Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Abstract Classes & Interfaces

CS2012: Introduction to Programming II

Introduction

- A ***superclass*** defines common behavior for related subclasses.
  - used for related classes only
- An ***interface*** can be used to define common behavior for any class.
  - used for related or unrelated classes

Abstract Classes

Abstract Classes

- inheritance hierarchy
  - going from superclasses to subclasses
    - each class is more specific and concrete with each new subclass
  - going from subclasses to superclasses
    - each class should become more general and less specific

- superclasses can be so *abstract* that it cannot be used to create any specific instances
  - this is an ***abstract class***
  - mainly used as a good template class from which all subclasses can be derived.

GeometricObject as an Abstract Class

- `GeometricObject` was previously defined as a superclass for `Circle` and `Rectangle`.
  - Both `Circle` and `Rectangle` have `getArea()` and `getPerimeter()` methods
  - Since these are value you can compute for all `GeometricObjects`, these methods should be defined in the `GeometricObject` class
  - catch: we cannot ***implement*** the methods in this class.
    - We wouldn't know which formulas for area or perimeter to use until we knew which specific object we were instantiating.
  - these would be called ***abstract methods*** and are indicated using the `abstract` modifier
- **Any class which has at least one abstract method becomes an abstract class.**
  - Abstract classes are indicated using the `abstract` modifier.

LISTING 13.1 GeometricObject.java

```
 1  public abstract class GeometricObject {
 2    private String color = "white";
 3    private boolean filled;
 4    private java.util.Date dateCreated;
 5
 6    /** Construct a default geometric object */
 7    protected GeometricObject() {
 8      dateCreated = new java.util.Date();
 9    }
10
11    /** Construct a geometric object with color and filled value */
12    protected GeometricObject(String color, boolean filled) {
13      dateCreated = new java.util.Date();
14      this.color = color;
15      this.filled = filled;
16    }
17
          (Code Omitted)

50    /** Abstract method getArea */
51    public abstract double getArea();
52
53    /** Abstract method getPerimeter */
54    public abstract double getPerimeter();
55  }
```

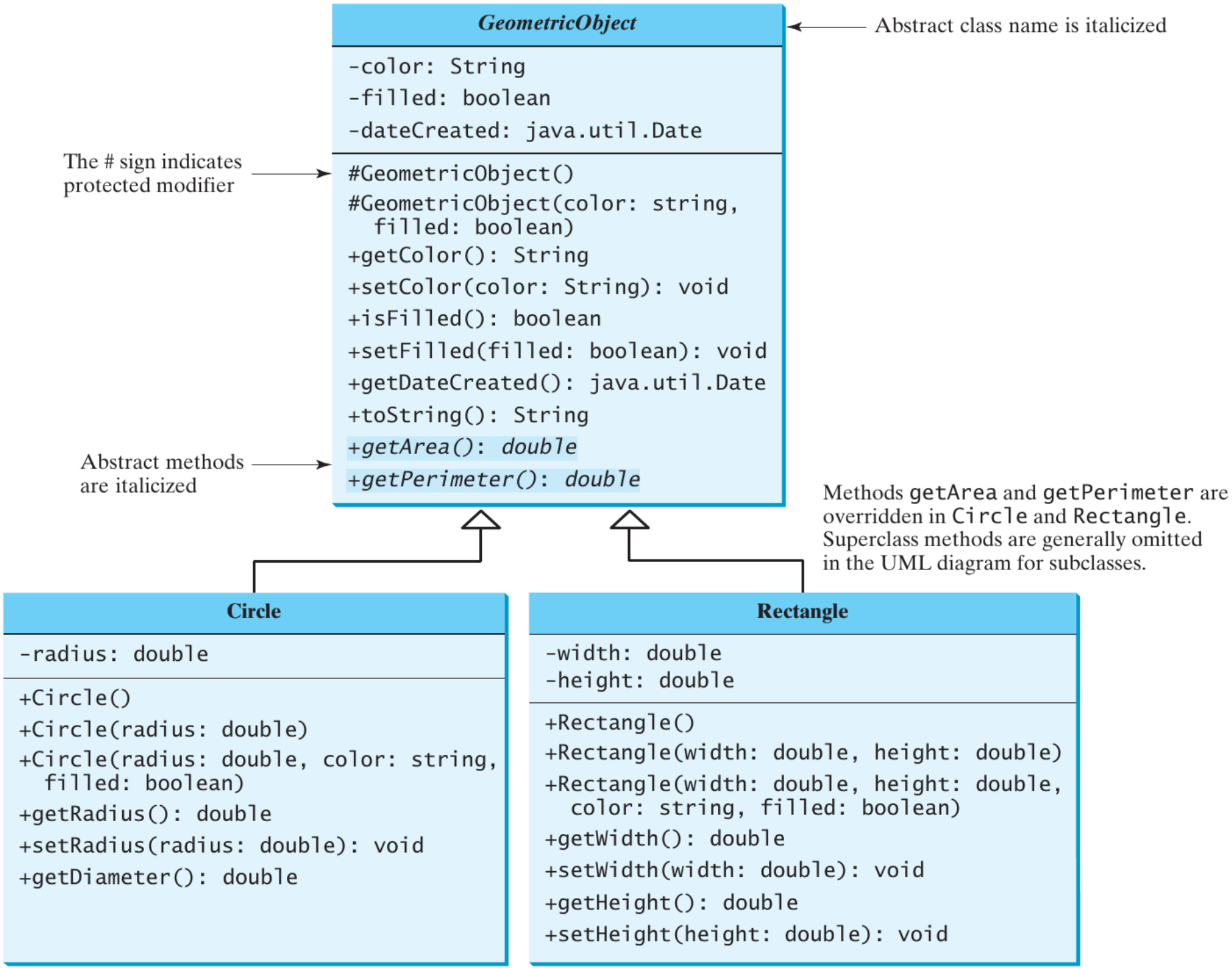

FIGURE 13.1 The new `GeometricObject` class contains abstract methods.

Abstract Classes

- like regular classes, but cannot instantiate with new
  - ~~`GeometricObject go = new GeometricObject();`~~
- abstract methods have no implementation
  - implementation is different from definition
  - implementation is provided by subclasses
- classes with abstract methods must be defined as abstract.
- constructors for abstract classes should be defined as protected so only subclasses in the same package can have access to them.

Rules For Abstract Classes

- An abstract method cannot be in a nonabstract class
  - if a subclass of an abstract superclass does not implement all the abstract methods, the subclass must also be defined as abstract.
  - abstract methods are nonstatic.

- An abstract class cannot be instantiated with new
  - it can still have constructors which can be used in the subclasses.

- A class with abstract methods must be abstract
  - it is possible to define an abstract class without any abstract methods.
  - cannot create an instance of the class
  - is used only as a base class for defining subclass.

Rules for Abstract Classes

- A subclass can override a method from its superclass and define it as abstract
  - very unusual, not commonly done
  - uses when implementation of the method in the superclass becomes invalid in the subclass
  - subclass must be defined as abstract.

- A subclass can be abstract even if the superclass is concrete
  - Example: `Object` is concrete, `GeometricObject` is abstract.

- Abstract classes cannot be instantiated using new
  - an abstract class can be used as a data type
  - Examples:

```
GeometricObject go = new Circle();
GeometricObject[] objects = new GeometricObjects[10];
```

Interfaces

Interfaces

- a class-like construct that contains only constants and abstract methods
- similar to an abstract class, but only used to specify common behaviors for objects of related OR unrelated classes
- Example: with interfaces you can say objects are comparable, edible, cloneable, and/or etc....

Interfaces

- Interfaces are defined as follows:

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

Here is an example of an interface:

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

Interfaces

- treated like special classes
  - each interface compiled into a separate bytecode file
- interfaces can be used more or less the same way as an abstract class
  - Example: an interface can be used as a data type for a reference variable, can be used in casting, etc...
- interface types cannot be instantiated with new

The Edible Interfaces

The `Edible` Interface Example

- specifies whether an object is edible
- any classes that wants to be "edible" should **implement** the Edible interface.
- relationship between an interface and a class is called ***interface inheritance***

The Edible Interface

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

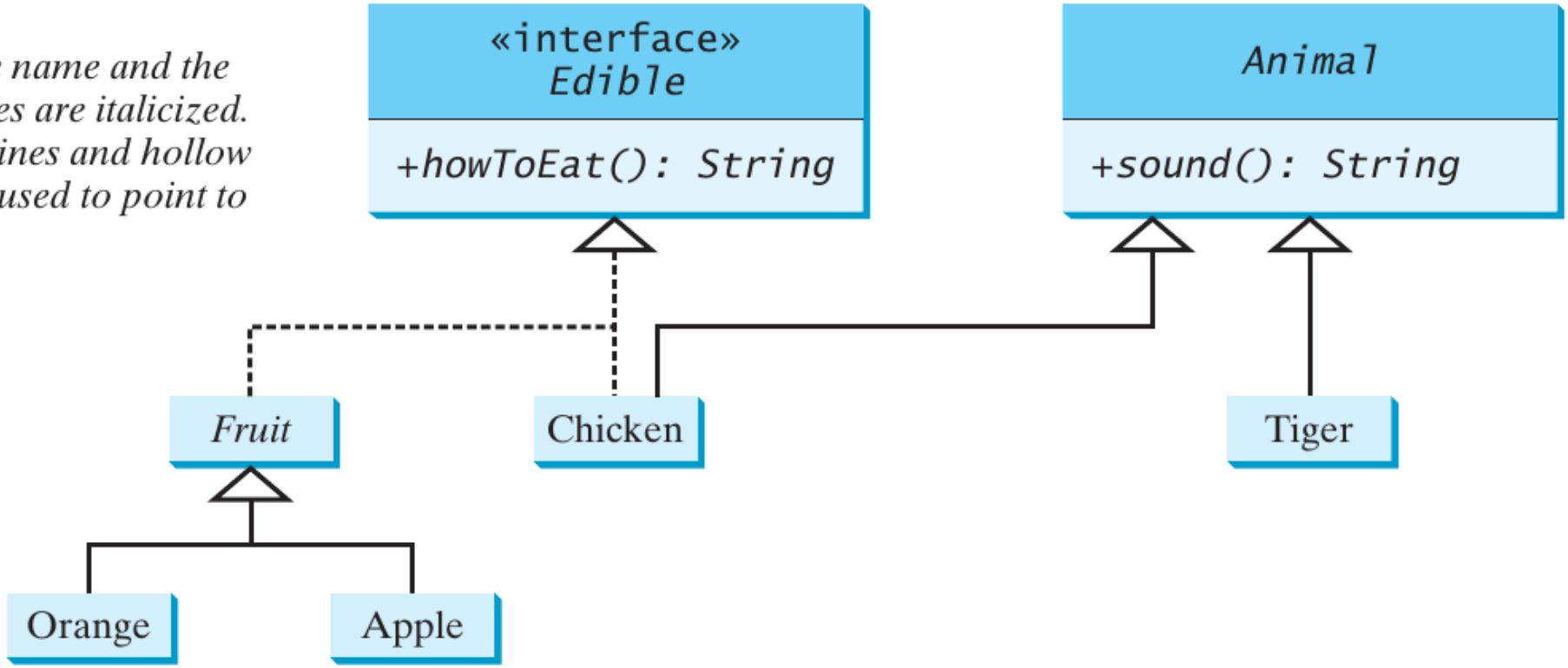

FIGURE 13.4 `Edible` is a supertype for `Chicken` and `Fruit`. `Animal` is a supertype for `Chicken` and `Tiger`. `Fruit` is a supertype for `Orange` and `Apple`.

The `Edible` Interface Example

- **Animal** defines the sound method
  - sound is abstract and will be implemented by a concrete subclass.

- **Chicken** implements **Edible** to say chickens are edible (and yummy!)
  - if a class implements an interface, it implements all of the methods defined in the interface with the exact same signature and return type
  - **Chicken** implements the **howToEat** method
  - **Chicken** extends **Animal** and also implements the sound method.

The `Edible` Interface Example

- **Fruit** implements **Edible**
  - does not implement **howToEat** method so **Fruit** must be **abstract**
  - concrete subclasses of **Fruit** must implement **howToEat**
- **Apple** and **Orange** implement **Edible**
  - they both implement **howToEat**
- **Edible** defines a common behavior for all edible objects
  - all edible objects have the **howToEat** method.

The `Comparable` Interface

The `Comparable` Interface

- Suppose you want to design a general method to find the larger of two objects of the same type
  - two students
  - two dates
  - two circles
  - two rectangles
  - two squares, etc.

- Two objects must be comparable so the common behavior for the objects must be comparable

- Java has a `Comparable` interface for just such a purpose.

The Comparable Interface

- comparable is defined as:

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

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

- `compareTo` determines the order of `this` object with the specified object o
  - returns a negative integer if The Comparable Interface object is less than o
  - returns 0 if both objects are equal
  - returns a positive integer if this object is greater then o

The Comparable Interface

- Comparable is a generic interface, its type E is replaced by a concrete type when implementing the interface.
  - you will learn about generics in CS2013

- all of the wrapper classes implement Comparable

```
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
  }
}
```

```
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 Comparable Interface

```
1  System.out.println(new Integer(3).compareTo(new Integer(5)));
2  System.out.println("ABC".compareTo("ABE"));
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));
```

displays

```
-1
-2
 1
```

The `Comparable` Interface

- Example: Let n be an Integer, s be a String, and d be a Date. All of 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
```

- Example: `Arrays.sort(Object[])` uses `compareTo` to sort the objects in an array. That means for this method to work, the type of the array must implement Comparable.

SortComparableObjects.java

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("232323109292399
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 }
```

```
Atlanta Boston Savannah Tampa
54623239292 432232323239292 232323109292399 2
```

ComparableRectangle.java

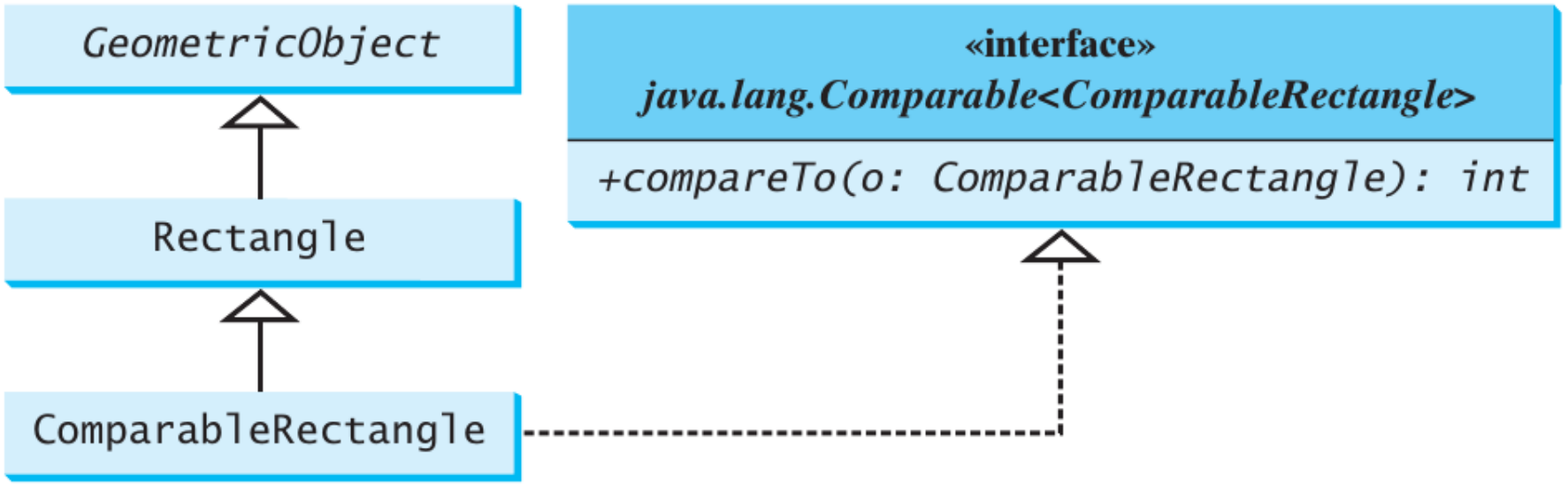

FIGURE 13.5 `ComparableRectangle` extends `Rectangle` and implements `Comparable`.

SortRectangles.java

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  }
```

```
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
```

ComparableRectangle.java

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 }
```

Copy Constructor

Copy Constructor

- A copy constructor, is a special type of constructor which allows one instance of an object to be copied into another instance with the same data.

- The constructor takes one parameter of the same type as its class and is responsible for making a copy of that class.

- Example: If I wanted a Copy Constructor for a Circle object I would have:
  - `public Circle(Circle old)`

Shallow Copy and Deep Copy

- ***shallow copy***:
  - refers to when the reference of an object is copied from one reference variable to another
  - both reference variables would point to the same object in memory.
  - changes to one reference will change the data for both references
- ***deep copy***:
  - refers to when a new instance of an object is created, and everything from the first object is copied over to the second.
  - any reference variables would now point to completely different instances in memory
  - changes to one reference will not affect the other.

Copy Constructor

- Generally, the copy constructor should ALWAYS create a deep copy of the object, otherwise there is no point in having one.

- A shallow copy is the same as doing the following

```
Circle c1 = new Circle(25);
Circle c2 = c1;
```

- So a shallow copy can always be created using the assignment operator, and the copy constructor should always be used to make a deep copy.

Interfaces vs. Abstract Classes

Interfaces vs. Abstract Classes

- classes can implement multiple interfaces, but can only extend one superclass.

- generally an interface can be used the same way as an abstract class, but defining an interface is a bit different from defining an abstract class.

TABLE 13.2 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

Interfaces vs. Abstract Classes

- You can only have single inheritance for extended classes, or multiple implementations for interfaces.

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

- Classes or Interfaces can inherit other interfaces using the extends keyword. this is called a subinterface

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

Interfaces vs. Abstract Classes

- A class implementing `NewInterface` must also implement the abstract methods defined in `NewInterface`, `Interface1`,... and `InterfaceN`
- Interfaces can extend other interfaces but not classes
- Classes can extend one superclass and implement multiple interfaces.

Interfaces vs. Abstract Classes

- All classes share a single root (Object)
  - interfaces do not have a single root
- Interfaces do define a type like classes
- A variable of an interface type can reference any instance of a class which implements that interface
  - almost like a superclass
  - interfaces can be data types
  - interfaces can be cast into a variable of its subclass type and vice versa

Interfaces vs. Abstract Classes

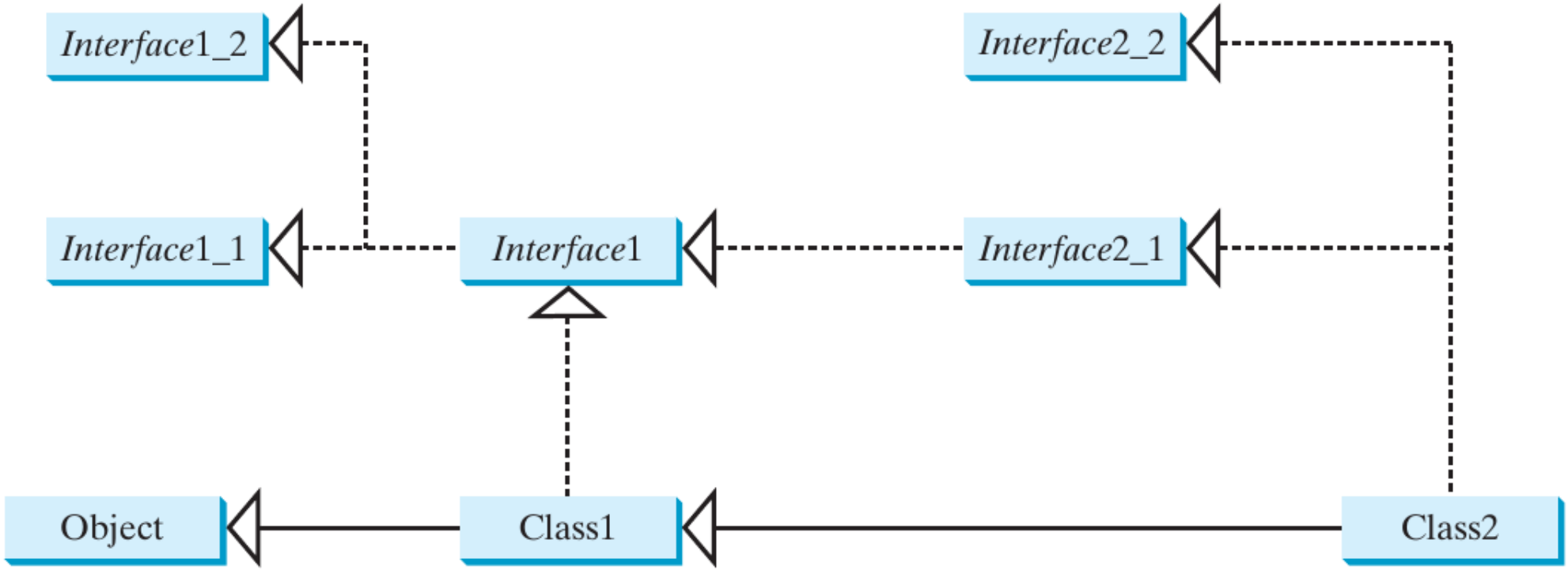

FIGURE 13.7 `Class1` implements `Interface1`; `Interface1` extends `Interface1_1` and `Interface1_2`. `Class2` extends `Class1` and implements `Interface2_1` and `Interface2_2`.

Interfaces vs. Abstract Classes

- Class names are normally nouns
  - Shape, Animal, Person, etc.

- Interface names can be adjectives or nouns
  - Comparable, Cloneable, etc.

Interfaces vs. Abstract Classes

- If Abstract Classes and Interfaces can be used to specify common behavior of objects who do you know which touse?

- If something has a strong "is-a" relationship which clearly indicates a parent-child relationship you should use inheritance / Abstract Classes
  - Student is-a Person

- A weak is-a relationship a.k.a. is-kind-of relationship, indicates that an object posses a certain property
  - this can be modeled using interfaces
  - All strings are comparable, so String implements Comparable.

Interfaces vs. Abstract Classes

- Interfaces are generally preferred over abstract classes, since interfaces can define a common supertype for unrelated classes

- Interfaces are more flexible

- Consider the Animal class, suppose howToEat is defined in Animal

```
abstract class Animal {
  public abstract String howToEat();
}
```

Interfaces vs. Abstract Classes

- Then two subclasses of Animal are:

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

class Duck extends Animal {
  @Override
  public String howToEat() {
    return "Roast it";
  }
}
```

Interfaces vs. Abstract Classes

- Polymorphism allows a variable of type Animal to hold an object of type Duck or Chicken

```
public static void main(String[] args) {
  Animal animal = new Chicken();
  eat(animal);

  animal = new Duck();
  eat(animal);
}

public static void eat(Animal animal) {
  animal.howToEat();
}
```

- JVM dynamically decides which howToEat gets invoked.

Interfaces vs. Abstract Classes

- Now, you can define any subclass of Animal, but we have a restriction:
  - the subclass MUST be another animal i.e. Turkey
  - Example: If you wanted to define a class called JollyRancher, this class could have a howToEat method, but you would have to define a whole new hierarchy (probably with Candy as a superclass) in order to make the inheritance relationship make sense.
  - Extending JollyRancher from Animal does not make any intuitive sense.

Interfaces vs. Abstract Classes

- The previous example shows where an interface would be more flexible in this instance.

- howToEat() can be defined in an interface which could be used for any time of object which can be eaten, not just an Animal.

```
public static void eat(Edible stuff) {
  stuff.howToEat();
}

interface Edible {
  public String howToEat();
}

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

class Duck implements Edible {
  @Override
  public String howToEat() {
    return "Roast it";
  }
}

class Broccoli implements Edible {
  @Override
  public String howToEat() {
    return "Stir-fry it";
  }
}
```

References

- Liang, Chapter 13: Abstract Classes and Interfaces