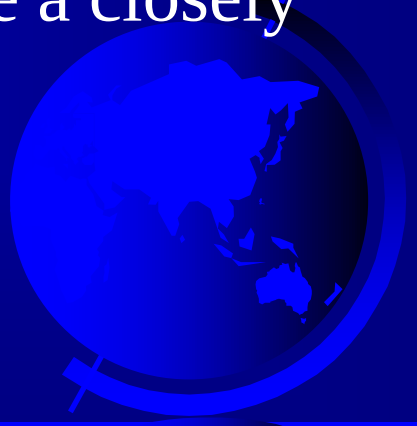
Chapter 13 Abstract Classes and Interfaces

Motivations

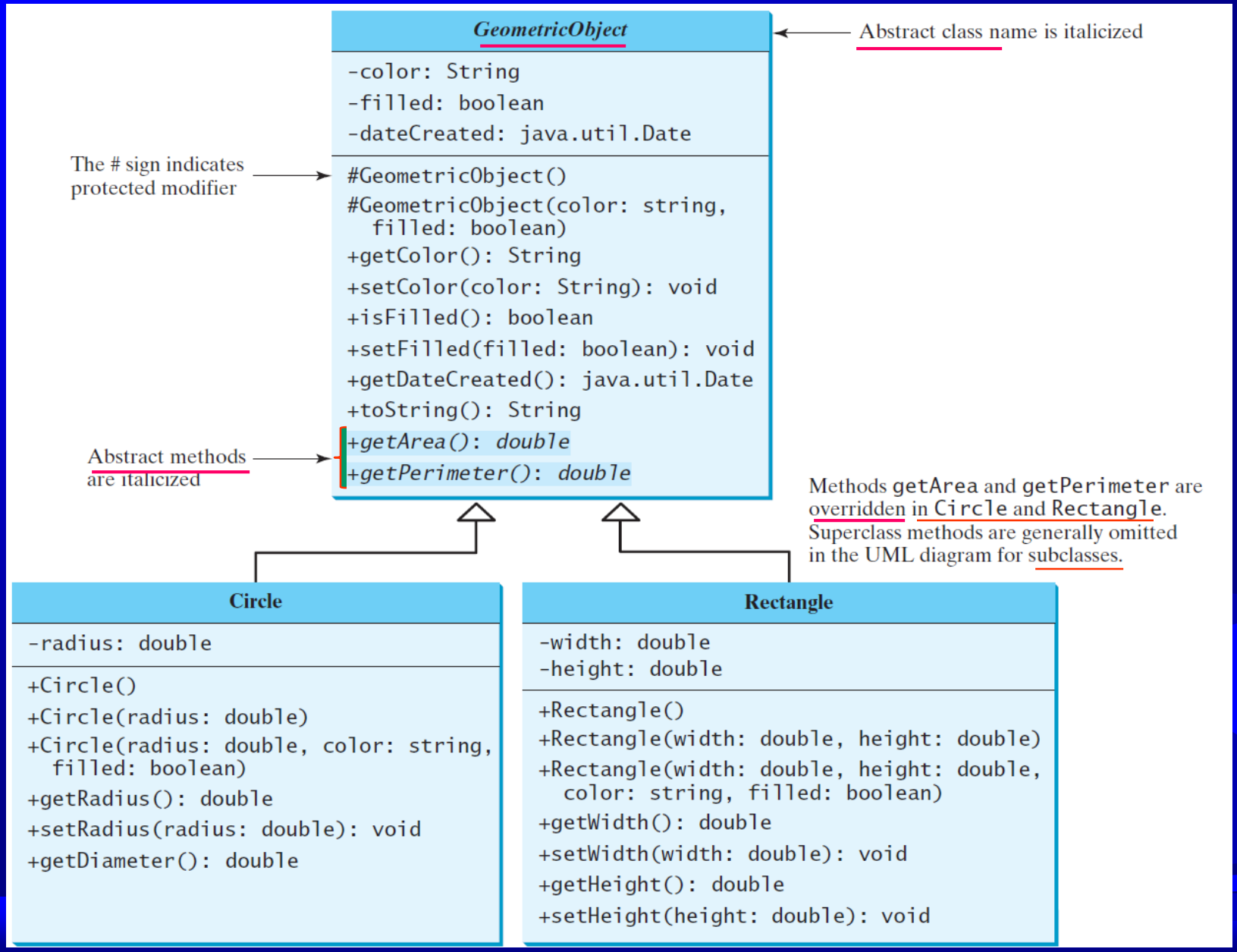
- You have learned how to write simple programs to create and display GUI components. But how to respond to user actions, such as clicking a button to perform an action?
- An **interface** is for defining common behavior for classes (including unrelated classes).
  - Before discussing interfaces, we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



```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

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

```
8      System.out.println("The two objects have the same area? " +
9        equalArea(geoObject1, geoObject2));
10
11     // Display circle
12     displayGeometricObject(geoObject1);
13
14     // Display rectangle
15     displayGeometricObject(geoObject2);
16   }
17
18   /** A method for comparing the areas of two geometric objects */
19   public static boolean equalArea(GeometricObject object1,
20       GeometricObject object2) {
21     return object1.getArea() == object2.getArea();
22   }
```

- 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

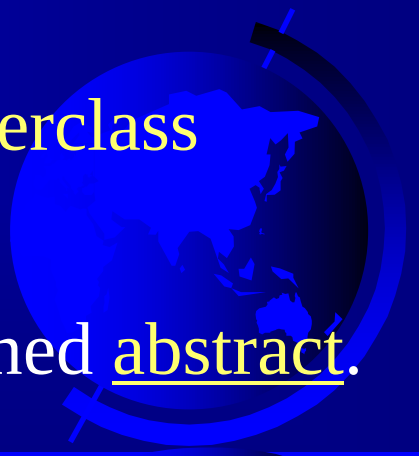
- Abstract methods must be in abstract class.
- However, an abstract class maybe contain no abstract methods.
  - In this case, the abstract class is used as a base/super class for defining a new subclass.

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.

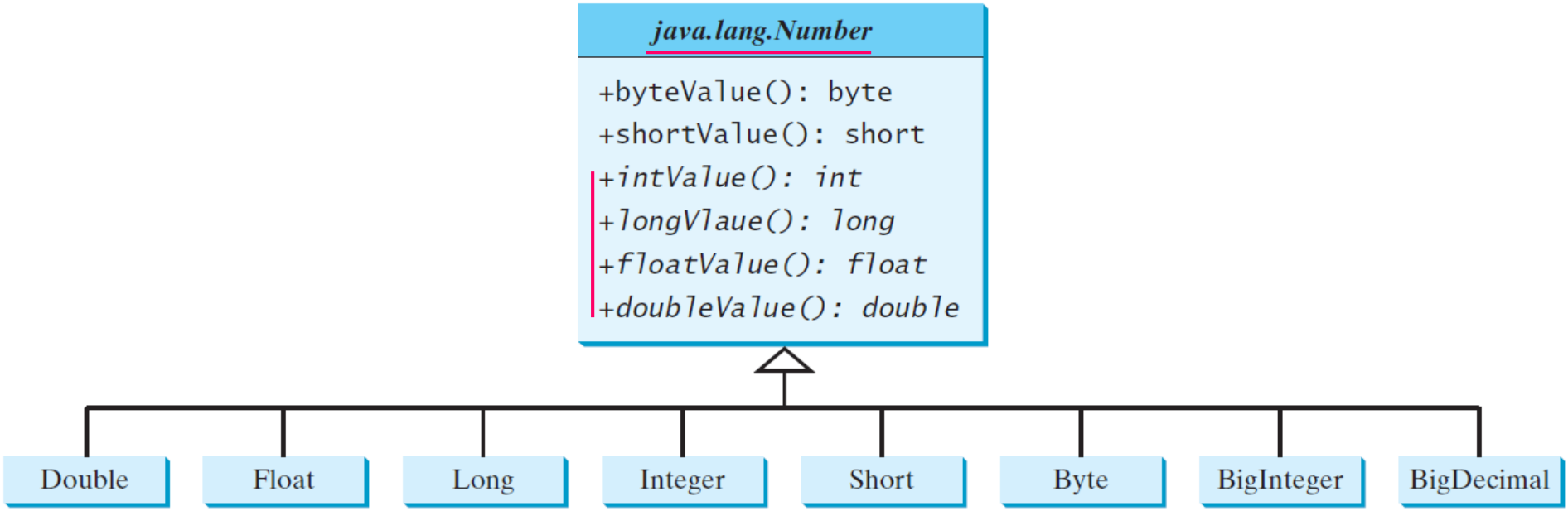


Abstract class as type

- 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



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

```
1  import java.util.ArrayList;
2  import java.math.*;
3
4  public class LargestNumber {
5    public static void main(String[] args) {
6      ArrayList<Number> list = new ArrayList<>();
7      list.add(45); // Add an integer
8      list.add(3445.53); // Add a double
9      // Add a BigInteger
10     list.add(new BigInteger("34323232344344343101"));
11     // Add a BigDecimal
12     list.add(new BigDecimal("2.0909090989091343433344343"));
13
14     System.out.println("The largest number is " +
15       getLargestNumber(list));
16   }
17
18   public static Number getLargestNumber(ArrayList<Number> list) {
19     if (list == null || list.size() == 0)
20       return null;
21
22     Number number = list.get(0);
23     for (int i = 1; i < list.size(); i++)
24       if (number.doubleValue() < list.get(i).doubleValue())
25         number = list.get(i);
26
27     return number;
28   }
29 }
```

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:

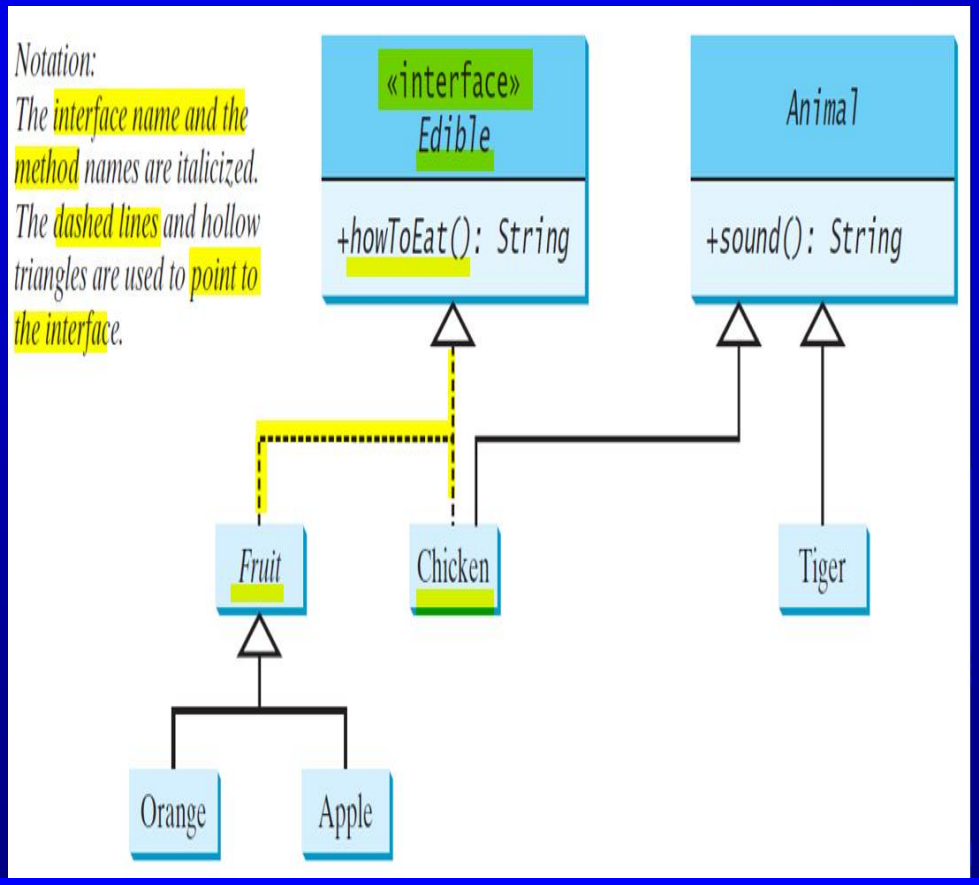
```
public interface Edible {
  public abstract String howToEat();
}
```

Interface is like a special class

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




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

```
49 abstract class Fruit implements Edible {
50   // Data fields, constructors, and methods omitted here
51 }
```

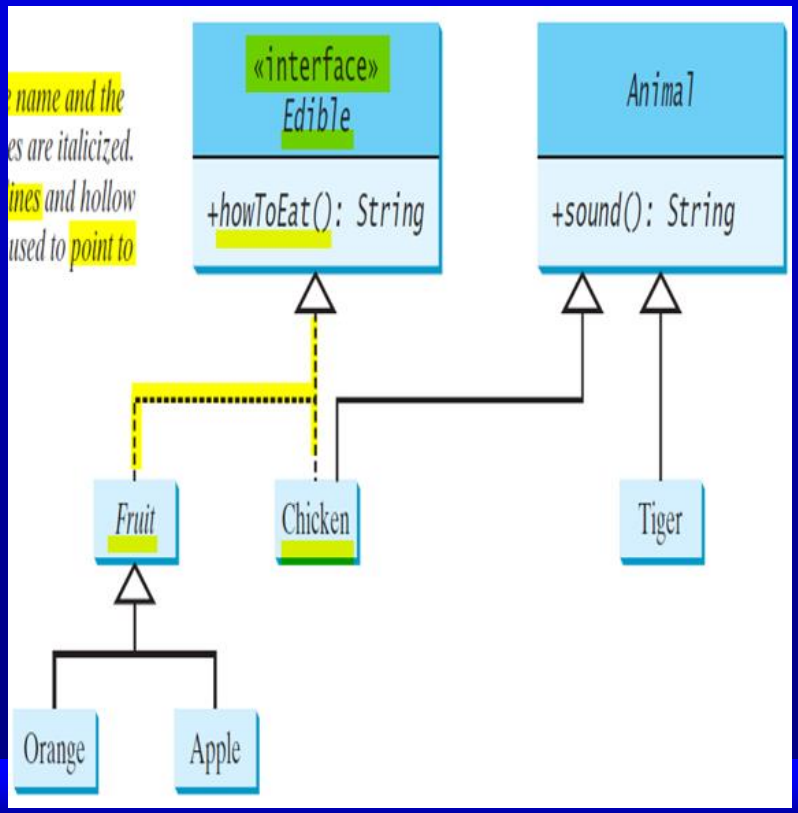
```
30 class Chicken extends Animal implements Edible {
31   @Override
32   public String howToEat() {
33     return "Chicken: Fry it";
34   }
35
36   @Override
37   public String sound() {
38     return "Chicken: cock-a-doodle-doo";
39   }
40 }
```

LISTING 13.7 TestEdible.java

```
1  public class TestEdible {
2    public static void main(String[] args) {
3      Object[] objects = {new Tiger(), new Chicken(), new Apple()};
4      for (int i = 0; i < objects.length; i++) {
5        if (objects[i] instanceof Edible)
6          System.out.println(((Edible)objects[i]).howToEat());
7
8        if (objects[i] instanceof Animal) {
9          System.out.println(((Animal)objects[i]).sound());
10       }
11     }
12   }
13 }
```

```
15 abstract class Animal {
16   private double weight;
17
18   public double getWeight() {
19     return weight;
20   }
21
22   public void setWeight(double weight) {
23     this.weight = weight;
24   }
25
26   /** Return animal sound */
27   public abstract String sound();
28 }
```

```
42 class Tiger extends Animal {
43   @Override
44   public String sound() {
45     return "Tiger: RROOAARR";
46   }
47 }
```




```
53 class Apple extends Fruit {
54   @Override
55   public String howToEat() {
56     return "Apple: Make apple cider";
57   }
58 }
59
60 class Orange extends Fruit {
61   @Override
62   public String howToEat() {
63     return "Orange: Make orange juice";
64   }
```

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, Date***

Integer, BigInteger Classes

```
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
  }
}
```

String, Date Classes

```
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
  }
}
```

Example

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

```
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));
```

- 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

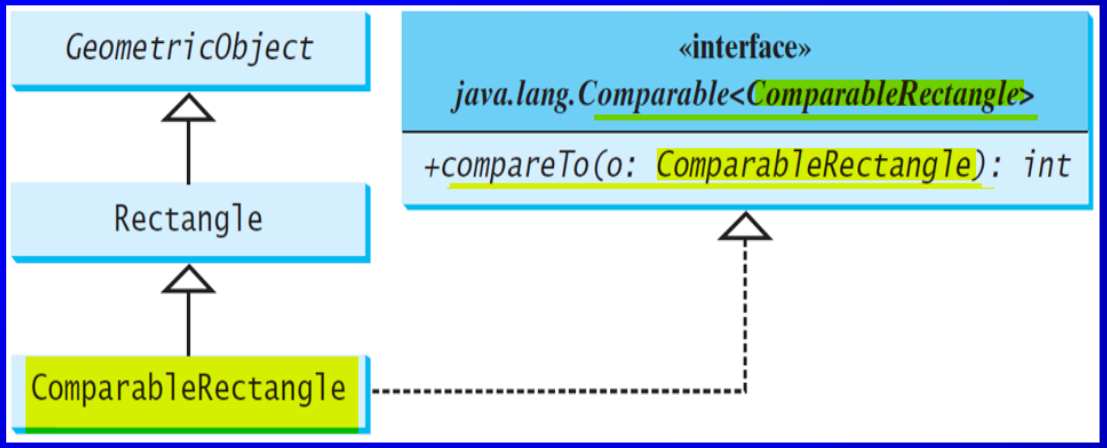
- The *java.util.Arrays.sort(array)* method:
  - requires that *array elements* are *instances of* the interface *Comparable<E>*.

SortComparableObjects

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

```
11    BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
12      new BigInteger("4322323239292"),
13      new BigInteger("54623239292")};
14    java.util.Arrays.sort(hugeNumbers);
```

Defining Class to Implement Comparable



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);
```

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
```

implement compareTo

```
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     }
```

The `Cloneable` Interface

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

- is empty interface: "marker Interface"
  - not contain constants or methods
- 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);

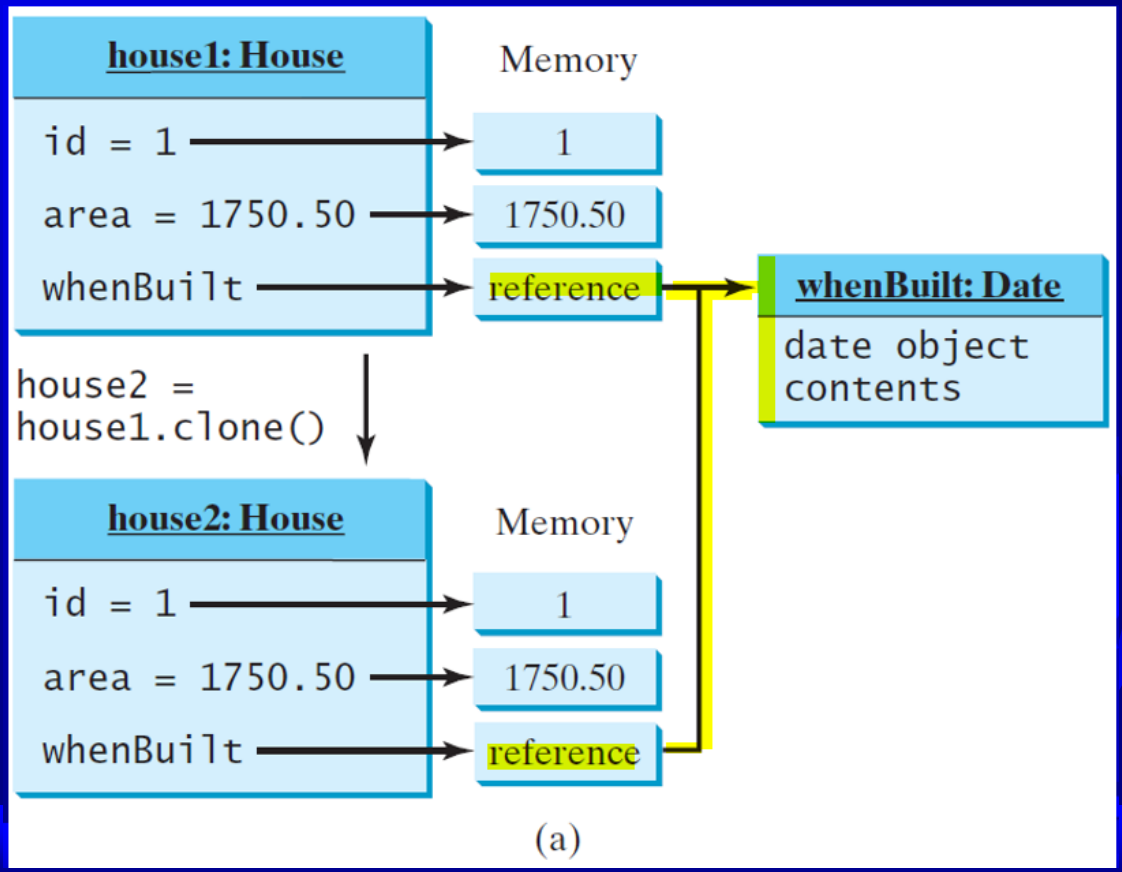
House house2 = (House)house1.clone();

```
24  @Override /** Override the protected clone method defined in
25    the Object class, and strengthen its accessibility */
26  public Object clone() {
27    try {
28      return super.clone();
29    }
30    catch (CloneNotSupportedException ex) {
31      return null;
32    }
33  }
```

This exception is thrown if House does not implement Cloneable

Shallow Copy

the reference is copied

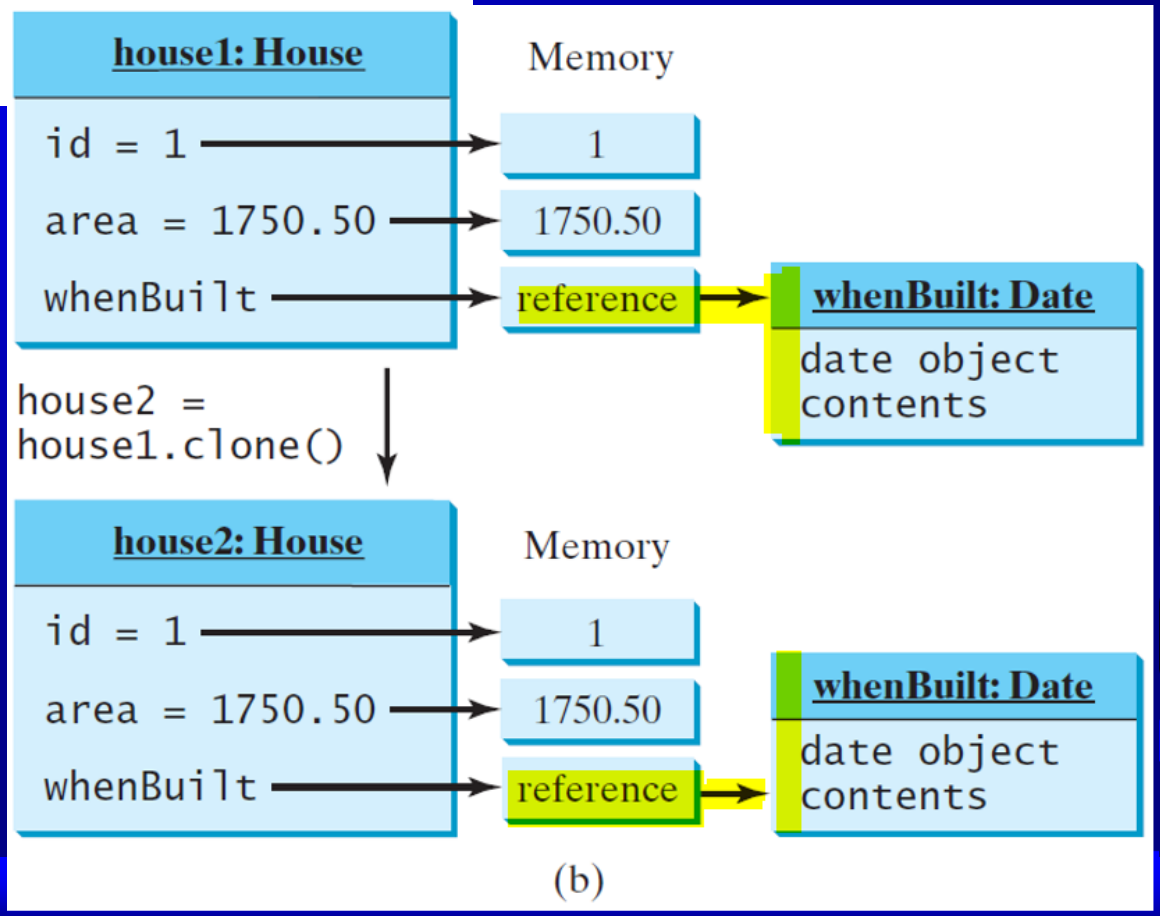


Shallow vs. Deep Copy

```
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*



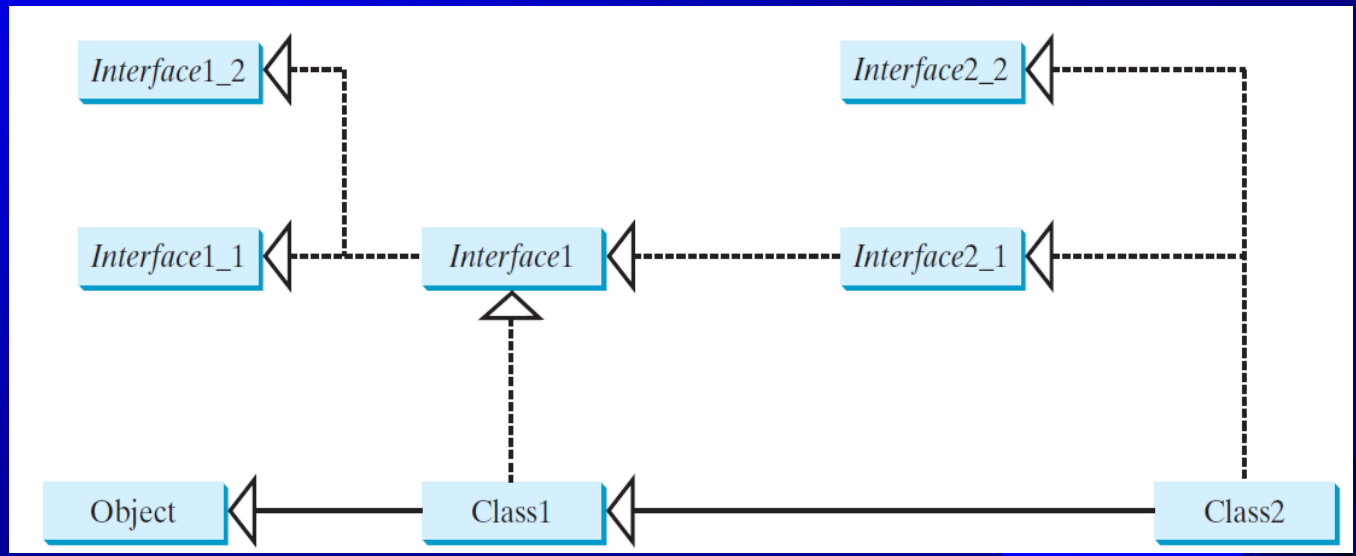
Interfaces vs. Abstract Classes

- 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.
  - eg. *Class1* implements *Interface1*, *Interface1* is like a *superclass* of *Class1*.
  - An instance of interface : can reference instance of the class that implements the interface.

Caution: conflict interfaces

- One class implements two **conflict interfaces**

  for example,

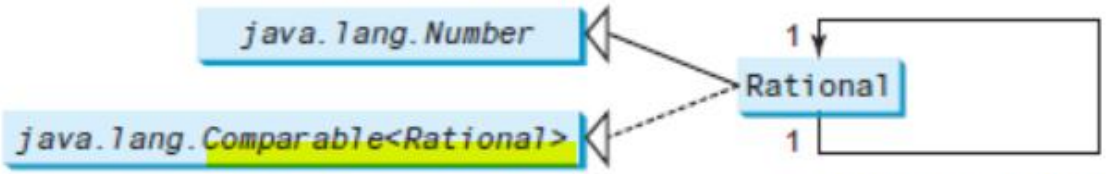
  - two same constants have : different values
  - two same methods (with same signature) have : different return type.

- This will cause **compilation error**.

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.

The `Rational` Class



Rational

```
-numerator: long
-denominator: long

+Rational()
+Rational(numerator: long,
  denominator: long)
+getNumerator(): long
+getDenominator(): long
+add(secondRational: Rational):
  Rational
+subtract(secondRational:
  Rational): Rational
+multiply(secondRational:
  Rational): Rational
+divide(secondRational:
  Rational): Rational
+toString(): String

-gcd(n: long, d: long): long
```

`numerator`	The numerator of this rational number.
`denominator`	The denominator of this rational number.
`Rational()`	Creates a rational number with numerator 0 and denominator 1.
`Rational(numerator, denominator)`	Creates a rational number with a specified numerator and denominator.
`getNumerator()`	Returns the numerator of this rational number.
`getDenominator()`	Returns the denominator of this rational number.
`add(secondRational)`	Returns the addition of this rational number with another.
`subtract(secondRational)`	Returns the subtraction of this rational number with another.
`multiply(secondRational)`	Returns the multiplication of this rational number with another.
`divide(secondRational)`	Returns the division of this rational number with another.
`toString()`	Returns a string in the form "numerator/denominator." Returns the numerator if denominator is 1.
`gcd(n, d)`	Returns the greatest common divisor of n and d.

Rational

TestRationalClass