COMP20220
Programming II (Conversion)

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.

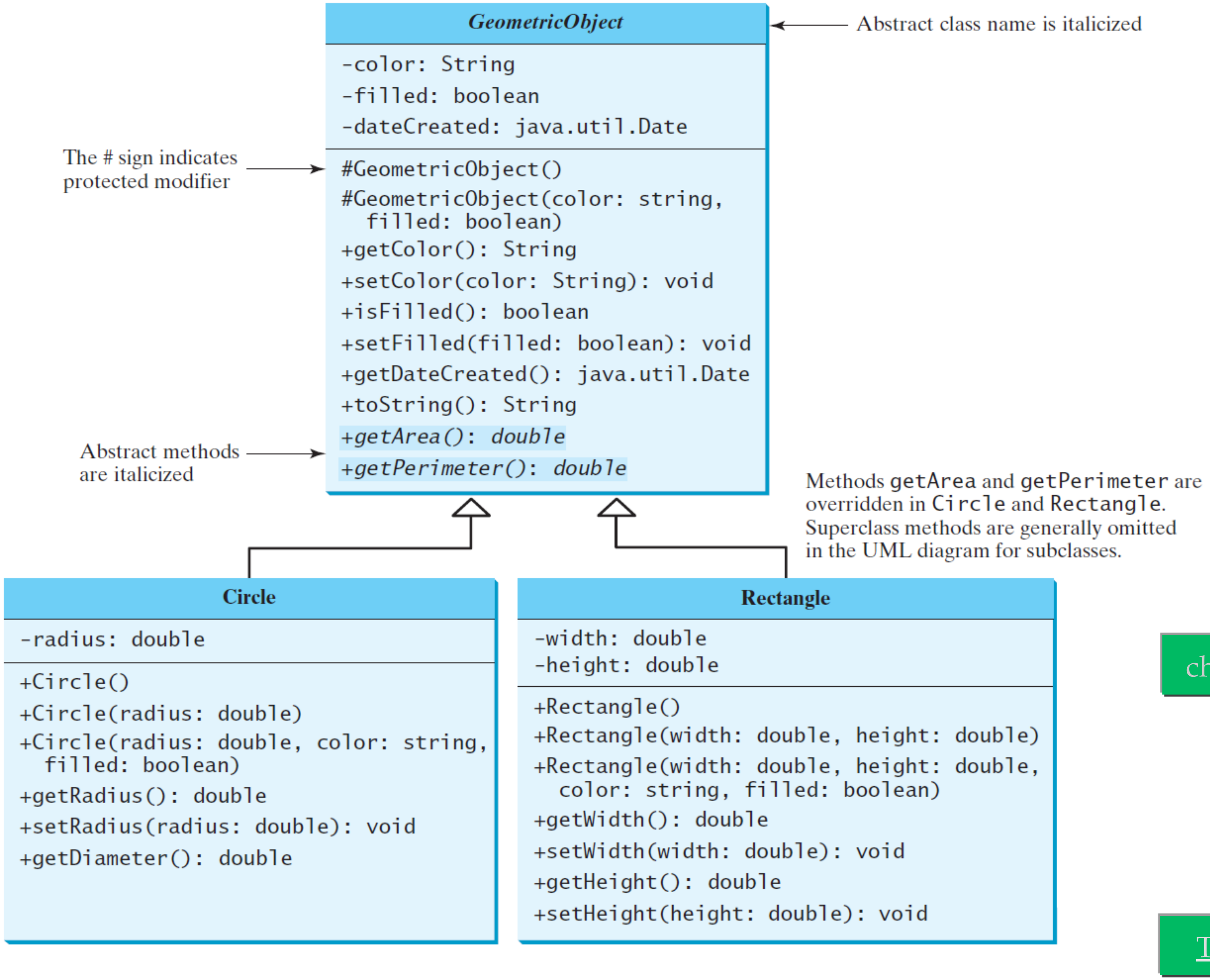
`GeometricObject` models the *common features* of geometric objects:

- 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

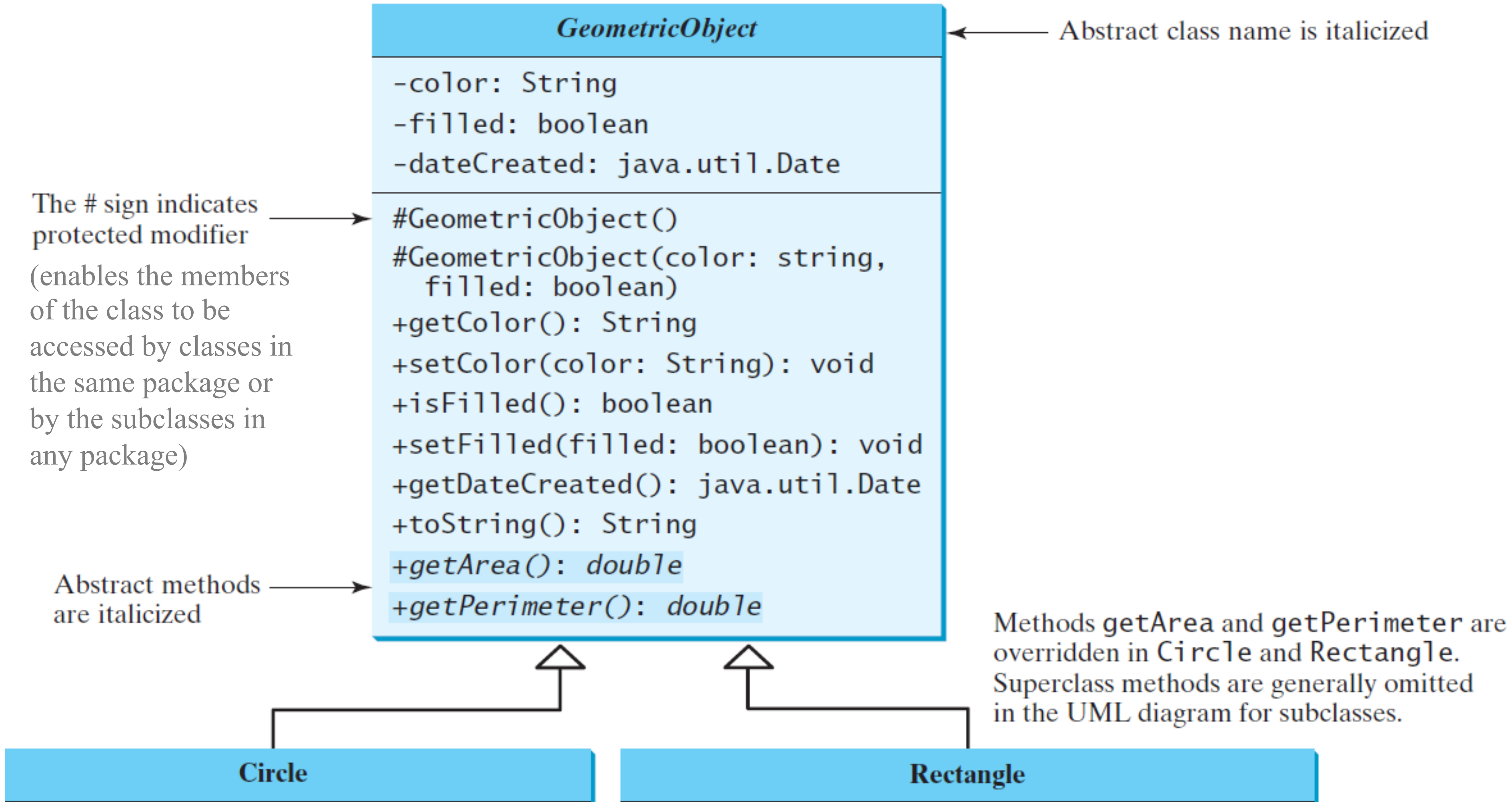
GeometricObject

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

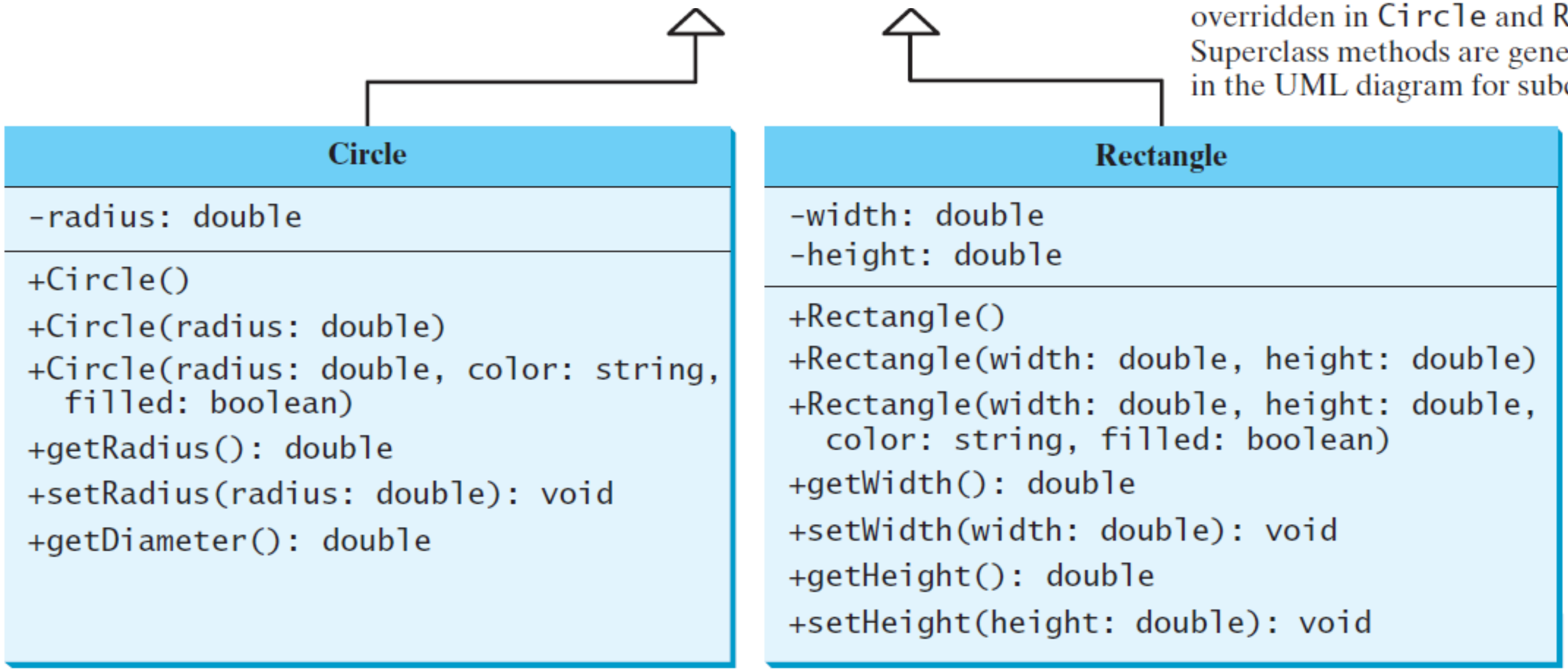
The # sign indicates protected modifier

(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



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
```

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 whose elements are of `GeometricObject` type:

```
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.

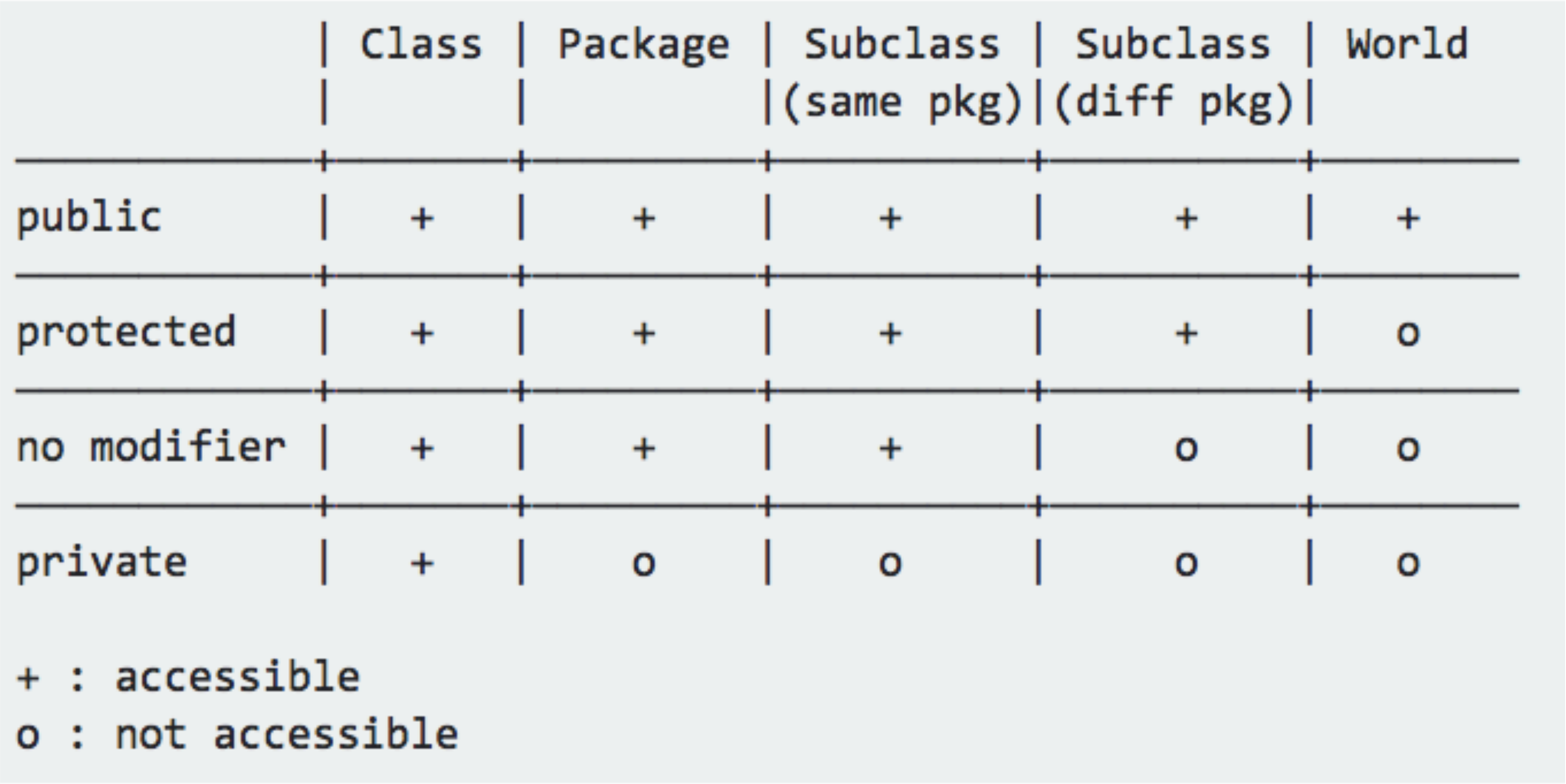
When you create an instance of a subclass, its superclass's constructor is invoked to initialize data fields defined in the superclass.

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.)

Visibility Modifiers



```
            | Class | Package | Subclass | Subclass | World
            |       |         |(same pkg)|(diff pkg)|
————————————+———————+—————————+——————————+——————————+————————
public      |   +   |    +    |    +     |     +    |   +
————————————+———————+—————————+——————————+——————————+————————
protected   |   +   |    +    |    +     |     +    |   o
————————————+———————+—————————+——————————+——————————+————————
no modifier |   +   |    +    |    +     |     o    |   o
————————————+———————+—————————+——————————+——————————+————————
private     |   +   |    o    |    o     |     o    |   o

+ : accessible
o : not accessible
```

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 only *constants* and *abstract methods*.

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.

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
}
```

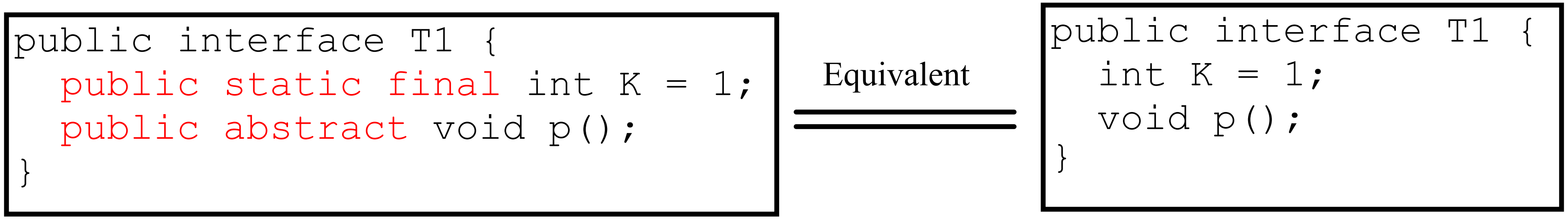
Example:

```
public interface T1 {
  public static final int K = 1;
  public abstract void p();
}
```

Omitting Modifiers in Interfaces

In an interface, all data fields are `public static final` and all methods are `public abstract`.

For this reason, these modifiers can be omitted:



```
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 the following syntax:

```
InterfaceName.CONSTANT_NAME
```

Example:

```
T1.K
```

The `Comparable` Interface

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

The interface is defined as follows:

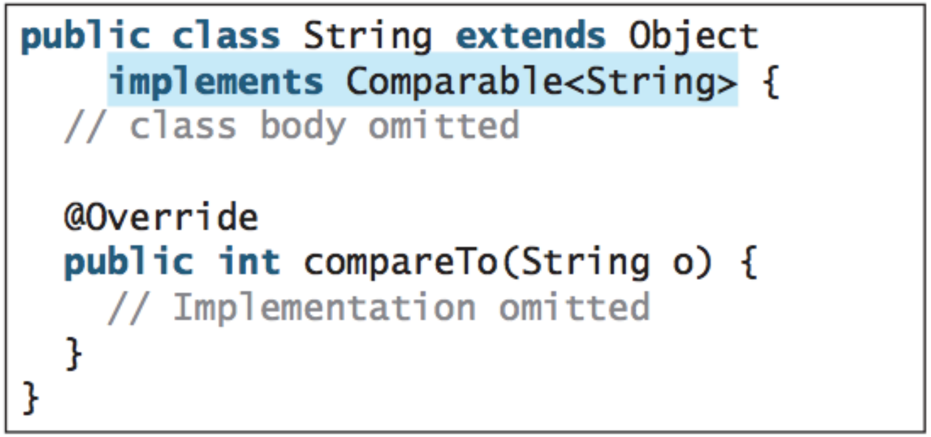
```
public interface Comparable<E> {
  int compareTo(E o);
}
```

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

The `Comparable` Interface

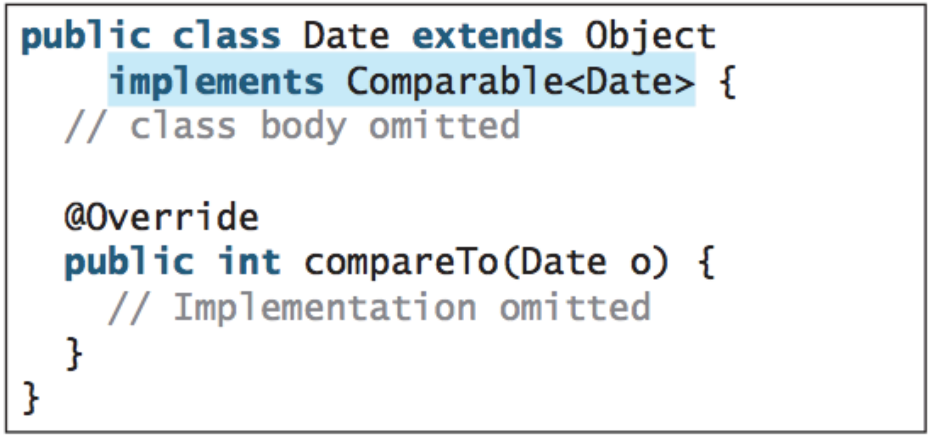
Many classes in the Java library implement `Comparable` to define a natural order for objects.

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
  }
}
```



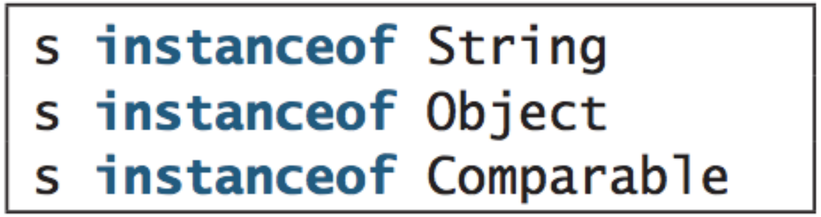
```
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

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

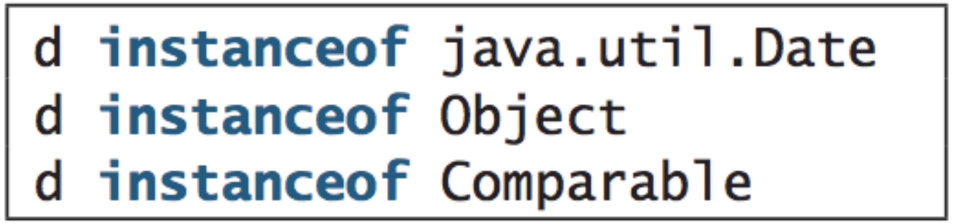
Note

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`.

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

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`.

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

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`.

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

```
Date date1 = new Date(2013, 1, 1);
Date date2 = new Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));
```

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`.

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

Date date1 = new Date(2013, 1, 1);
Date date2 = new Date(2012, 1, 1);
System.out.println(date1.compareTo(date2)); // 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 + " ");
   }
}
```

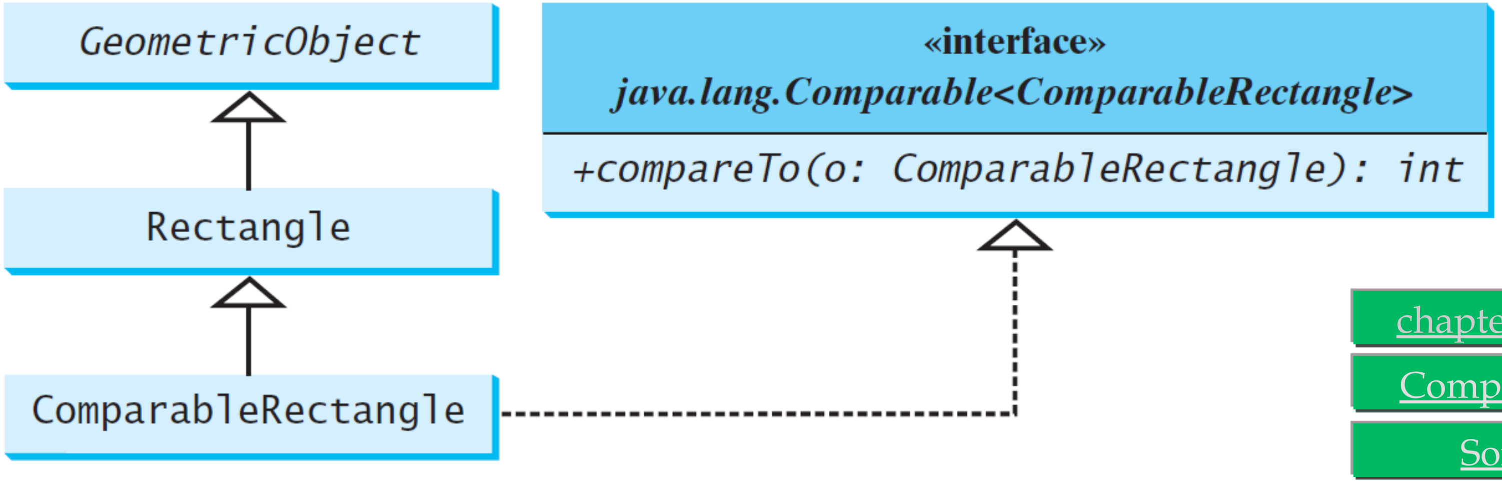
Displays: `Boston Savannah Tampa`

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

The `Cloneable` Interface

Often it is desirable to create a copy of an object.

The `Cloneable` interface specifies that an object can be cloned.

Interfaces contain constants and abstract methods – however, the `Cloneable` interface is a special case. It is defined as follows:

```
public interface Cloneable {
}
```

The body of the interface is empty – such an interface is referred to as a *marker interface*. It is used to denote that a class possesses certain desirable properties.

Instances of classes that implement the `Cloneable` interface can be cloned by overriding the `clone` method defined in the `Object` class.

The `Cloneable` Interface

The `Date` class in the Java API implements the `Cloneable` interface. Thus, instances of this class can be cloned. For example:

```
Date d1 = new Date(); // Creates a new Date object
```

The `Cloneable` Interface

The `Date` class in the Java API implements the `Cloneable` interface. Thus, instances of this class can be cloned. For example:

```
Date d1 = new Date(); // Creates a new Date object
Date d2 = d1; // d2 refers to the same object as d1
```

The `Cloneable` Interface

The `Date` class in the Java API implements the `Cloneable` interface. Thus, instances of this class can be cloned. For example:

```
Date d1 = new Date(); // Creates a new Date object
Date d2 = d1; // d2 refers to the same object as d1
Date d3 = (Date)d1.clone(); // Creates a new object that is a
                            // clone (copy) of d1 - d1 and d3
                            // are different objects with the
                            // same contents
```

The `Cloneable` Interface

The `Date` class in the Java API implements the `Cloneable` interface. Thus, instances of this class can be cloned. For example:

```
Date d1 = new Date(); // Creates a new Date object
Date d2 = d1; // d2 refers to the same object as d1
Date d3 = (Date)d1.clone(); // Creates a new object that is a
                            // clone (copy) of d1 - d1 and d3
                            // are different objects with the
                            // same contents
```

What do the following statements display?

```
System.out.println(d1.equals(d2));
System.out.println(d1.equals(d3));
System.out.println(d1 == d2);
System.out.println(d1 == d3);
```

The `Cloneable` Interface

The `Date` class in the Java API implements the `Cloneable` interface. Thus, instances of this class can be cloned. For example:

```
Date d1 = new Date(); // Creates a new Date object
Date d2 = d1; // d2 refers to the same object as d1
Date d3 = (Date)d1.clone(); // Creates a new object that is a
                            // clone (copy) of d1 – d1 and d3
                            // are different objects with the
                            // same contents
```

What do the following statements display?

```
System.out.println(d1.equals(d2)); // true
System.out.println(d1.equals(d3)); // true
System.out.println(d1 == d2);      // true
System.out.println(d1 == d3);      // false
```

Implementing `Cloneable` Interface

To define a class that implements the `Cloneable` interface, the class must override the `clone` method defined in the `Object` class.

The method header of `clone` in `Object` is:

```
protected native Object clone() throws
CloneNotSupportedException;
```

The keyword `native` indicates that this method is not written in Java but is implemented in the JVM for the native platform.

The keyword `protected` restricts accessibility to classes in the same package and subclasses. The classes which override this method change the visibility modifier to `public` so that the method can be used in any package.

The `CloneNotSupportedException` exception is thrown if the class does not implement the `Cloneable` interface. (Exceptions will be covered in a future lecture.)

Implementing Cloneable Interface

Since the clone method implemented in the Object class performs the task of cloning objects (for the native platform), the overridden clone method simply invokes super.clone() as follows:

```
// Override the protected clone method defined in
// the Object class, and strengthen its accessibility
@Override
public Object clone() throws CloneNotSupportedException {
  return super.clone();
}
```

The following example defines a class named House that implements Cloneable and Comparable.

chapter13_examples_3

House

TestHouse

Implementing `Cloneable` Interface

An identical copy of a house object is created as follows:

```
House house1 = new House(1, 1759.50);
House house2 = (House)house1.clone();
```

`house1` and `house2` are two different objects with identical contents.

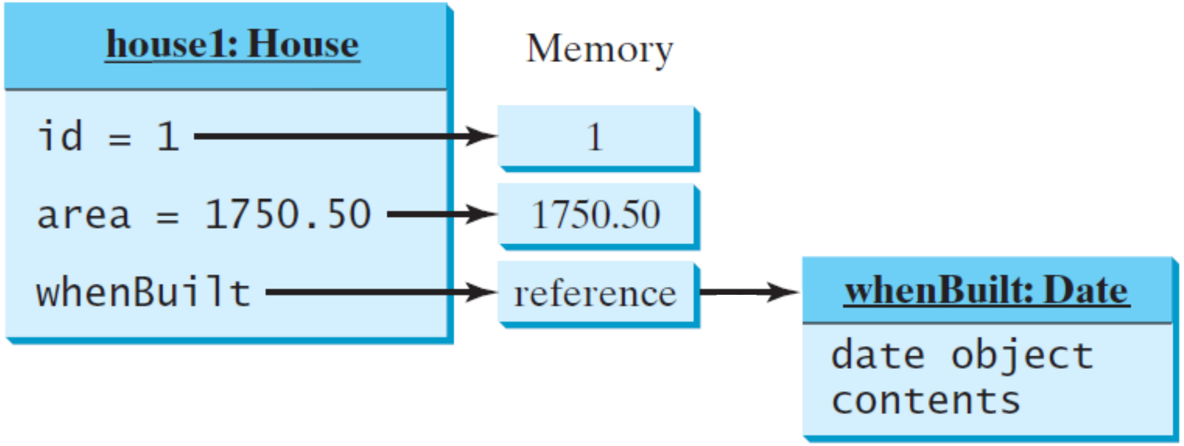
The `clone` method in the `Object` class copies each data field from the original object to the target object:

- If the data field is of a primitive type, its value is copied. For example, the value of `area` (of type `double`) is copied from `house1` to `house2`.
- If the data field is of an object, the reference of the field is copied. For example, the date field `whenBuilt` is of the `Date` class, so its reference is copied into `house2`. Therefore, `house1.whenBuilt == house2.whenBuilt` is true, although `house1 == house2` is false.
- This is referred to as a *shallow copy* rather than a *deep copy*, meaning that if the field is of an object type, the object's reference is copied rather than its contents.

Shallow vs. Deep Copy

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

Shallow Copy

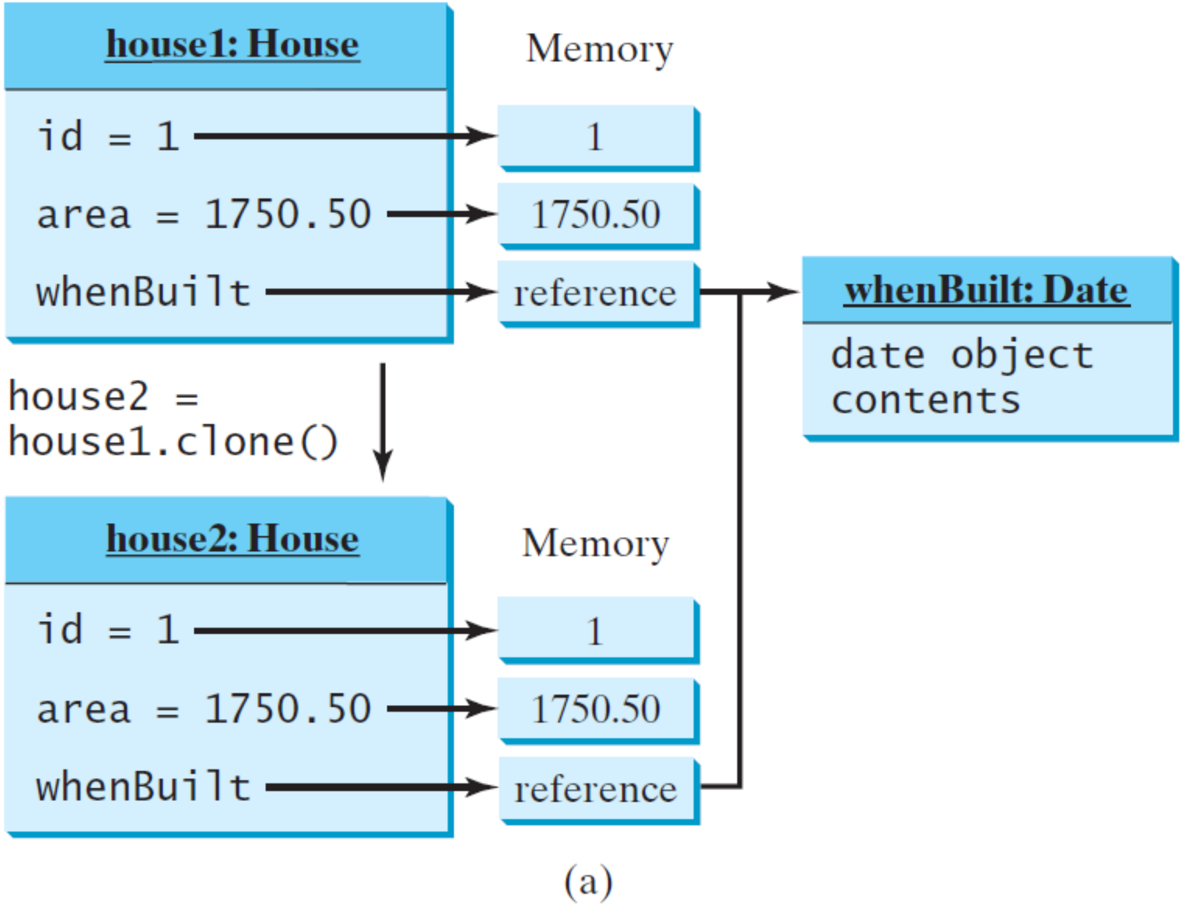


(a)

Shallow vs. Deep Copy

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

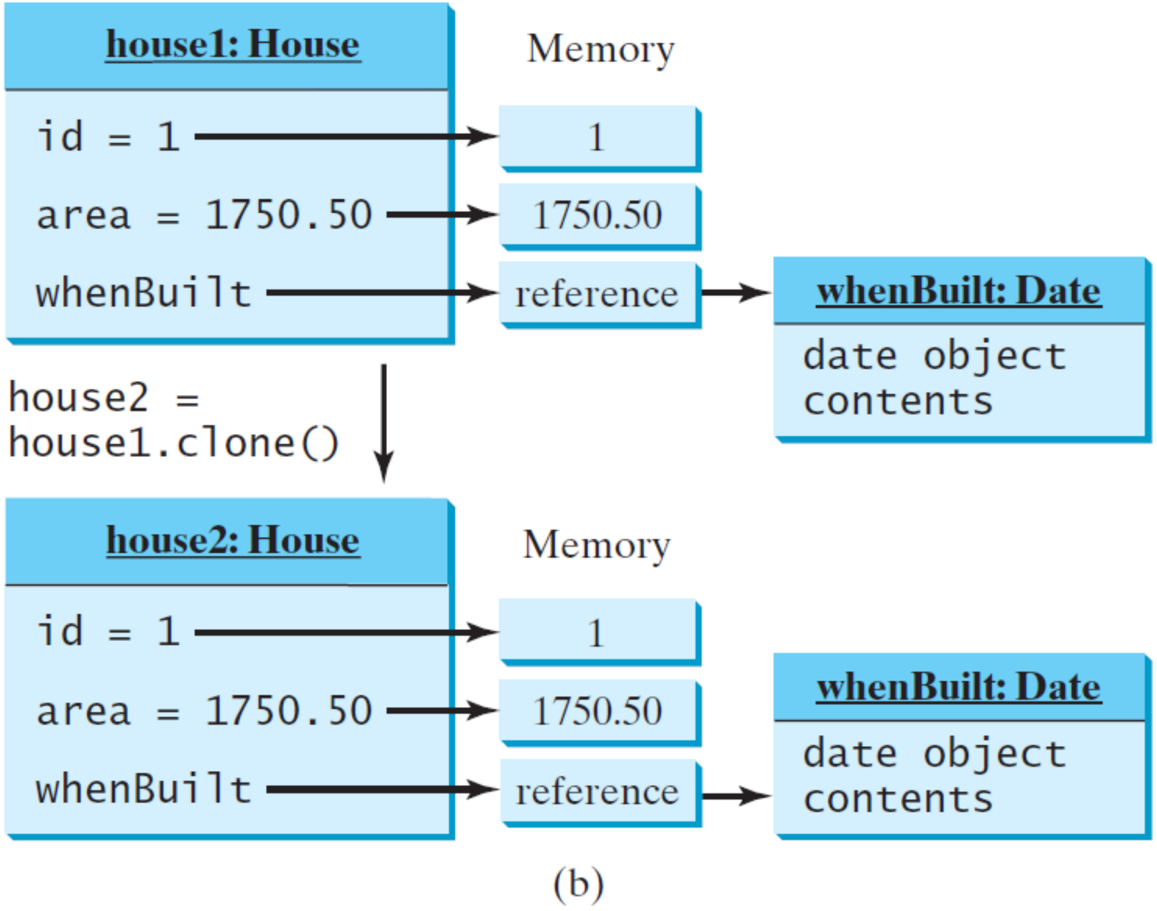
Shallow Copy



Shallow vs. Deep Copy

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```

Deep Copy



Shallow vs. Deep Copy

To perform a *deep copy* for a `House` object, change the `clone` method from:

```
public Object clone() throws CloneNotSupportedException {
  return super.clone();
}
```

to:

```
public Object clone() throws CloneNotSupportedException {
  // Perform a shallow copy
  House houseClone = (House)super.clone();

  // Perform a deep copy on whenBuilt
  houseClone.whenBuilt = (Date)whenBuilt.clone();

  return houseClone;
}
```

Now, `house1` and `house2` contain different `Date` objects; `house1.whenBuilt == house2.whenBuilt` is false.

chapter13_examples_3

HouseDeepCopy

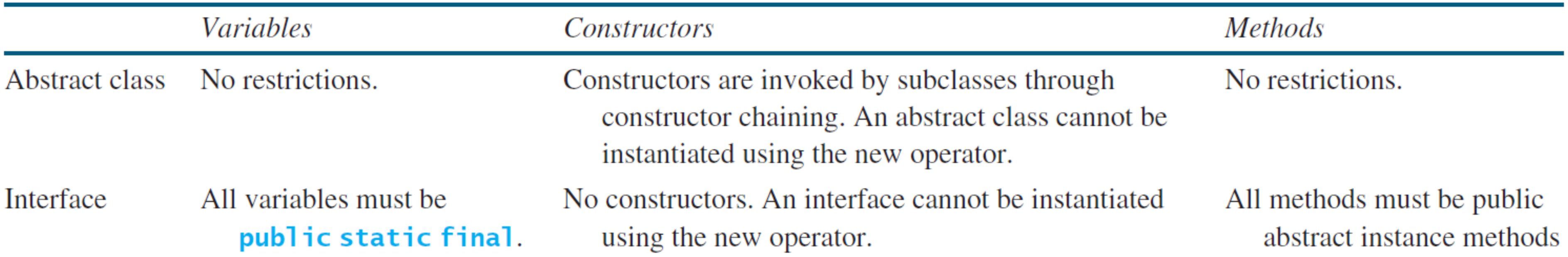
TestHouseDeepCopy

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.

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.

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 {
  // constants and abstract methods
}
```

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.

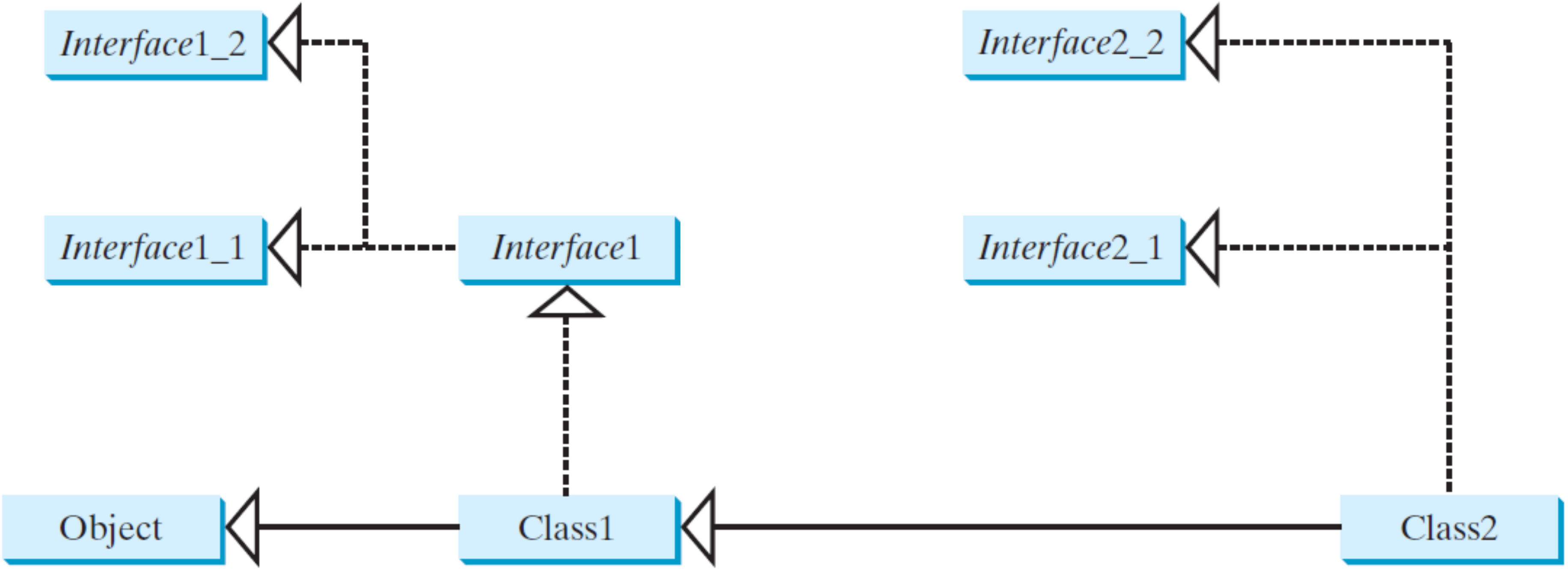
Like a class, an interface also defines a type.

A variable of an interface type can reference any instance of a class that implements the interface.

Interfaces vs. Abstract Classes, cont.

Consider the following. `Class1` implements *`Interface1`*; *`Interface1`* extends *`Interface1_1`* and *`Interface1_2`*. `Class2` extends `Class1` and implements *`Interface2_1`* and *`Interface2_2`*.

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



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*.

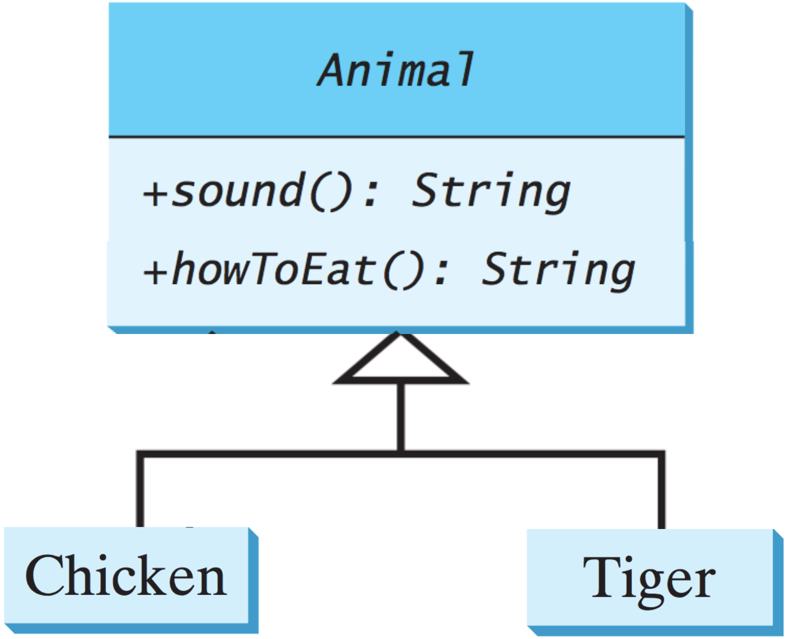
First approach:

- 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

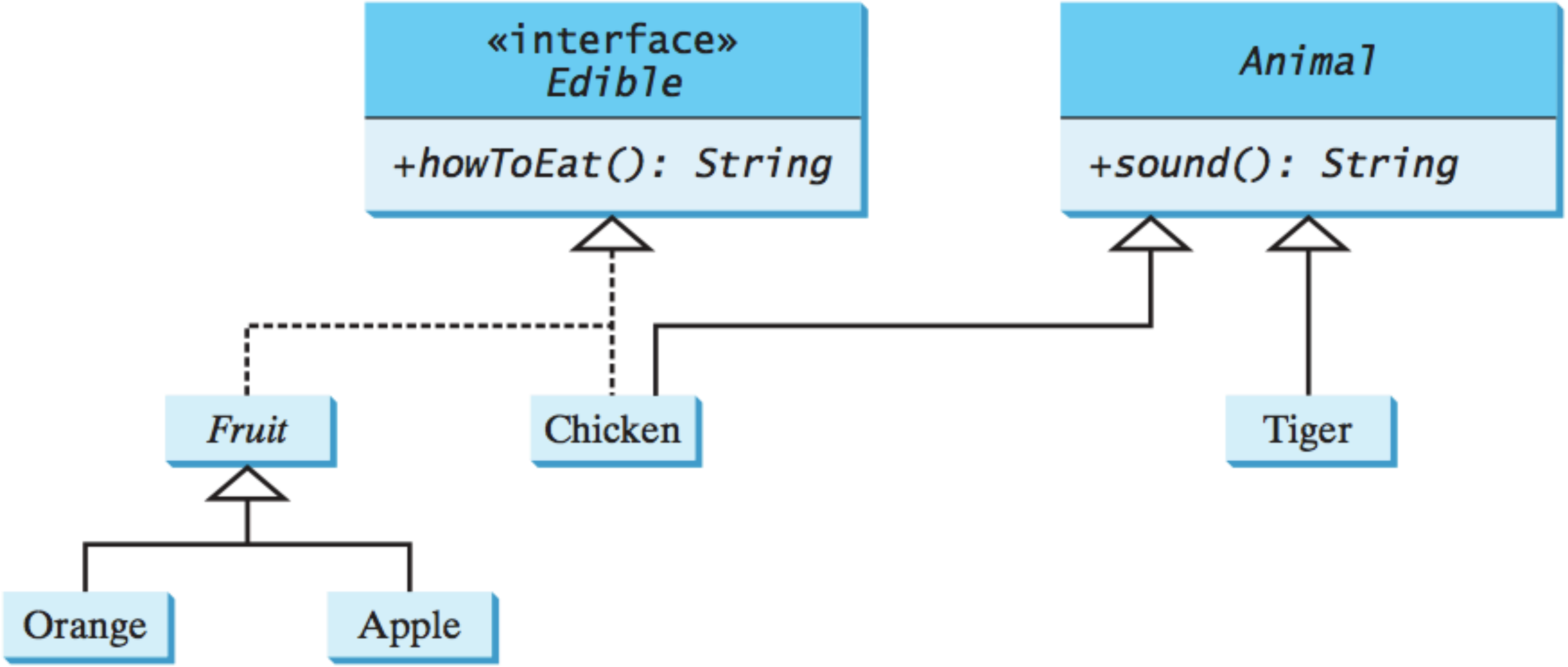
But are all animals edible? Moreover, other entities (fruit, fish, etc.) are also edible… Since “edible” is a property possessed by diverse entities, this property is better modeled using an interface.

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. In general, interfaces provide more flexibility than classes, because different types of classes can implement the same interfaces.

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.

Remaining topics: exception handling and text I/O.