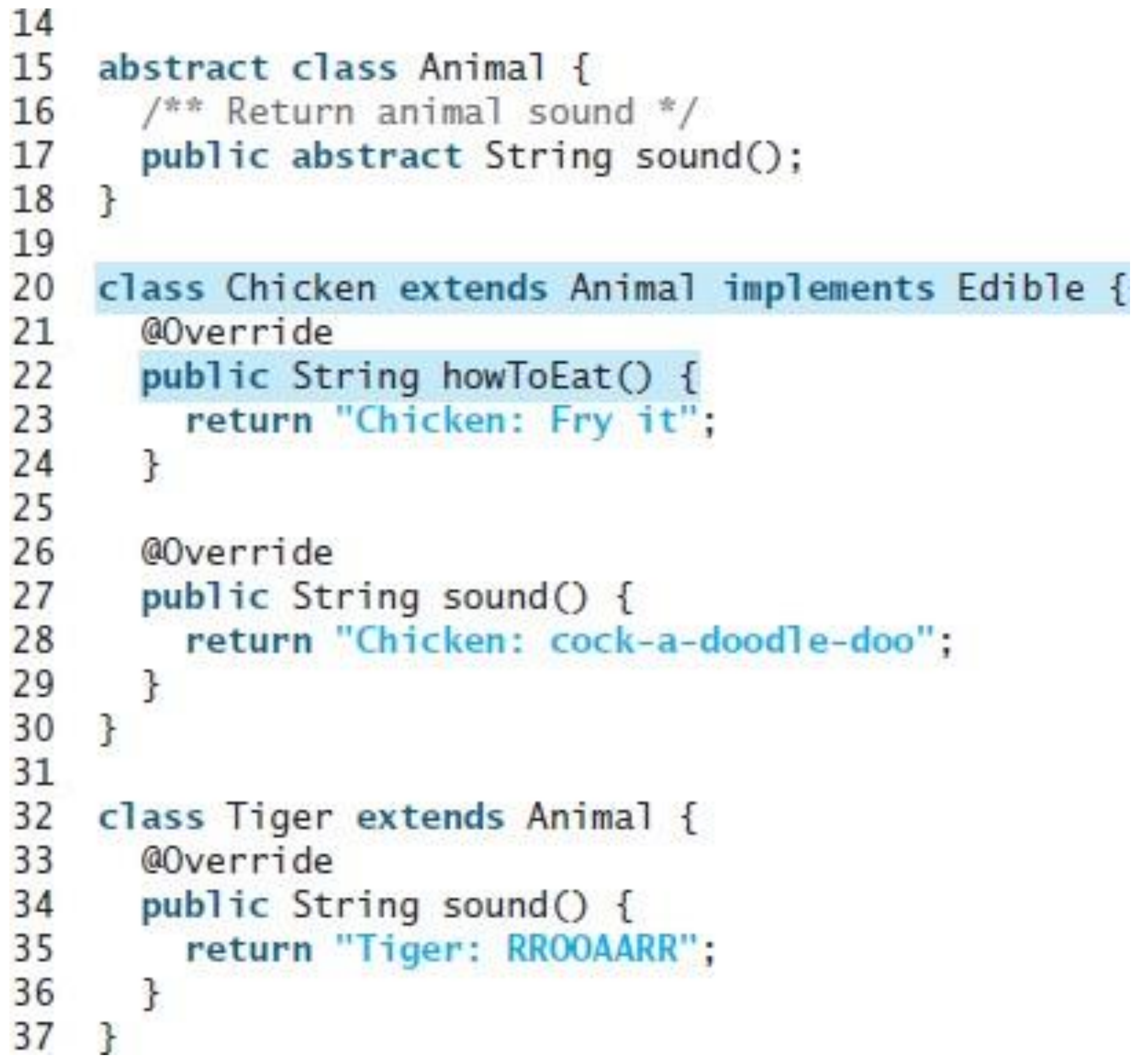

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.
«interface»
Edible
+howToEat(): String
Animal
+sound(): String
Fruit
Chicken
Tiger
Orange
Apple

Implementing an Interface (Example)

LISTING 13.7 TestEdible.java

```java
public class TestEdible {
  public static void main(String[] args) {
    Object[] objects = {new Tiger(), new Chicken(), new Apple()};
    for (int i = 0; i < objects.length; i++) {
      if (objects[i] instanceof Edible)
        System.out.println(((Edible)objects[i]).howToEat());

      if (objects[i] instanceof Animal) {
        System.out.println(((Animal)objects[i]).sound());
      }
    }
  }
}
```

Implementing an Interface (Example)



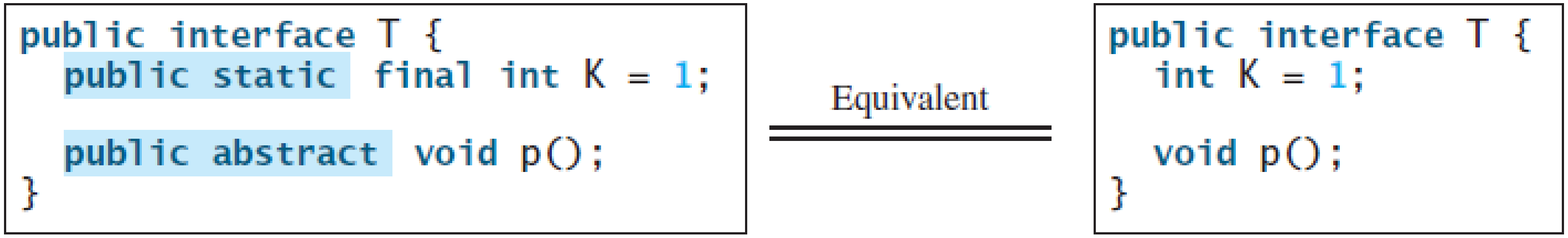
```java

abstract class Animal {
  /** Return animal sound */
  public abstract String sound();
}

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

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

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

Implementing an Interface (Example)

```

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

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

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

A Note Regarding Interfaces

- All data fields of an interface are *public static final*.
- All methods of an interface are *public abstract*.
- Therefore, Java allows these modifiers to be omitted as follows:

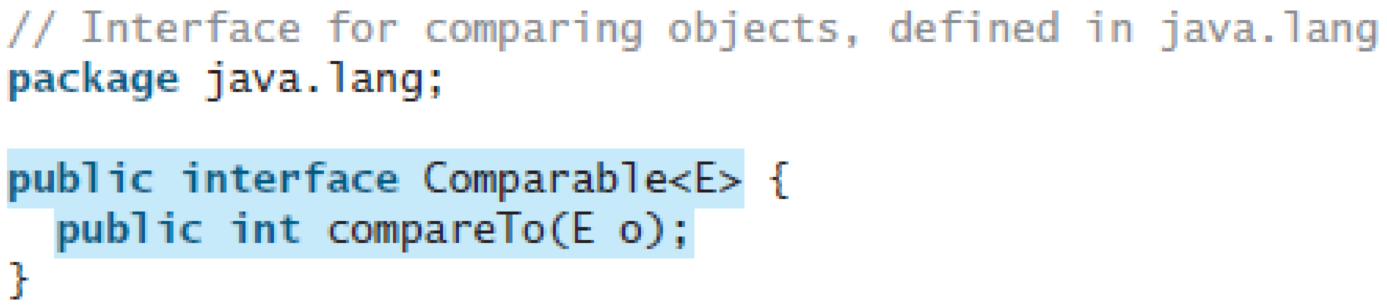


```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

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

  void p();
}
```

The Comparable Interface

- Suppose you want to design a generic method to find the larger of two objects of the same type, such as two students, two dates, two circles, two rectangles, or two squares.
- In order to accomplish this, the two objects must be comparable, so the common behavior for the objects must be comparable.
- Java provides the *Comparable* interface for this purpose.
- The interface is defined as follows:



```
// Interface for comparing objects, defined in java.lang
package java.lang;

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

The Comparable Interface (Cont.)

- 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*.
- The *Comparable* interface is a generic interface.
  - The generic type *E* is replaced by a concrete type when implementing this interface.
- Many classes in the Java library implement *Comparable* to define a natural order for objects.
  - The classes *Byte, Short, Integer, Long, Float, Double, Character, BigInteger, BigDecimal, Calendar, String,* and *Date* all implement the *Comparable* interface.

The Comparable Interface (Example)

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

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

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

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

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

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

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

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

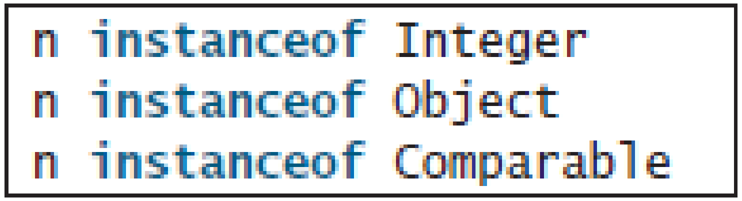
The Comparable Interface (Cont.)

- 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.
- Example:

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

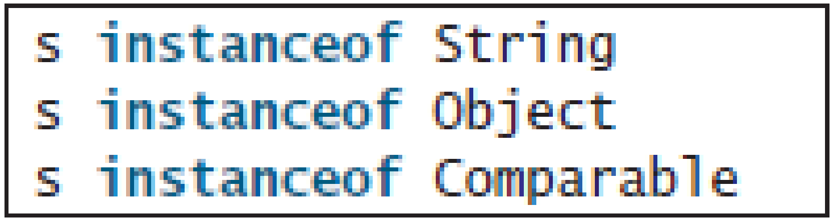
The Comparable Interface (Cont.)

- Let *n* be an *Integer* object:



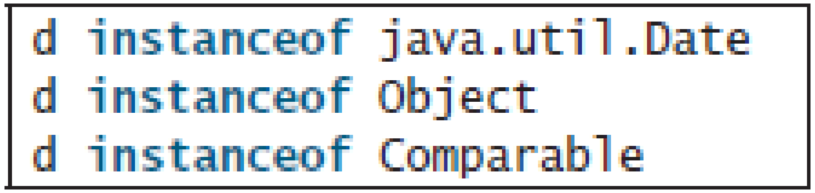
```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

- Let *s* be a *String* object:



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

- Let *d* be a *Date* object:



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

The Comparable Interface (Cont.)

- 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.

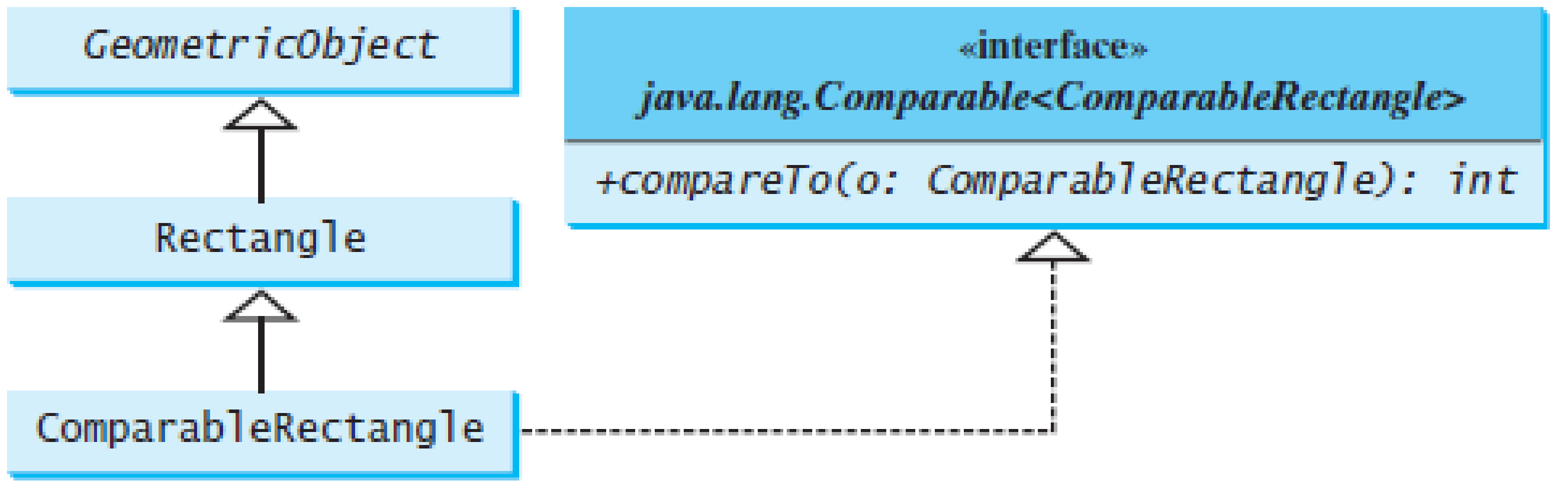
The Comparable Interface (Example)

LISTING 13.8 SortComparableObjects.java

```java
import java.math.*;

public class SortComparableObjects {
  public static void main(String[] args) {
    String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
    java.util.Arrays.sort(cities);
    for (String city: cities)
      System.out.print(city + " ");
    System.out.println();

    BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
      new BigInteger("432232323239292"),
      new BigInteger("54623239292")};
    java.util.Arrays.sort(hugeNumbers);
    for (BigInteger number: hugeNumbers)
      System.out.print(number + " ");
  }
}
```

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

The Comparable Interface
Comparable Rectangle (UML Diagram)


GeometricObject
Rectangle
ComparableRectangle
«interface»
java.lang.Comparable<ComparableRectangle>
+compareTo(o: ComparableRectangle): int

The Comparable Interface
Comparable Rectangle (Code)

```java
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;
  }

  @Override // Implement the toString method in GeometricObject
  public String toString() {
    return super.toString() + " Area: " + getArea();
  }
}
```

The Comparable Interface SortRectangles.java

LISTING 13.10 SortRectangles.java

```java
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 (Rectangle rectangle: rectangles) {
      System.out.print(rectangle + " ");
      System.out.println();
    }
  }
}
```

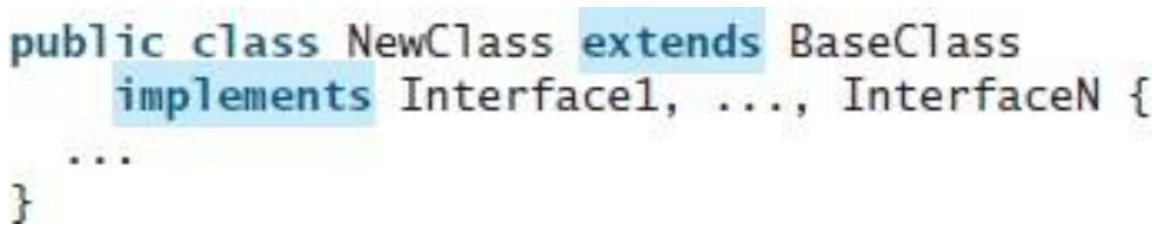
```
Width: 3.4 Height: 5.4 Area: 18.36
Width: 1.4 Height: 25.4 Area: 35.559999999999995
Width: 7.4 Height: 35.4 Area: 261.96
Width: 13.24 Height: 55.4 Area: 733.496
```

Interfaces vs. Abstract Classes

	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.

More on Interfaces

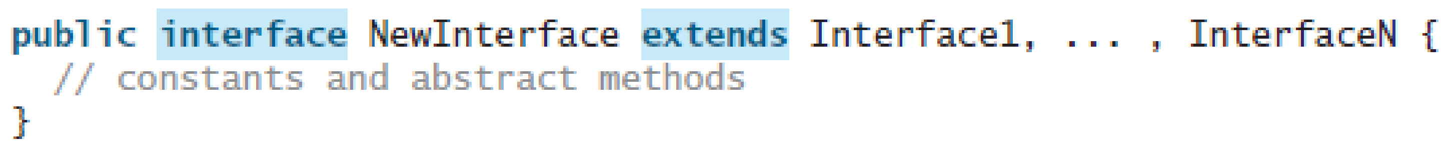
- Java allows only *single inheritance* for class extension, but allow *multiple inheritance* for interface extension.
- For example:



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

More on Interfaces (Cont.)

- An interface can inherit other interfaces using the *extends* keyword.
  - Such an interface is called a sub-interface.
  - An interface can extend other interfaces but not classes.
- For example, *NewInterface* in the following code is a sub-interface of *interface1*, …., and *interfaceN*:



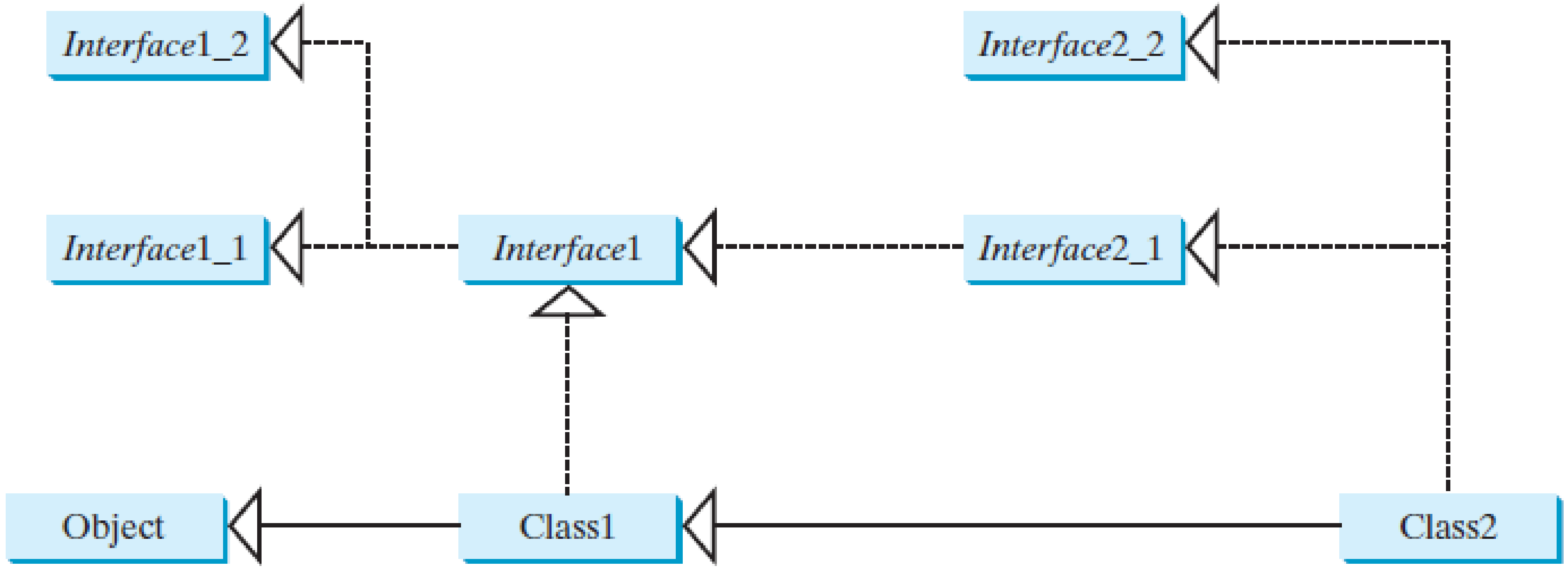
```
public interface NewInterface extends Interface1, ... , InterfaceN {
  // constants and abstract methods
}
```

  - A class implementing NewInterface must implement the abstract methods defined in NewInterface, Interface1, …., and InterfaceN.

More on Interfaces (Cont.)

- All classes share a single root, the *object* class, but there is no single root for interfaces.
- A variable of an interface type can reference any instance of the class that implements the interface.
  - If a class implements an interface, the interface is like a superclass for the class.

More on Interfaces (Cont.)



- Suppose c is an instance of class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Design Guide: Interface or Abstract Class

- Abstract classes and interfaces can both be used to specify common behavior of objects.
- How to decide whether to use an interface or an abstract class?
  - In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes.
    - For example, since an orange is a fruit, their relationship should be modeled using class inheritance.
- 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.
  - When it is desired to define a common supertype for unrelated classes, an interface should be used.

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, for example, but you should not combine students and staff in the same class, because students and staff are different entities.
- A single entity with many responsibilities can be broken into several classes to separate the responsibilities.
- The classes *String*, *StringBuilder*, and *StringBuffer* all deal with strings, for example, but have different responsibilities.
  - The *String* class deals with immutable strings.
  - The *StringBuilder* class is for creating mutable strings.
  - The *StringBuffer* class is similar to *StringBuilder* except that *StringBuffer* contains synchronized methods for updating strings.

Class Design Guidelines Consistency

- Follow standard Java programming style and naming conventions.
- Choose informative names for classes, data fields, and methods.
- A popular style is to place the data declaration before the constructor and place constructors before methods.
- Make the names consistent.
  - It is not a good practice to choose different names for similar operations. For example, the *length()* method returns the size of a *String*, a StringBuilder, and a *StringBuffer*. It would be inconsistent if different names were used for this method in these classes.
- In general, you should consistently provide a public no-arg constructor for constructing a default instance.
  - If a class does not support a no-arg constructor, document the reason.
  - If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.
- If you want to prevent users from creating an object for a class, you can declare a private constructor in the class, as is the case for the *Math* class.

Class Design Guidelines Encapsulation

- A class should use the ***private*** modifier to hide its data from direct access by clients.
- This makes the class easy to maintain.
- 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.