COMP30820
Java Programming (Conv)

Michael O’Mahony

Chapter 13 Abstract Classes and Interfaces

Objectives

- To design and use abstract classes.
- To specify common behavior for objects using interfaces.
- To define interfaces and define classes that implement interfaces.
- To explore the similarities and differences among concrete classes, abstract classes, and interfaces.

Abstract Classes

The `Circle` and `Rectangle` classes extend the `GeometricObject` class.

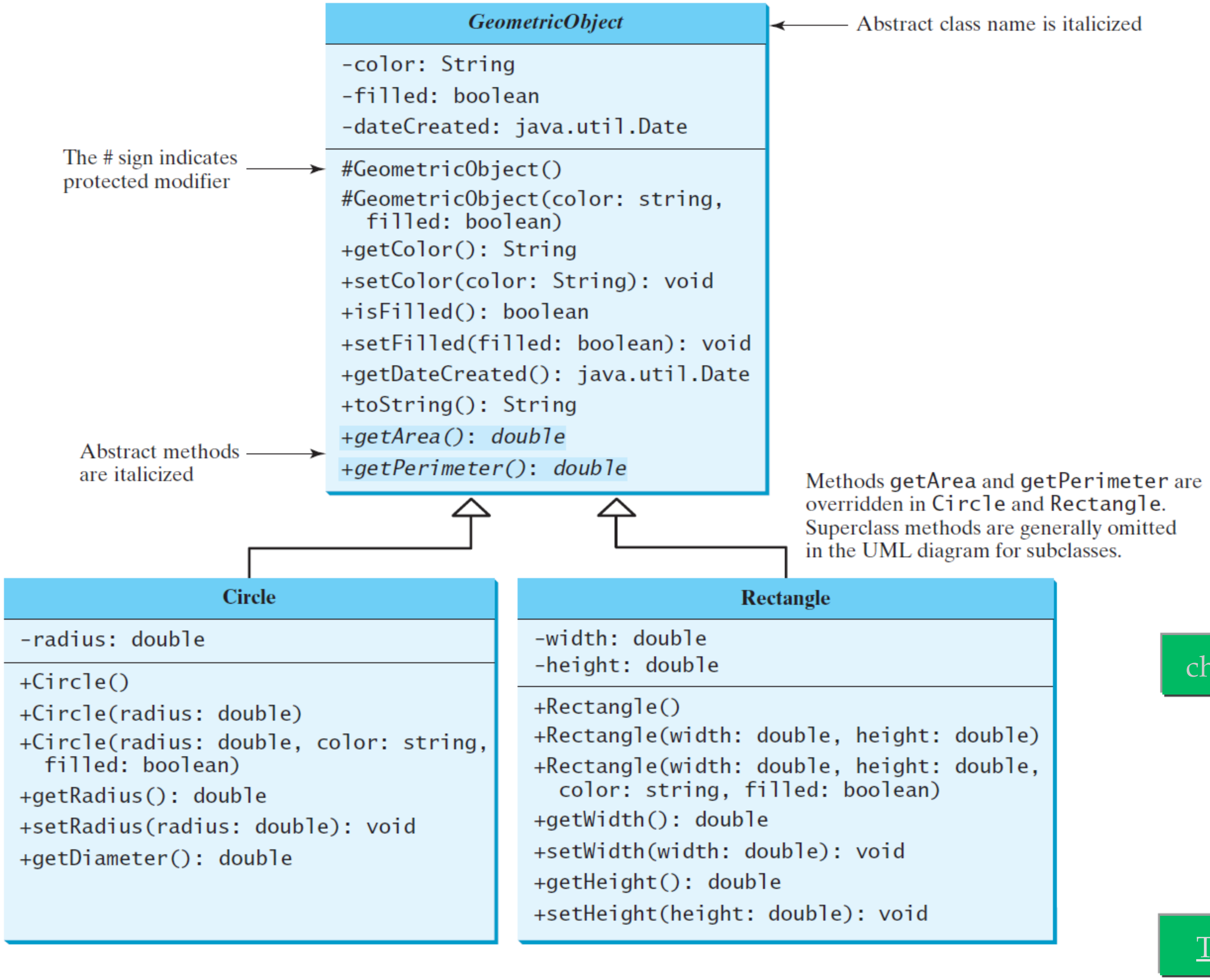
`GeometricObject` models the *common features* of geometric objects:

- Both `Circle` and `Rectangle` contain the `getArea` and `getPerimeter` methods for computing the area and perimeter of a circle and a rectangle.
- Since areas and perimeters can be computed for all geometric objects, ideally `getArea` and `getPerimeter` should be defined in class `GeometricObject`.
- However, these methods cannot be implemented in the `GeometricObject` class, because their implementation depends on the specific type of geometric object….

Solution – the above methods are can be defined as *abstract methods* in class `GeometricObject`.

A class with abstract methods becomes an *abstract class*.

Abstract Classes and Abstract Methods



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

chapter13_examples_1

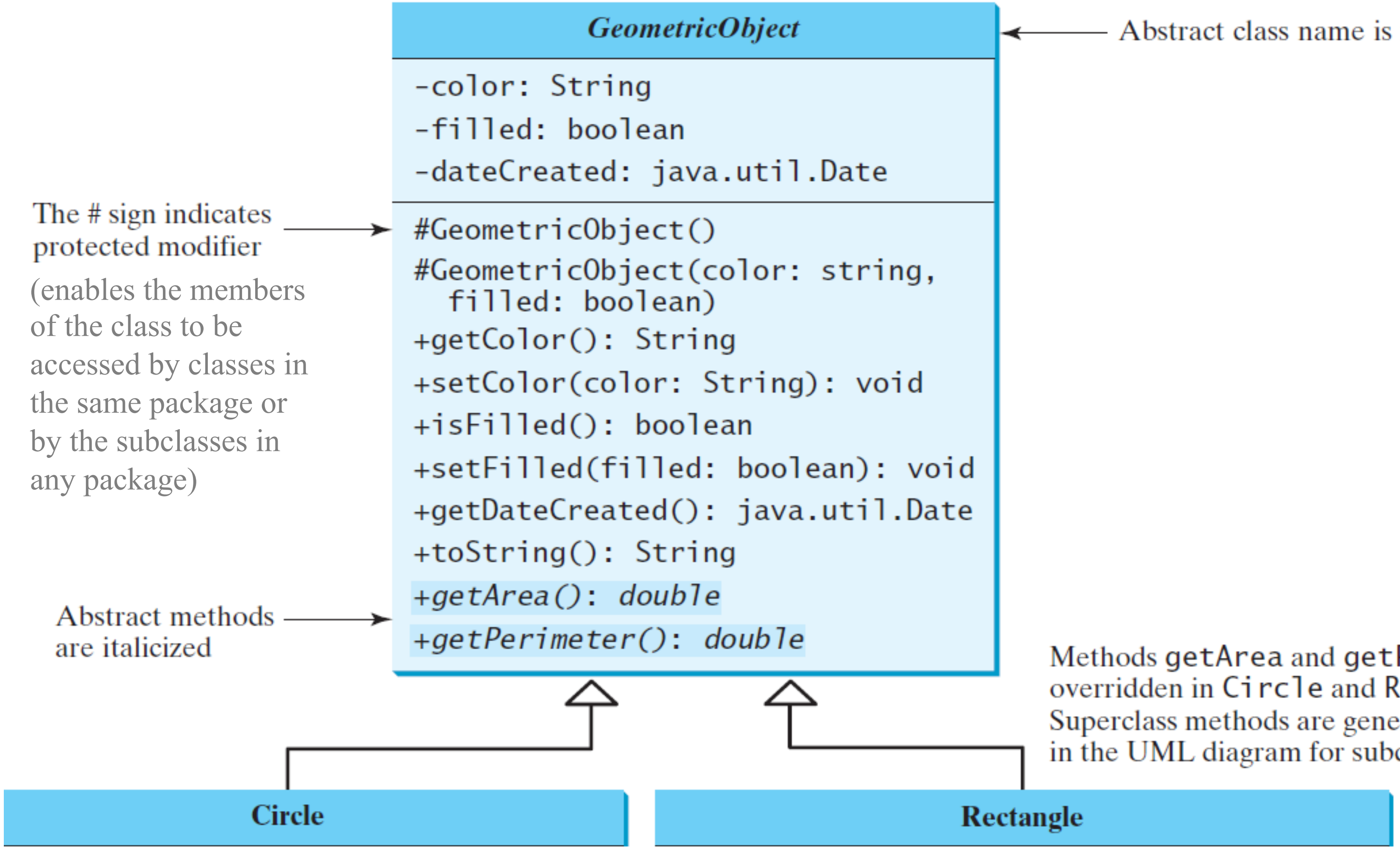
GeometricObject

Circle

Rectangle

TestGeometricObject

Abstract Classes and Abstract Methods



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
Circle
Rectangle

Abstract class name is italicized

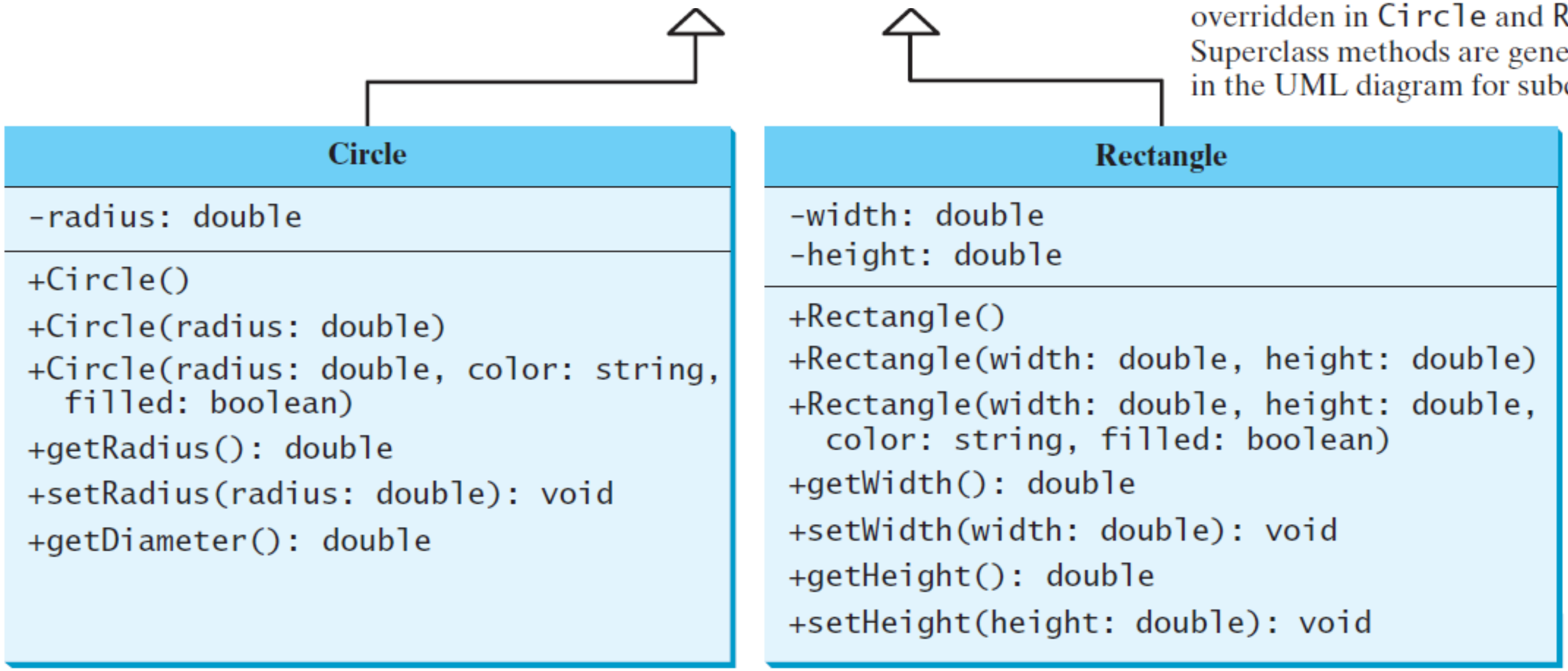
The # sign indicates protected modifier

(enables the members of the class to be accessed by classes in the same package or by the subclasses in any package)

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.

Abstract Classes and Abstract Methods



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
```

Methods `getArea` and `getPerimeter` are overridden in `Circle` and `Rectangle`. Superclass methods are generally omitted in the UML diagram for subclasses.

Abstract Classes and Methods

A class that contains abstract methods must be defined as abstract.

An abstract method is defined without implementation. Its implementation is provided by the subclasses.

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.

Abstract Classes as Types

You cannot create an instance of an abstract class using the `new` operator, but an abstract class can be used as a data type; for example:

```
GeometricObject o = new GeometricObject() // illegal

GeometricObject c = new Circle() // legal
```

As a further example, the following statement creates an array with elements of type `GeometricObject`:

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

You can then create instances of `GeometricObject` and assign their references to the array as follows:

```
objects[0] = new Circle();

objects[1] = new Rectangle(1, 5);
```

Abstract Classes – Constructors

Although an abstract class cannot be instantiated using the `new` operator, you can still define its constructors.

When you create an instance of a subclass, its superclass's constructor is invoked to initialize data fields defined in the superclass.

For example, the constructors of `GeometricObject` are invoked in the `Circle` class and the `Rectangle` class.

The constructor in an abstract class is defined as `protected`, because it is used only by subclasses.

(Recall: the `protected` modifier enables the members of the class to be accessed by classes in the same package or by the subclasses in any package.)

Interfaces

A *superclass* defines common behaviour for **related** *subclasses*.

An *interface* is used to define common behaviour for classes, including **unrelated** classes.

An interface is treated like a special class in Java that contains:

- Initially − only *public constants* and *public abstract methods*.
- Now, can also include *public and private methods*, *static methods*...

Interfaces – Example

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 Test {
  public static final int K = 1;
  public abstract void p();
  public abstract int q(int r);
}
```

To use an interface: `public class A implements Test`

- Class `A` needs to override the methods `p` and `q`

Interfaces

An interface can be used in similar ways to an abstract class:

- For example, an interface can be used as a data type for a reference variable and as the result of casting.
- As with an abstract class, you cannot create an instance from an interface using the `new` operator.

The relationship between a class and an interface is known as *interface inheritance*:

- Since *interface inheritance* and *class inheritance* are essentially the same, both are often referred to as simply inheritance.

The `Comparable` Interface

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

The interface is defined as follows:

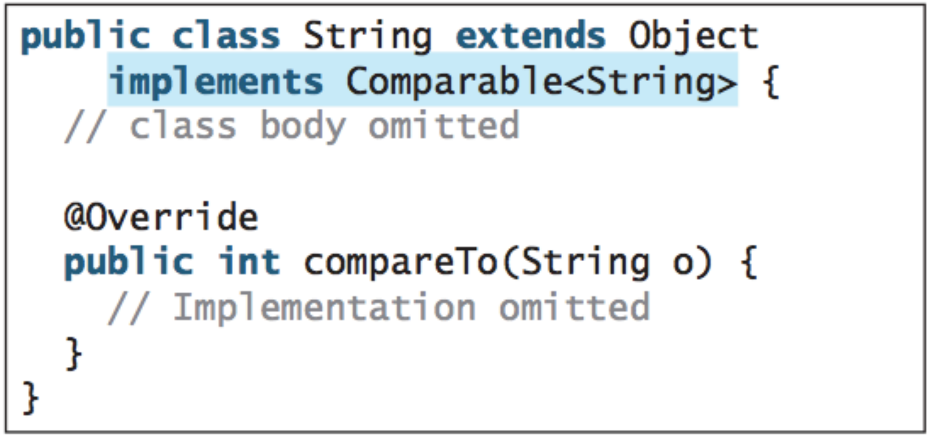
```
public interface Comparable<E> {
  public abstract int compareTo(E o);
}
```

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

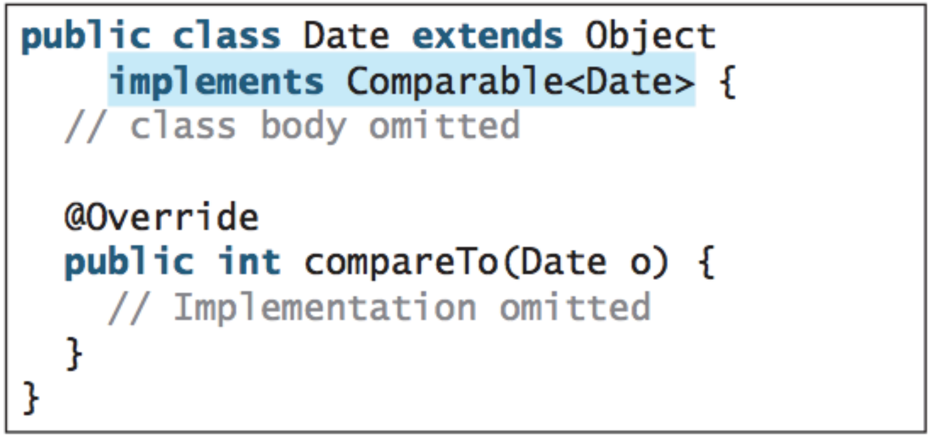
The `Comparable` Interface

Many classes in the Java library implement `Comparable` to define a natural order for objects.

For example, the classes `String` and `Date` (and many others) implement the `Comparable` interface.



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

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



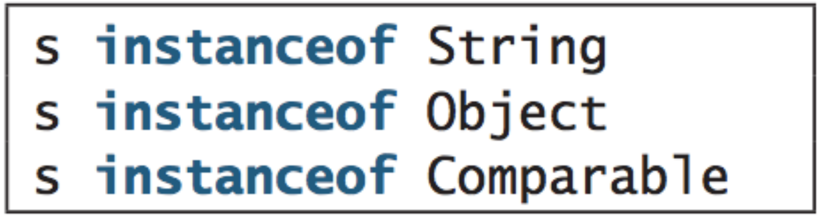
```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

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

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

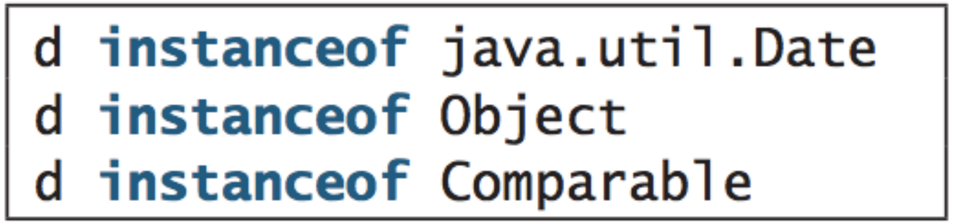
Note

Let `s` be a `String` object and `d` be a `Date` object.

Since both `String` and `Date` extend `Object` and implement the `Comparable` interface, the following expressions are true:



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



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

The `Comparable` Interface

The `compareTo` method determines the order of this object with respect to the specified object `o`.

It returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than object `o`.

The `Comparable` Interface

The `compareTo` method determines the order of this object with respect to the specified object `o`.

It returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than object `o`.

Examples:

```
System.out.println("ABC".compareTo("ABD")); // prints -1

System.out.println("ABC".compareTo("ABC")); // prints 0

System.out.println("ABD".compareTo("ABC")); // prints 1
```

Example: `java.util.Arrays`

The `java.util.Arrays.sort` method in the Java API uses the `compareTo` method to compare and sort objects in an array – *provided that the objects implement the Comparable interface*.

```
public class SortTest {
   public static void main(String[] args) {
      String[] cities = {"Savannah", "Boston", "Tampa"};

      java.util.Arrays.sort(cities);

      for (String city: cities)
         System.out.print(city + " ");
   }
}
```

Example: `java.util.Arrays`

The `java.util.Arrays.sort` method in the Java API uses the `compareTo` method to compare and sort objects in an array – *provided that the objects implement the Comparable interface*.

```
public class SortTest {
   public static void main(String[] args) {
      String[] cities = {"Savannah", "Boston", "Tampa"};

      java.util.Arrays.sort(cities);

      for (String city: cities)
         System.out.print(city + " ");
   }
}
```

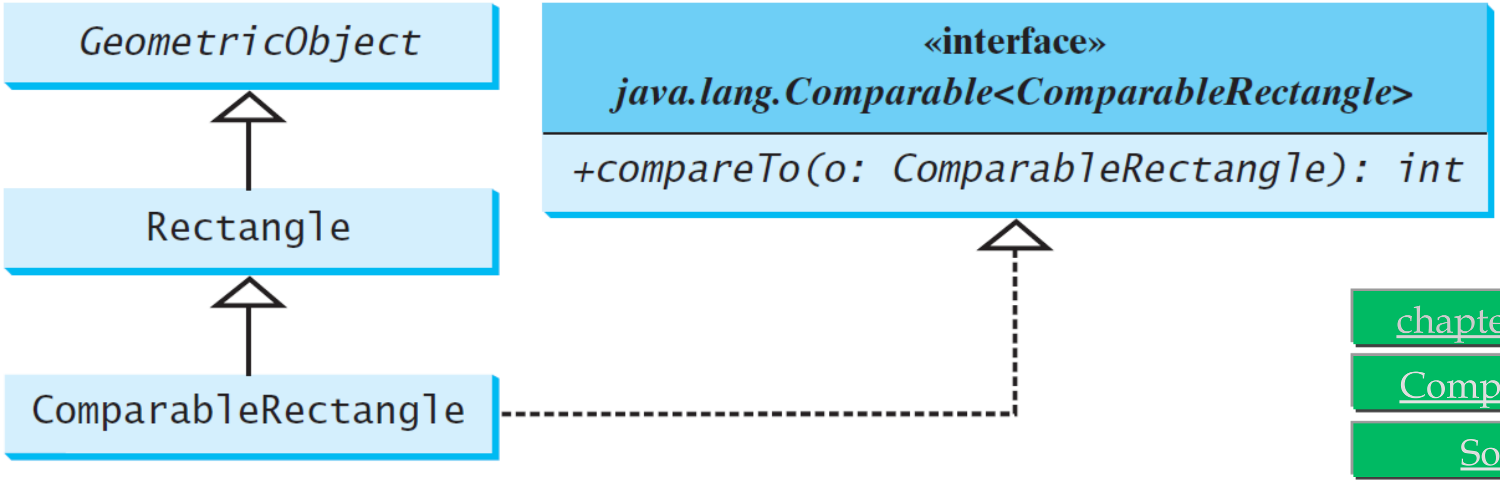
Displays: `Boston Savannah Tampa`

Defining Classes to Implement Comparable

The `java.util.Arrays.sort` method cannot be used to sort an array of `Rectangle` objects, because `Rectangle` does not implement `Comparable`.

In this example, a new rectangle class (`ComparableRectangle`) that implements `Comparable` is defined. The instances of this new class are comparable:

- `ComparableRectangle` extends `Rectangle` and implements `Comparable`
- `ComparableRectangle` inherits the `compareTo` method – in this example, `compareTo` compares two rectangles based on area.
- Note that an instance of `ComparableRectangle` is also an instance of `Rectangle`, `GeometricObject`, `Object`, and `Comparable`.



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

chapter13_examples_2

ComparableRectangle

SortRectangles

Interfaces vs. Abstract Classes

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

Interfaces do not have constructors; abstract classes can have constructors (these are invoked by subclasses when instances of subclasses are created).

Neither abstract classes nor interfaces can be instantiated using the `new` operator.

Interfaces vs. Abstract Classes, cont.

A class can only extend one superclass, but it can implement multiple interfaces.

For example:

```
public class A extends B
  implements Interface1, ..., InterfaceN {
  ...
}
```

Interfaces vs. Abstract Classes, cont.

An interface can inherit other interfaces using the `extends` keyword. Such an interface is called a *subinterface*.

For example, `NewInterface` in the following code is a subinterface of `Interface1`, . . . , and `InterfaceN`:

```
public interface NewInterface extends Interface1, ... , InterfaceN {
  ...
}
```

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

Note that an interface can extend other interfaces but not classes.

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 a class that implements the interface.

Interfaces vs. Abstract Classes, cont.

Both abstract classes and interfaces can be used to model common properties.

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

- For example, an employee *is-a* person, an apple *is-a* fruit…

A **weak** *is-a* relationship (aka *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 and dates are comparable, so the `String` and `Date` classes implement the `Comparable` interface.

You can also use interfaces to circumvent the single inheritance restriction if multiple inheritance is desired:

- In this case, only one superclass but multiple interfaces are permitted.

Example

Suppose we wish to model animals… An animal is a distinct entity and all animals share some common properties. So we use a class to model animals.

- In this example, assume all animals make a *sound*. Also, animals may (or may not) be *edible*.

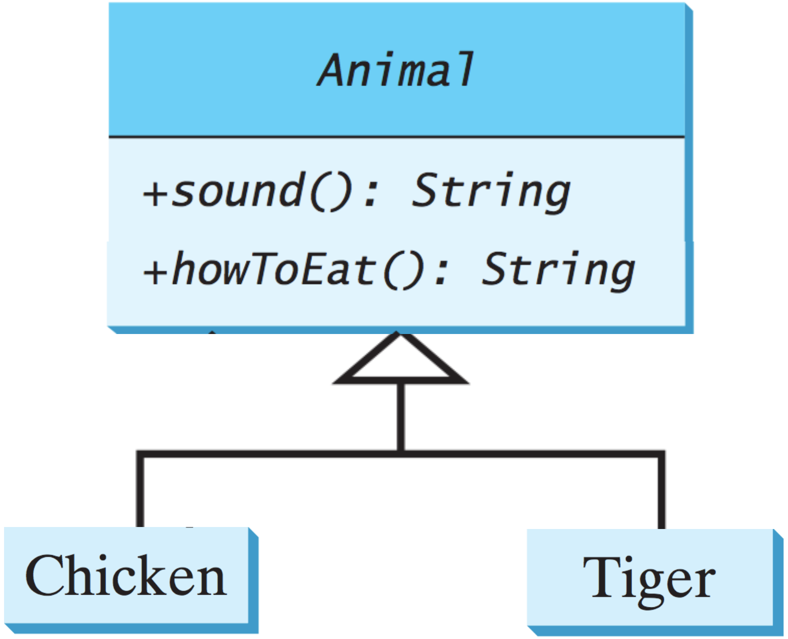
First approach:

- Define a class `Animal` to model the common properties of all animals. Different kinds of animals (cats, dogs) can be modeled as subclasses of `Animal`.
- Use class inheritance because a clear parent-child relationship exists (e.g. a cat *is-an* animal).

Considerations:

- Different animals make different sounds… Also, there are different ways to eat different animals…
- Define abstract methods `sound` and `howToEat` in the `Animal` class, and subclasses of `Animal` will provide suitable implementations for these methods.
- Since `Animal` contains abstract methods, it must be defined as an abstract class.

Example



chapter13_examples_4

TestAnimal

Example

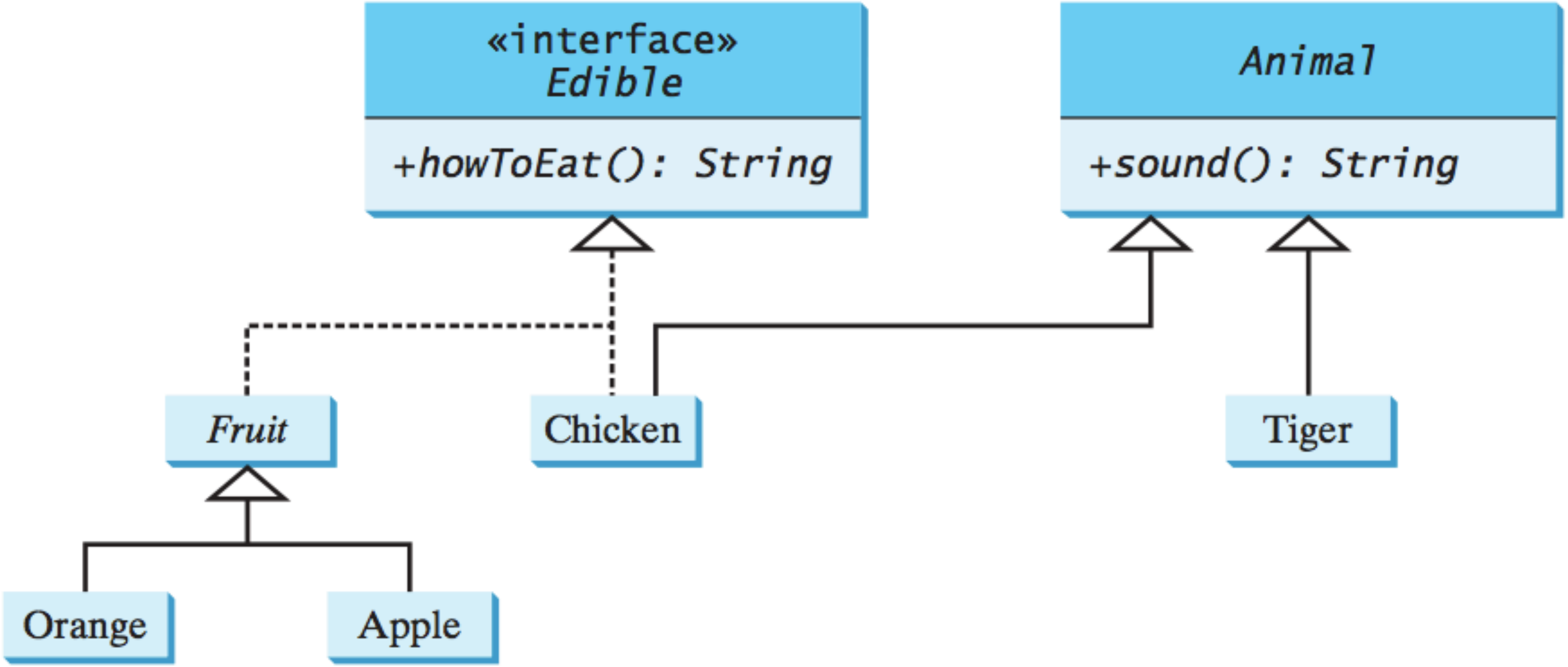
But are all animals edible? Moreover, other entities (fruit, fish, etc.) are also edible… Since “edible” is a property possessed by diverse entities, this property is better modeled using an interface.

Expand the example – consider animals and fruit. Animals and fruit are distinct entities, but they also share certain properties (e.g. edible).

Second approach:

- As before, define a class `Animal` to model the common properties of all animals. Different kinds of animals can be modeled as subclasses of `Animal`.
- Likewise, define a class `Fruit` to model the common properties of all fruit. Different kinds of fruit (apples, oranges) can be modeled as subclasses of `Fruit`.
- As before, use class inheritance for both animals and fruit because clear parent-child relationships exist (e.g. a chicken *is-an* animal, an apple *is-a* fruit).
- Use an interface to specify whether particular animals or pieces of fruit are edible, since this property is possessed by both entities.

Example



Note:

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

chapter13_examples_5

TestEdible

This Lecture…

The three pillars of object-oriented programming are: *encapsulation*, *inheritance*, and *polymorphism*.

Previously, we covered encapsulation, inheritance and polymorphism.

In this lecture, we considered abstract classes and interfaces.