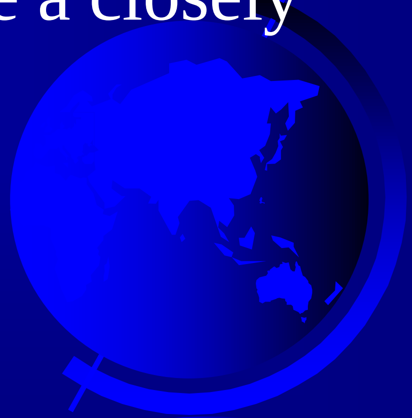
Chapter 13 Abstract Classes and Interfaces

Motivations

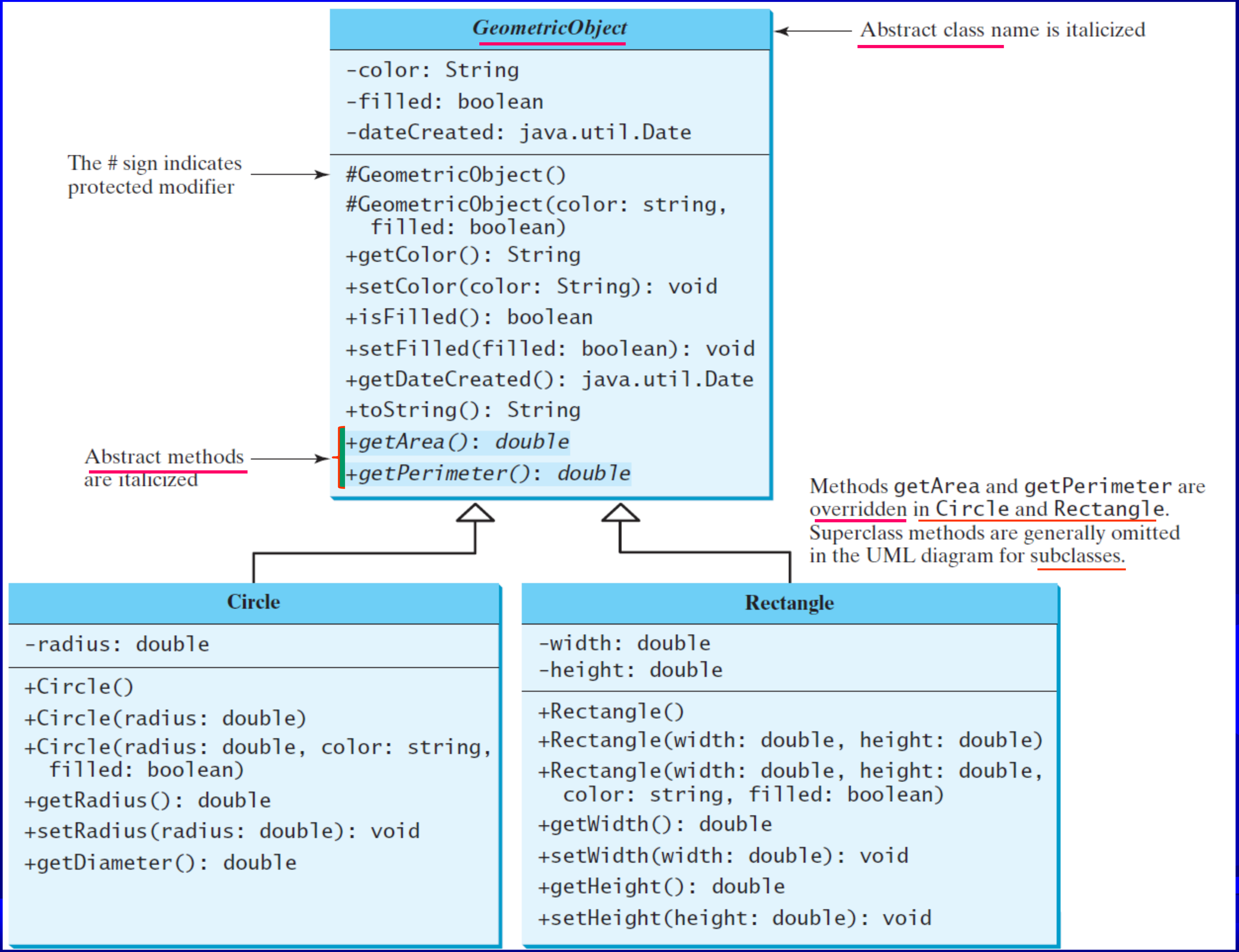
- You have learned how to write simple programs to create and display GUI components. But how to respond to user actions, such as clicking a button to perform an action?
- An **interface** is for defining common behavior for classes (including unrelated classes).
  - Before discussing interfaces, we introduce a closely related subject: **abstract classes**.



Objectives

- To design and use abstract classes (§13.2).
- To generalize numeric wrapper classes, **BigInteger**, and **BigDecimal** using the abstract **Number** class (§13.3).
- To process a calendar using the **Calendar** and **GregorianCalendar** classes (§13.4).
- To specify common behavior for objects using interfaces (§13.5).
- To define interfaces and define classes that implement interfaces (§13.5).
- To define a natural order using the **Comparable** interface (§13.6).
- To make objects cloneable using the **Cloneable** interface (§13.7).
- To explore the similarities and differences among concrete classes, abstract classes, and interfaces (§13.8).
- To design the **Rational** class for processing rational numbers (§13.9).
- To design classes that follow the class-design guidelines (§13.10).

Abstract Classe/Method



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

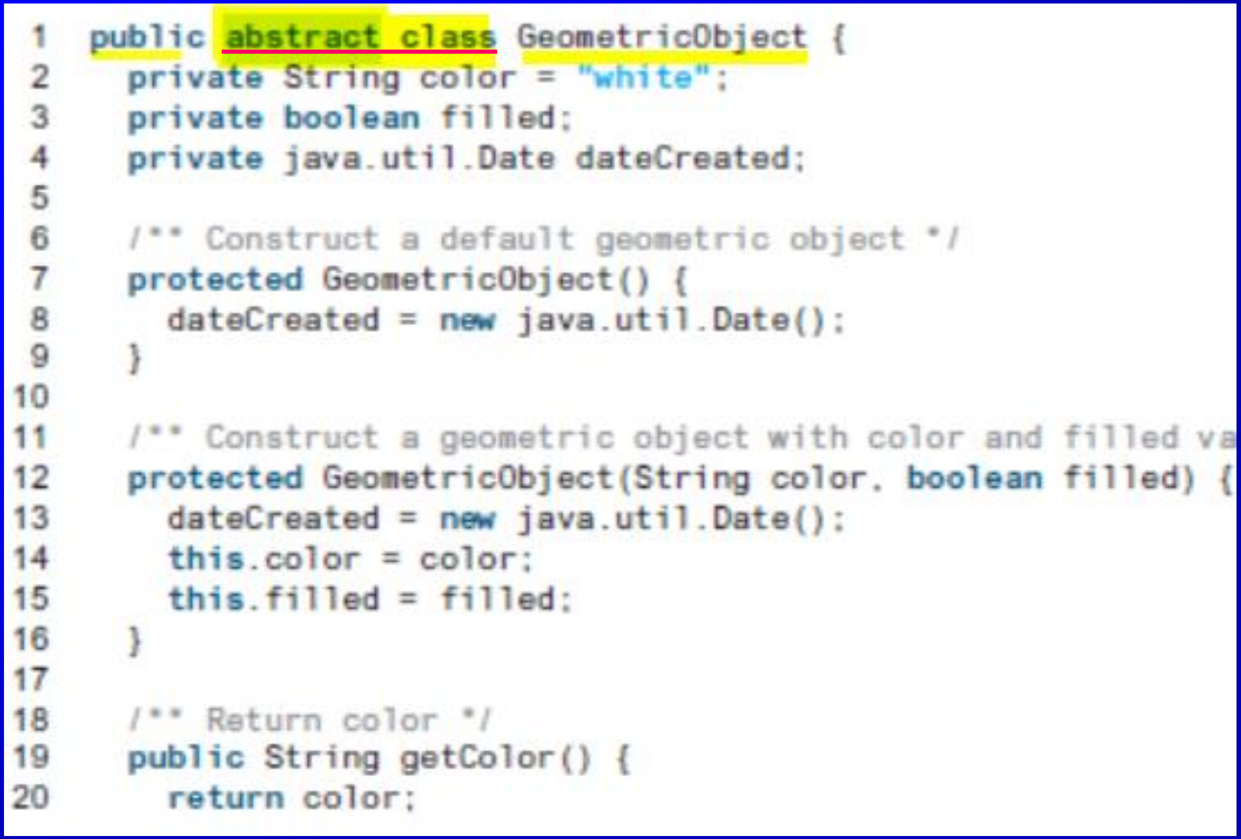
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double

Rectangle

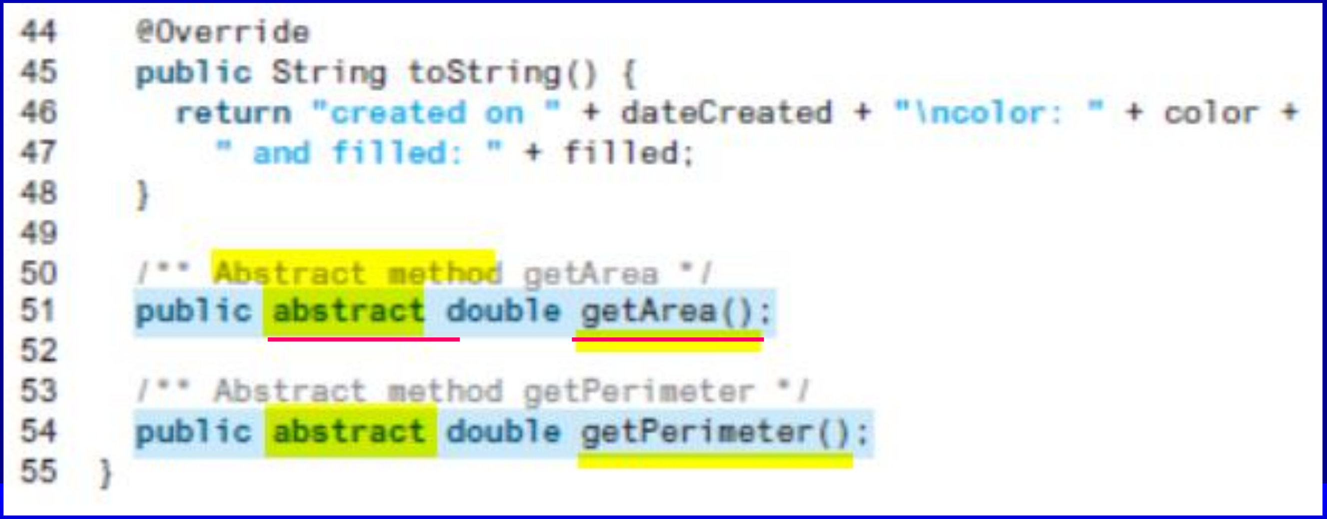
-width: double
-height: double

+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

]

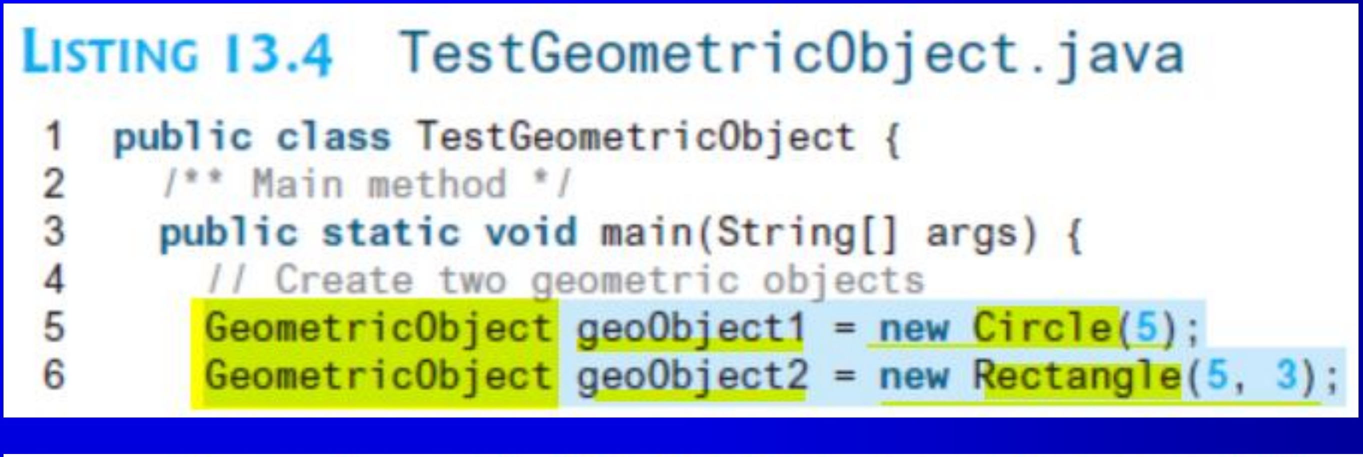


```java
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

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

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

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



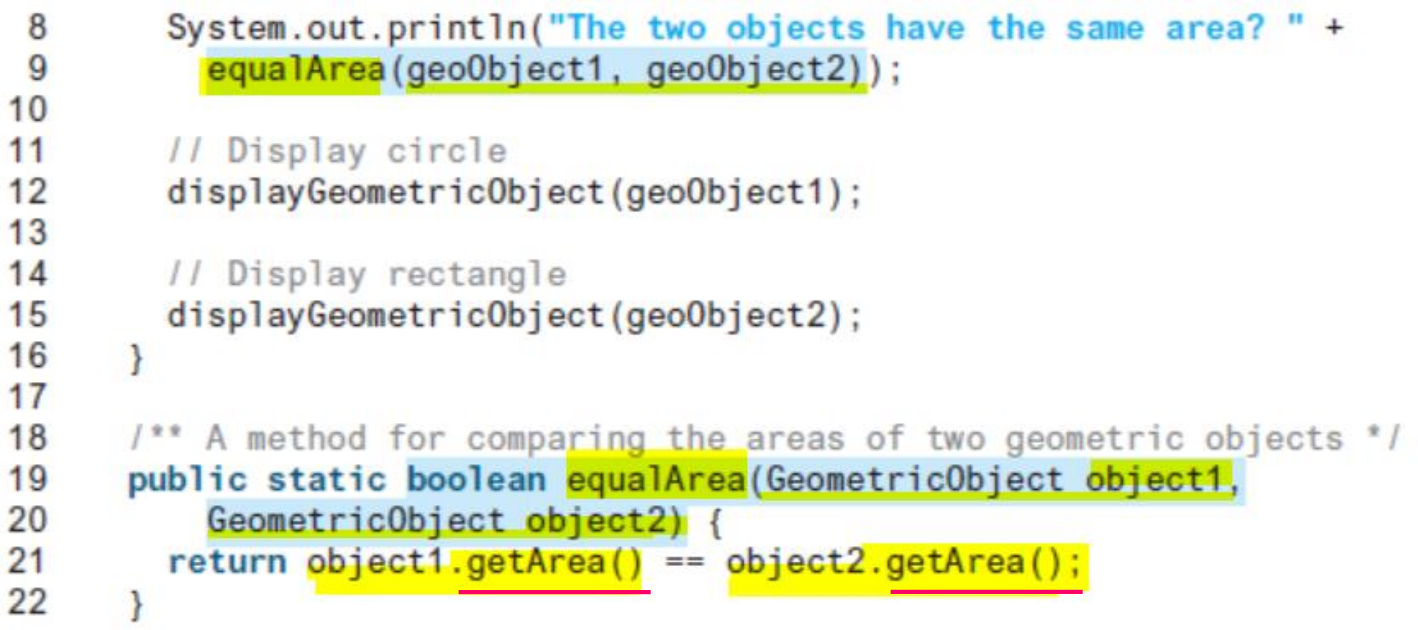
```java
  @Override
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

Why Abstract Methods?



LISTING 13.4 TestGeometricObject.java

```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);
```



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

    // Display circle
    displayGeometricObject(geoObject1);

    // Display rectangle
    displayGeometricObject(geoObject2);
  }

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

- You could not use **equalArea()** for comparing two geometric objects' areas, if the abstract **getArea()** were not defined in **GeometricObject**

Abstract Classe/Method

- **Abstract Method :**
  - only contained/declared in Abstract Class
  - cannot be declared in Non-abstract class.

- Abstract **subclass** (extended from abstract superclass):
  - contain abstract methods (which are not implemented)
- Non-abstract **subclass** (extended from abstract superclass):
  - all the abstract methods must be implemented,

    even if they are not used in the subclass.

Object cannot be created from abstract class

- Abstract class
  - cannot be instantiated using the ***new*** operator
  - but can still have constructors
    - which are invoked by its subclasses' constructors

      e.g.,

      subclass' constructor: Circle() , Rectangle()

      can invoke

      superclass' constructor GeometricObject()

Abstract class without abstract method

- Abstract methods must be in abstract class.
- However, an abstract class maybe contain no abstract methods.
  - In this case, the abstract class is used as a base/super class for defining a new subclass.

Superclass of Abstract class may be concrete

- ☞ A subclass may be abstract even if its superclass is concrete
- ☞ e.g.,
  - **Superclass**: **concrete**
    - ◆ the **Object** class
  - **Subclass**: **abstract**
    - ◆ the **GeometricObject** class

Concrete method overridden to be abstract

- ☞ A subclass can override a method from its superclass to define it abstract.
  - Superclass method: concrete
  - Subclass method (overridden): abstract
- ☞ This is rare, but useful in case:
  - the method implementation in the superclass becomes invalid in the subclass.
  - In this case, the subclass must be defined abstract.

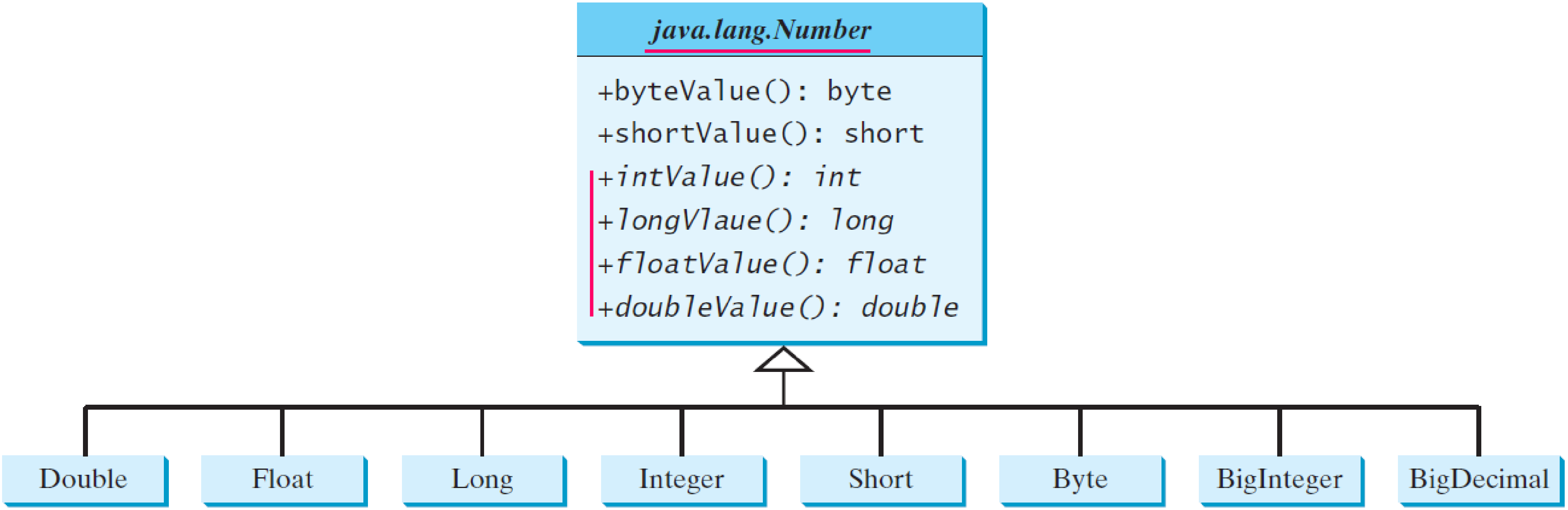
Abstract class as type

- ☞ You cannot create an instance from an abstract class using the new operator

- ☞ but an abstract class can be used as data type.

- ☞ GeometricObject[] geo = new GeometricObject[10];
  - – array elements are of GeometricObject type

Case Study: the Abstract Number Class



java.lang.Number

+byteValue(): byte
+shortValue(): short
+*intValue(): int*
+*longVlaue(): long*
+*floatValue(): float*
+*doubleValue(): double*

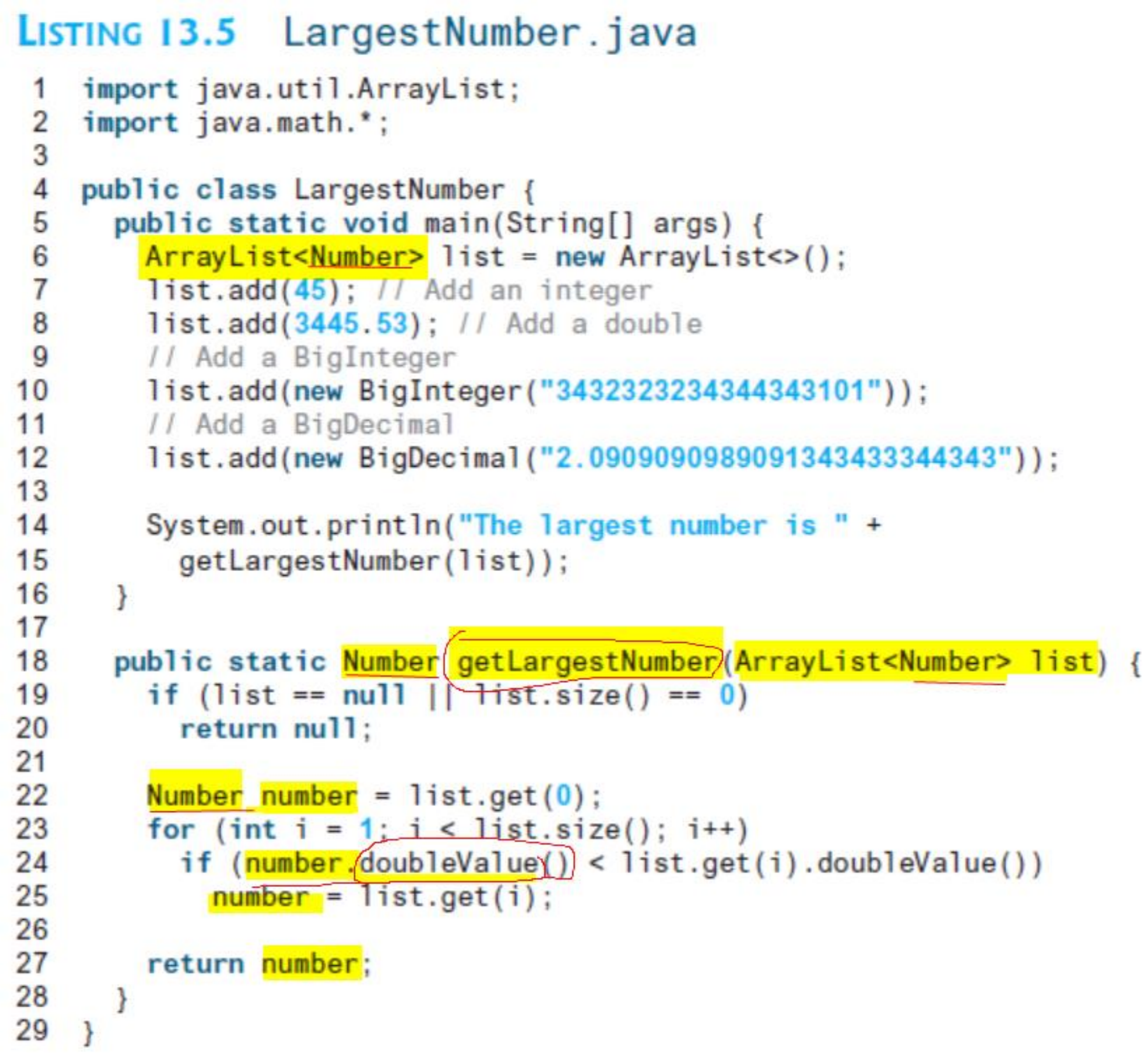
Double Float Long Integer Short Byte BigInteger BigDecimal

in the **Number** class, **intValue(), longValue(), floatValue(), doubleValue()** methods:

- cannot be implemented
- so defined as abstract

Case Study: the Abstract Number Class

With **Number** defined as the superclass, we can define common methods for the subclasses



LISTING 13.5 LargestNumber.java

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

public class LargestNumber {
  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

What is an interface?

Why is an interface useful?

How to define an interface?

How to use an interface?

Interface

☞ **an interface**

- *a class-like construct*
- *for defining **common methods/behaviors** of objects*
- contains only:
  - **constants**
  - **abstract methods**

- similar to an abstract class

Define an Interface

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

Example:

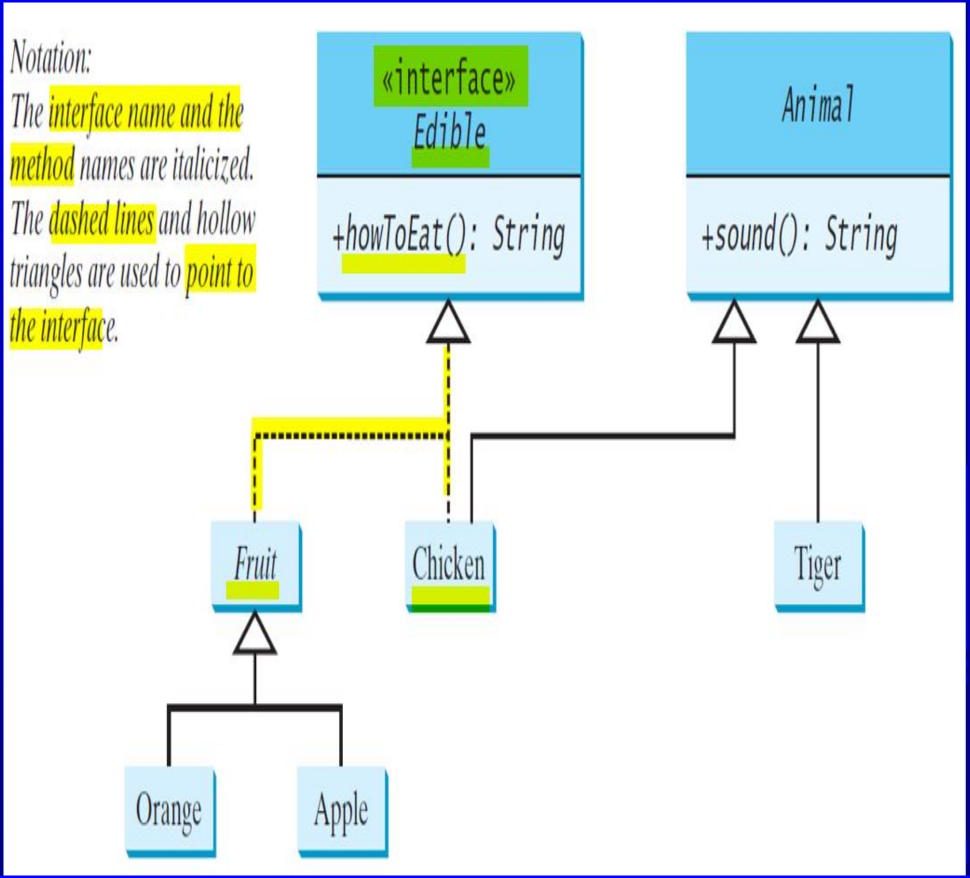
```
public interface Edible {
  public abstract String howToEat();
}
```

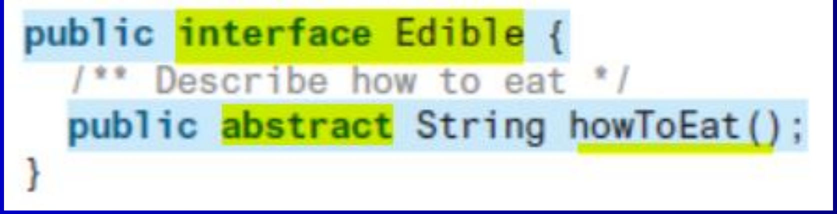
Interface is like a special class

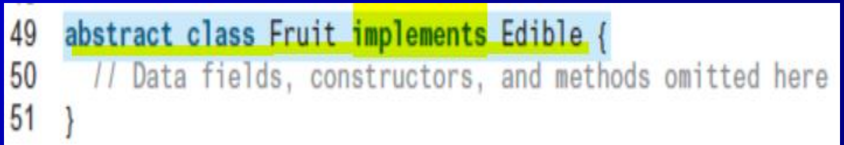
- An interface
  - **compiled** into a separate bytecode file . (like a class)
  - **cannot create an instance** from an interface using the *new* operator. (like an abstract class)
  - can be used as a **data type** for a variable, as the result of casting, etc. (like an abstract class)

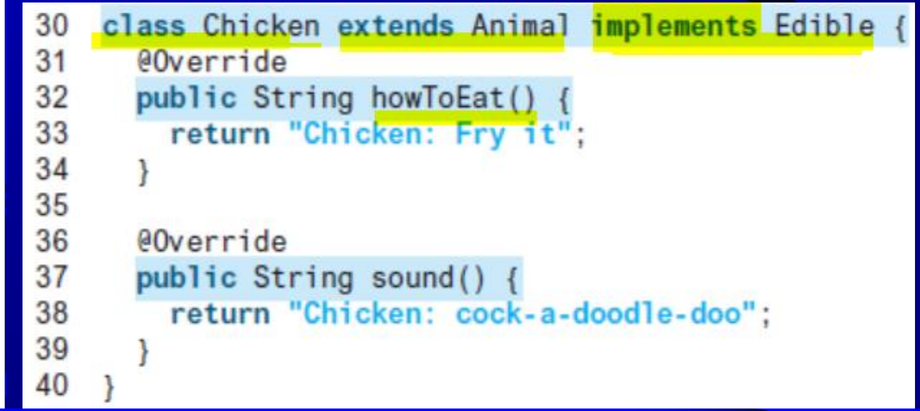
Example

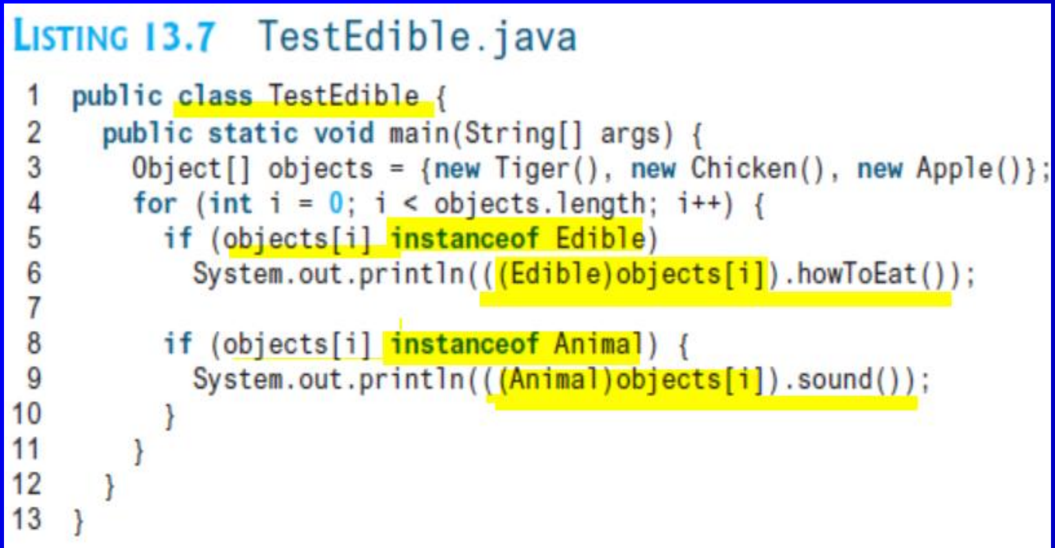
- Use the Edible interface to specify whether an object is edible.
  - The object's class (Chicken, Fruit) implements the (Edible) interface
    - keyword: implements


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


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

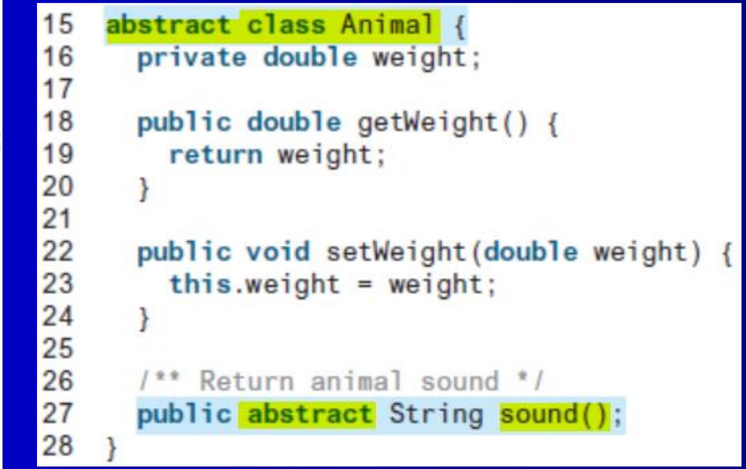

```java
abstract class Fruit implements Edible {
  // Data fields, constructors, and methods omitted here
}
```


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

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

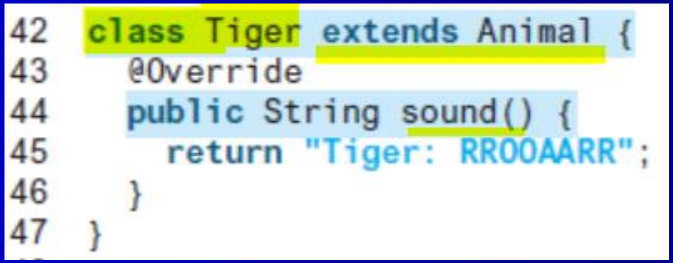


LISTING 13.7 TestEdible.java

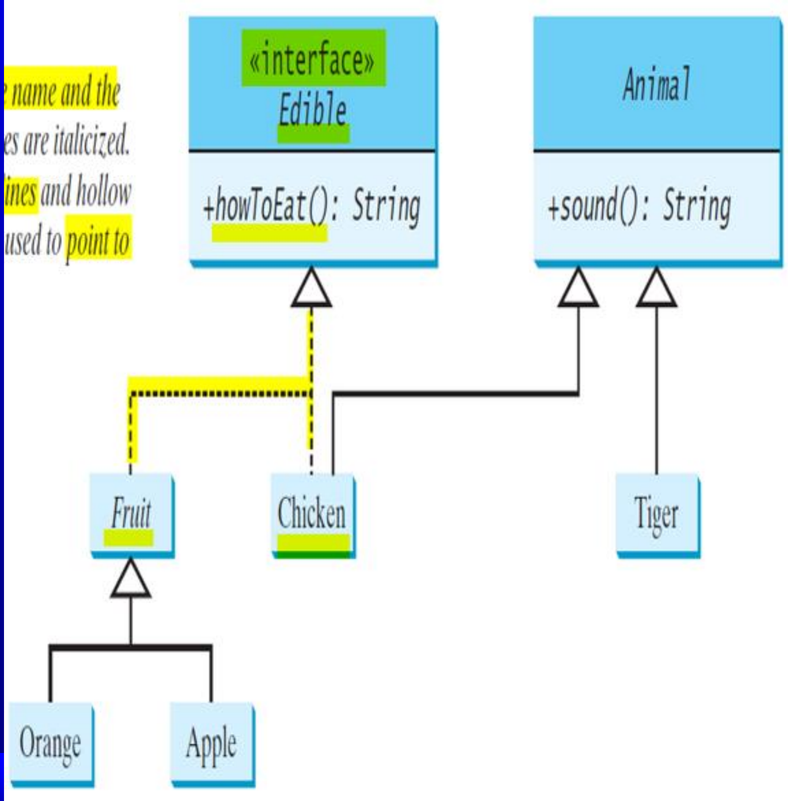
```java
public class TestEdible {
  public static void main(String[] args) {
    Object[] objects = {new Tiger(), new Chicken(), new Apple()};
    for (int i = 0; i < objects.length; i++) {
      if (objects[i] instanceof Edible)
        System.out.println(((Edible)objects[i]).howToEat());

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

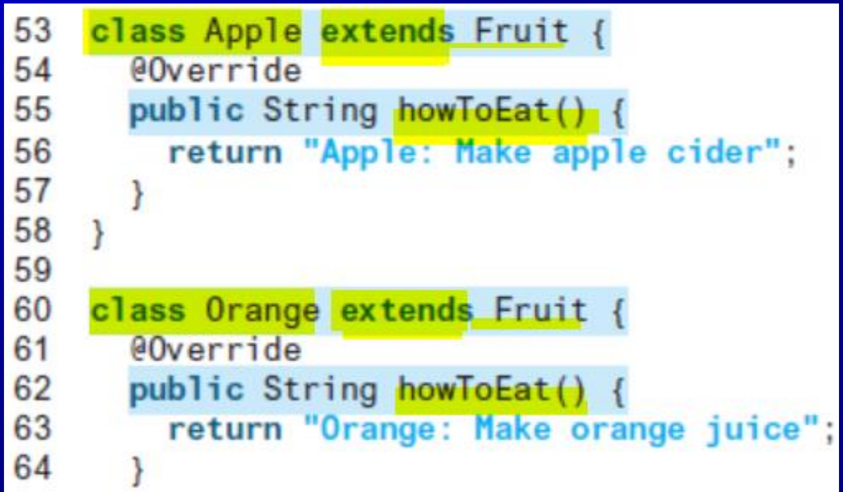


```java
abstract class Animal {
  private double weight;

  public double getWeight() {
    return weight;
  }

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

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



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





```java
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";
  }
```

Omitting Modifiers in Interfaces

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

  public abstract void p();
}
```

Equivalent

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

  void p();
}
```

- In interface
  - modifiers can be **omitted**:
    - Because: all data fields are *public final static* ; all methods are *public abstract*
  - constant can be accessed using:
    - **InterfaceName.CONSTANT_NAME** (e.g., **T1.K**).

Example: the Comparable Interface

```
package java.lang;
public interface Comparable<E> {
   public int compareTo(E o);
}
```

- `compareTo(E o)`:
  - compare this object with the specified object o
  - returns a negative integer, zero, or a positive integer

    if this object is <, ==, or > o.
- *Many classes in Java library implement* Comparable *to compare objects*
  - The classes: **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character**, **BigInteger**, **BigDecimal**, **Calendar**, **String**, and **Date**

Integer and BigInteger Classes

```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
  }
}
```

String and Date Classes

```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
  }
}
```

Example

*Use **compareTo()** method to compare*
two numbers, two strings, and two dates :

```
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));
```

- All numeric wrapper classes and Character class implement/override
  - ***compareTo()*** method

    (declared in the Comparable interface)

- All wrapper classes implement/override
  - ***toString()*, *equals()*, and *hashCode()*** methods

    (defined in the Object class)

- Supertype: interface/superclass

Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object.

All the following expressions are **true**.

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

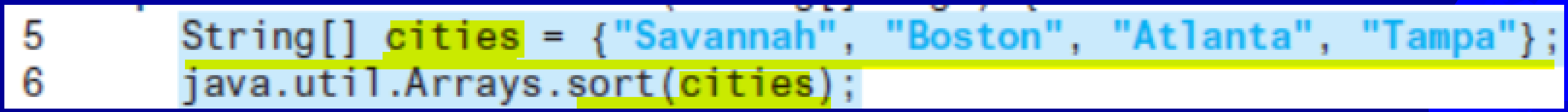
```
s instanceof String
s instanceof Object
s instanceof Comparable
```

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

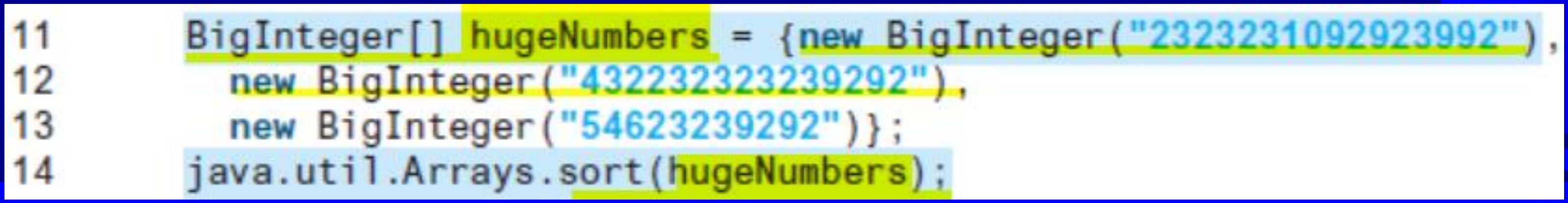
Generic `sort` Method

- The *java.util.Arrays.sort(array)* method:
  - requires that *array elements* are *instances of* the interface *Comparable<E>*.

SortComparableObjects

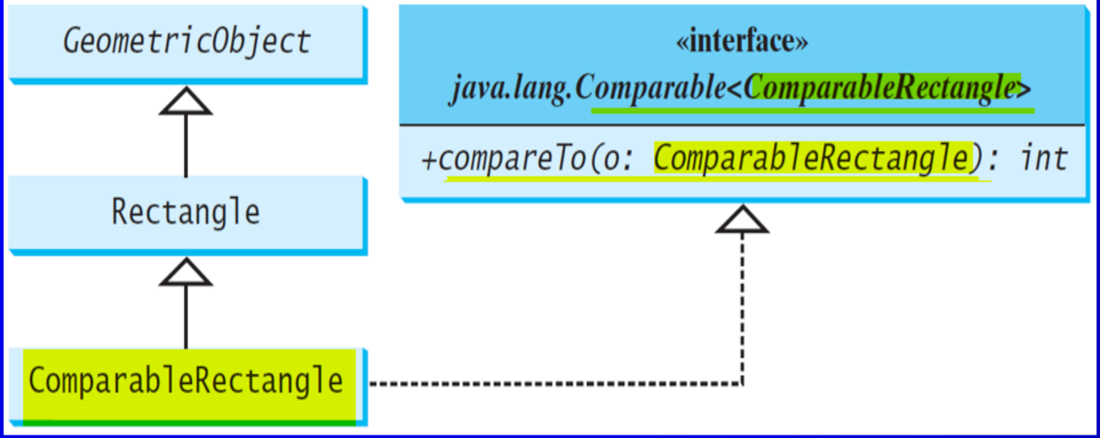


```
String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
java.util.Arrays.sort(cities);
```

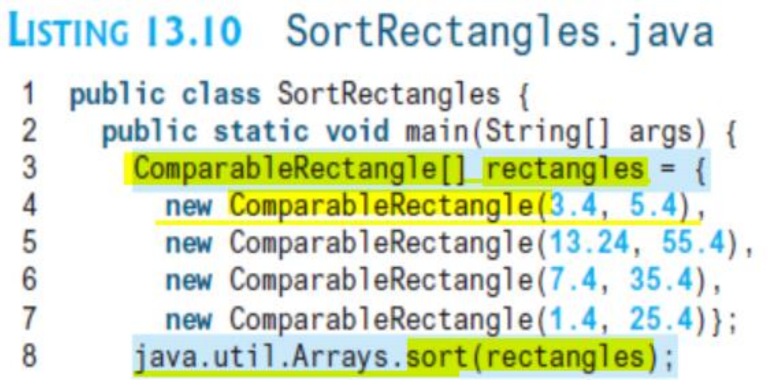


```
BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
  new BigInteger("432232323239292"),
  new BigInteger("54623239292")};
java.util.Arrays.sort(hugeNumbers);
```

Defining Class to Implement Comparable

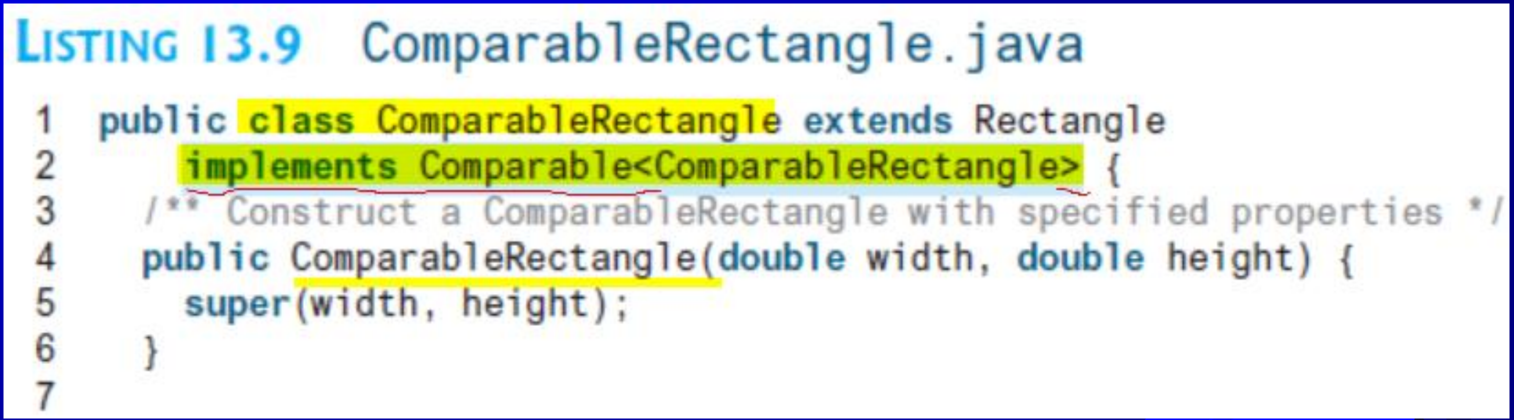


GeometricObject
Rectangle
ComparableRectangle
«interface»
java.lang.Comparable<ComparableRectangle>
+compareTo(o: ComparableRectangle): int



LISTING 13.10 SortRectangles.java

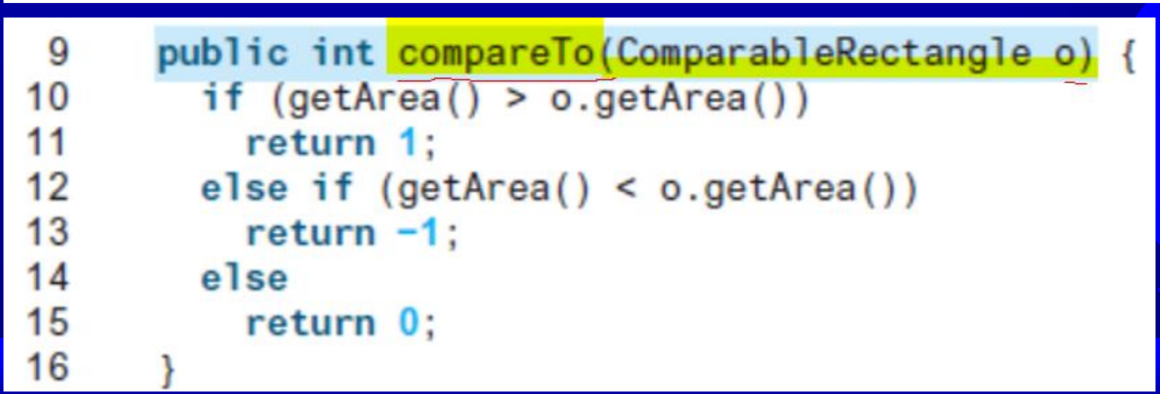
```java
public class SortRectangles {
  public static void main(String[] args) {
    ComparableRectangle[] rectangles = {
      new ComparableRectangle(3.4, 5.4),
      new ComparableRectangle(13.24, 55.4),
      new ComparableRectangle(7.4, 35.4),
      new ComparableRectangle(1.4, 25.4)};
    java.util.Arrays.sort(rectangles);
```



LISTING 13.9 ComparableRectangle.java

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

```

implement compareTo



```java
  public int compareTo(ComparableRectangle o) {
    if (getArea() > o.getArea())
      return 1;
    else if (getArea() < o.getArea())
      return -1;
    else
      return 0;
  }
```

The Cloneable Interface

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

- is empty interface: “marker Interface”
- to specify that an object can be cloned/copied
  - A **class** that implements the Cloneable interface:
    - is marked cloneable
    - its **objects** can be cloned using: clone() method in Object class

Examples

Many classes (e.g., Date, Calendar) in the Java library implement Cloneable. Thus, their instances can be cloned.

For example,

```
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar) calendar.clone();
System.out.println("calendar == calendarCopy is " +
  (calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " +
  calendar.equals(calendarCopy));
```

displays

```
calendar == calendarCopy is false
calendar.equals(calendarCopy) is true
```

Implementing Cloneable Interface

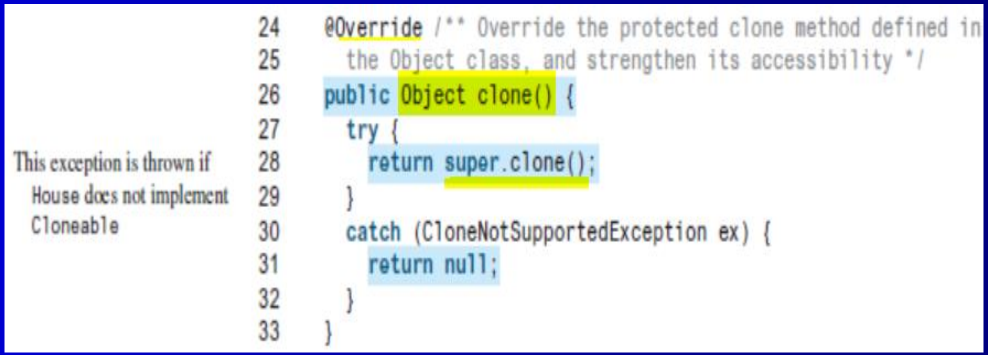
- ☞ A class implementing the Cloneable interface must override the clone() method in the Object class.

- ☞ Example (Book): Listing 13.11
  - a class named House implements Cloneable and Comparable

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

House house2 = (House)house1.clone();

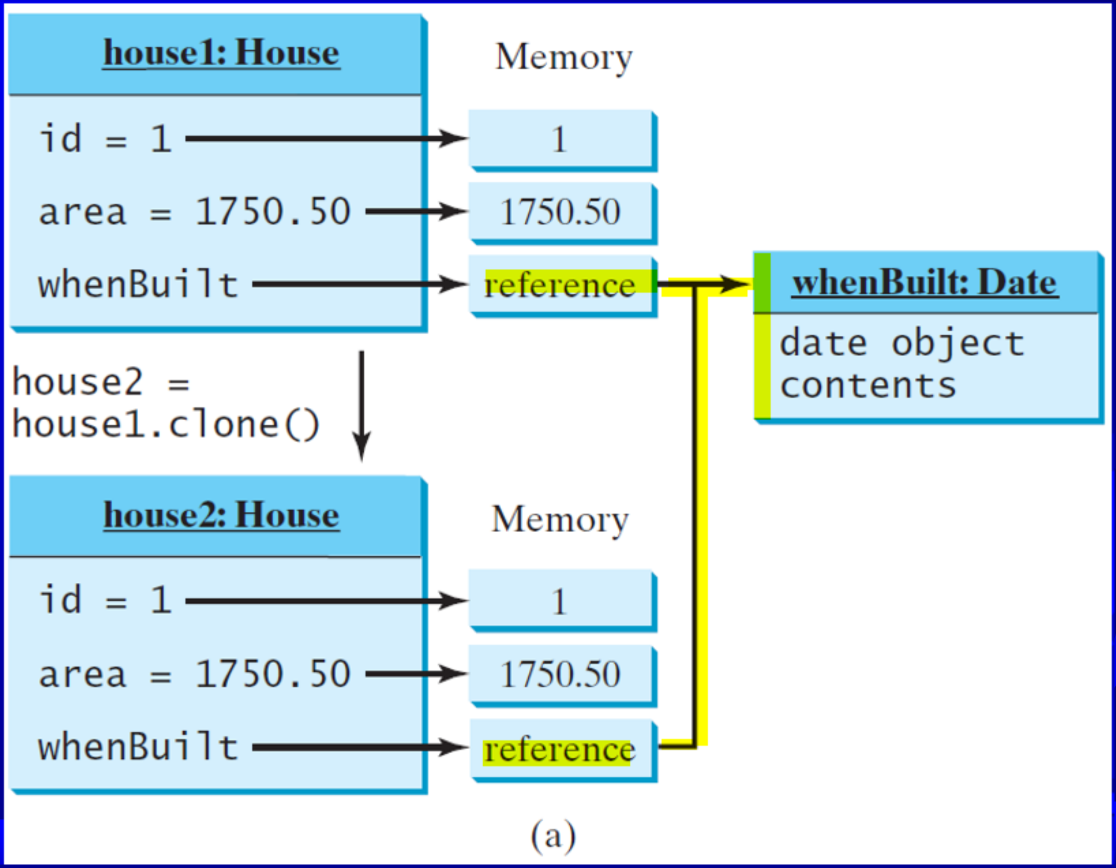


This exception is thrown if House does not implement Cloneable

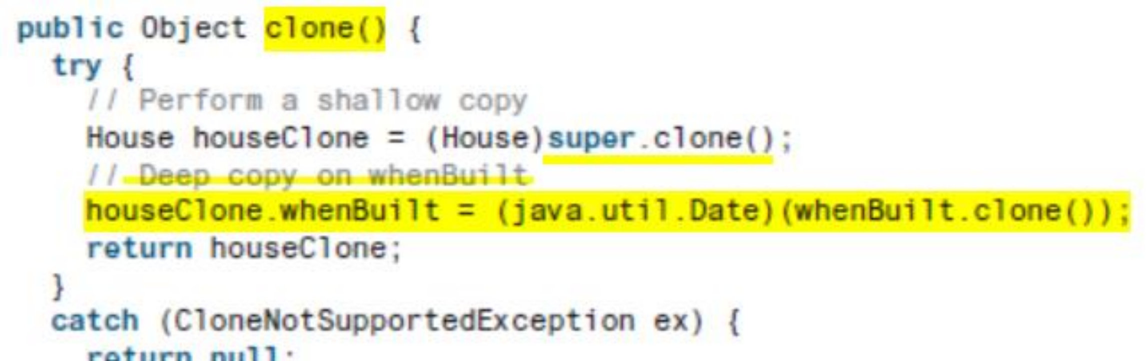
```
@Override /** Override the protected clone method defined in
  the Object class, and strengthen its accessibility */
public Object clone() {
  try {
    return super.clone();
  }
  catch (CloneNotSupportedException ex) {
    return null;
  }
}
```

Shallow Copy

the reference is copied

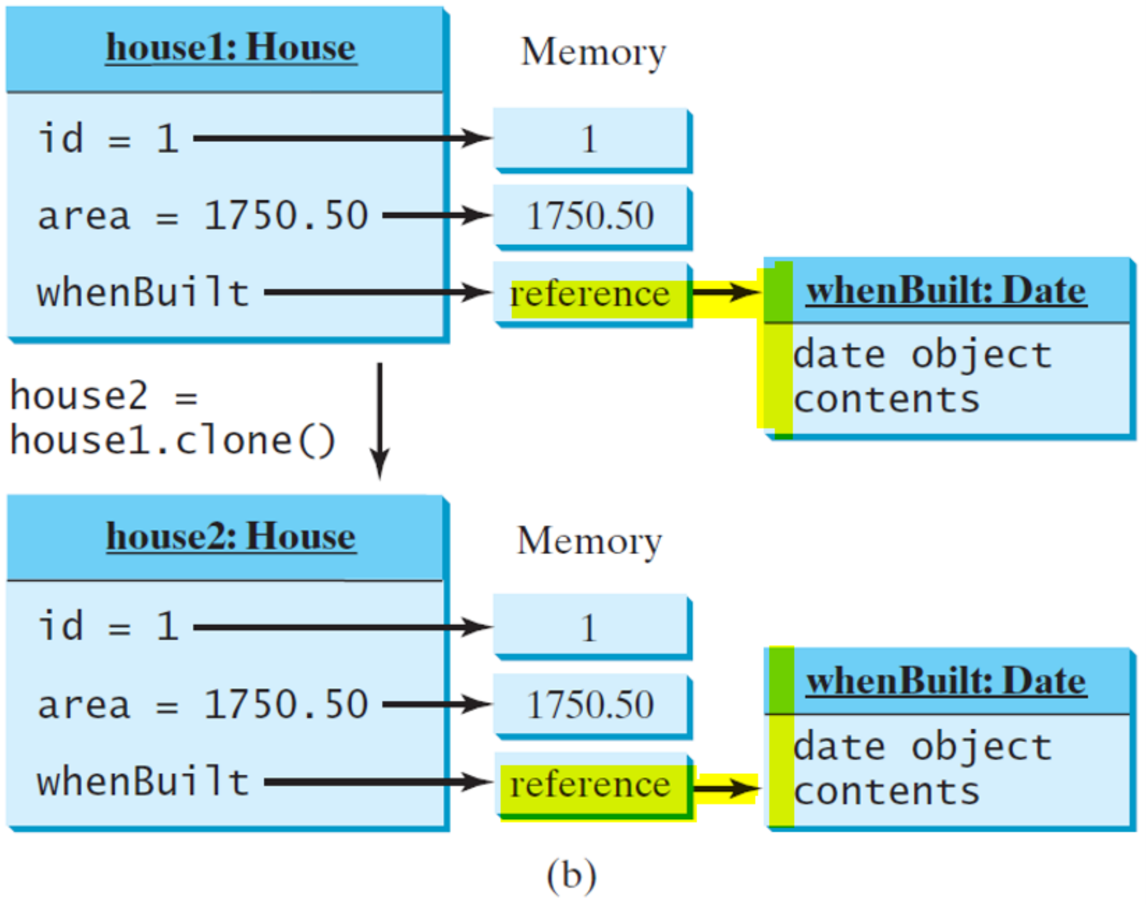


Shallow vs. Deep Copy



```
public Object clone() {
  try {
    // Perform a shallow copy
    House houseClone = (House)super.clone();
    // Deep copy on whenBuilt
    houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
    return houseClone;
  }
  catch (CloneNotSupportedException ex) {
    return null;
  }
}
```

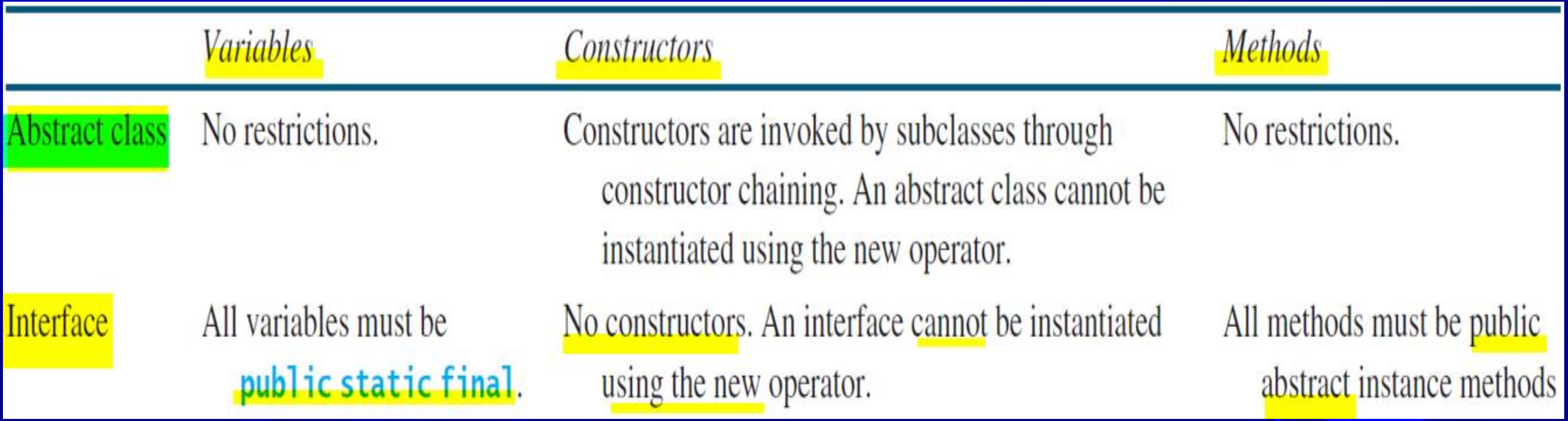
Deep Copy



*New/coloned **Date** object is created*

Interfaces vs. Abstract Classes

- Abstract class:
  - can have all types of data.
  - can have concrete methods.
- In an **Interface**:
  - data must be constants;
  - only abstract method (signature without implementation)



	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

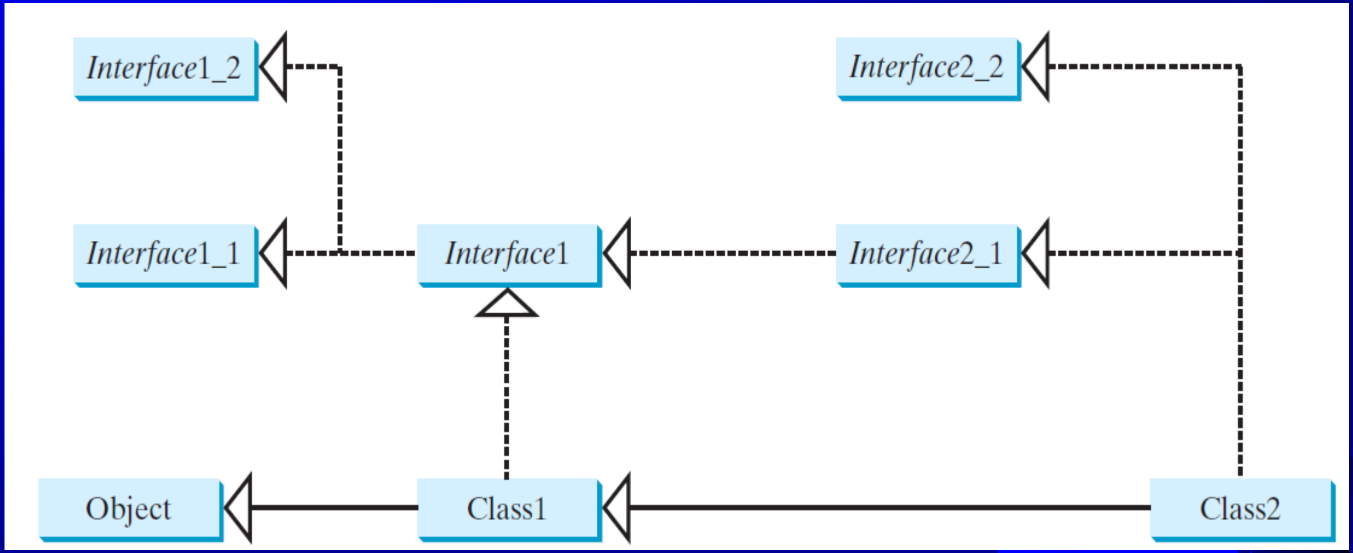
Interfaces vs. Abstract Classes

– All classes share a single root, the Object class

☞ but interfaces have no single root

– If c is an instance of *Class2*.

◆ c is also an instance of *Object*, *Class1*, *Interface1*, *Interface1_1*, *Interface1_2*, *Interface2_1*, *Interface2_2*.



– A Class/Interface can be used as a data type.

☞ A variable of interface type: can reference instance of the class that implements the interface.

☞ If *Class1* extends *Interface1*, *Interface1* is like a *superclass* of *Class1*.

Caution: conflict interfaces

– One class implements two **conflict interfaces**

for example,

- two same constants have : different values
- two same methods (with same signature) have : different return type.

– This will cause **compilation error**.

Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features. How do you decide whether to use an interface or a class? In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface.

When to use interface/class?

- both model **common features**.
- **AbstracClass:** for strong parent-child relationship
  - e.g., a staff member is a person.
- **Interface:** for weak is-kind-of relationship
  - e.g., , all strings are comparable, so the String class implements the Comparable interface.
- **Interface:** for multiple inheritance (multiple supertypes)
  - one as a superclass, and others as interfaces.

Designing a Class

- (Coherence) A class should describe a single entity
  - Do’not combine students and staff in the same class, because they have different entities.
- (Separating responsibilities) A single entity with too many responsibilities can be broken.
  - The String class deals with immutable strings, the StringBuilder class is for creating mutable strings.
- Provide a public no-arg constructor and override the equals() and toString() method defined in the Object class.
- Always provide a constructor and initialize variables to avoid programming errors.