Abstract Classes and Interfaces

Why abstract class ?

* Class design should ensure that,

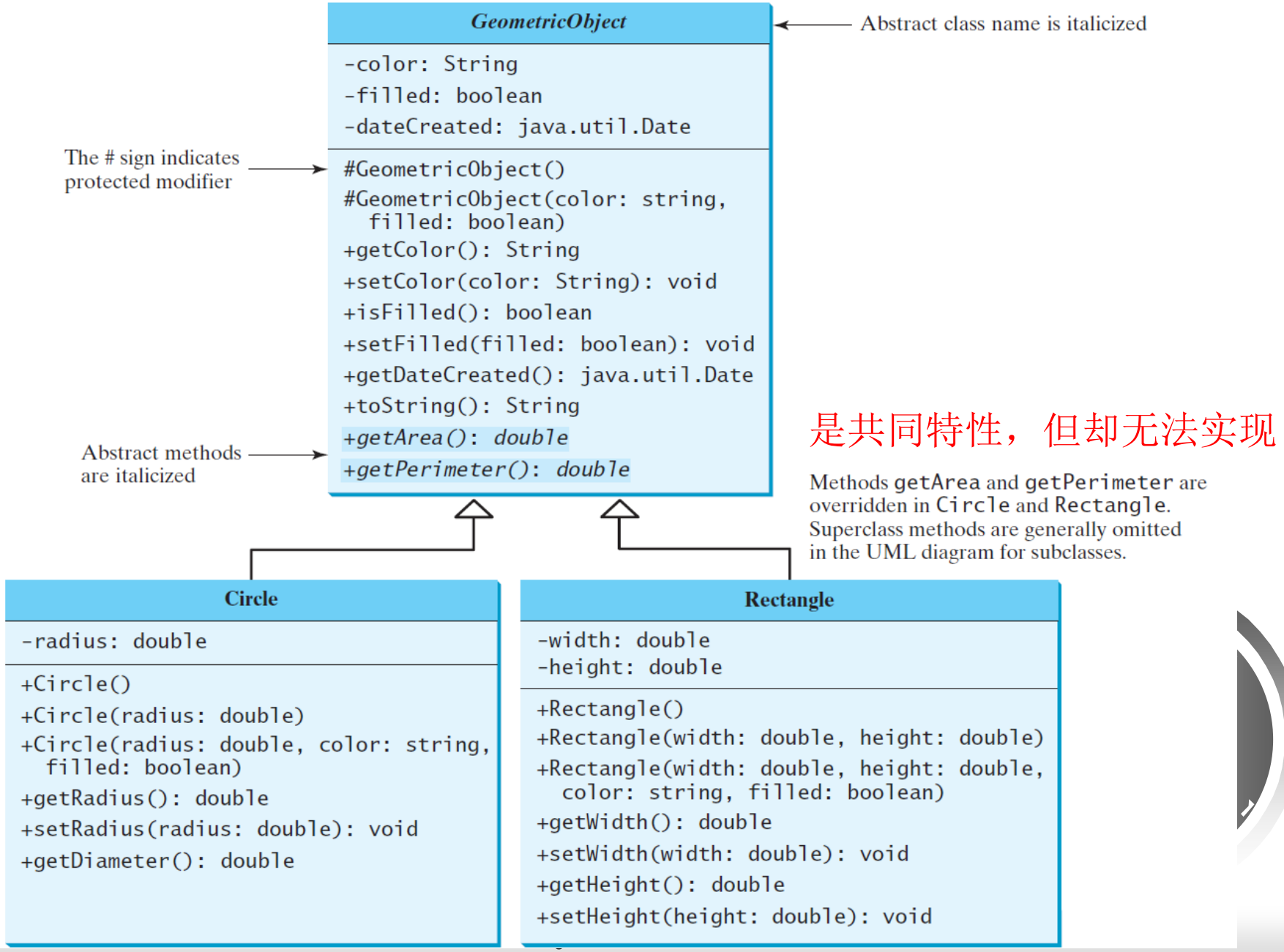
  - a superclass contains common features of its subclasses.

* abstract class:

  - Sometimes a superclass is so abstract that it cannot be used to create any specific instances.

  - Such a class is referred to as an *abstract class*.

Abstract Classes and Abstract Methods



```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 value */
  protected GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

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

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

  /** Return filled. Since filled is boolean,
   *  the get method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** Get dateCreated */
  public java.util.Date getDateCreated() {
    return dateCreated;
  }

  @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();
}
```

```java
public class Circle extends GeometricObject {
  // Same as lines 3-48 in Listing 11.2, so omitted
}
```

```java
public class Rectangle extends GeometricObject {
  // Same as lines 3-51 in Listing 11.3, so omitted
}
```

Explanation

* GeometricObject models common features.

  - Both Circle and Rectangle contain the *getArea()* and *getPerimeter()*.

  - Since you can compute areas and perimeters for *all* geometric objects, it is better to define the getArea() and getPerimeter() methods in the *GeometricObject* class.

* However, these methods cannot be implemented in the GeometricObject class, because their implementation depends on the specific type of geometric object.

  - Such methods are referred to as *abstract methods* and are denoted using the *abstract* modifier in the method header.

abstract method in abstract class

* An abstract method cannot be contained in a nonabstract class.

* If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract.

object cannot be created from abstract class

* An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses.

abstract class without abstract method

* it is possible to define an abstract class that contains no abstract methods.

* In this case, you cannot create instances of the class using the new operator.

so

* A subclass can be abstract even if its superclass is concrete.

* A subclass can override a method from its superclass to define it abstract.

abstract class as type

* an abstract class can be used as a data type.

- Therefore, the following statement is correct.

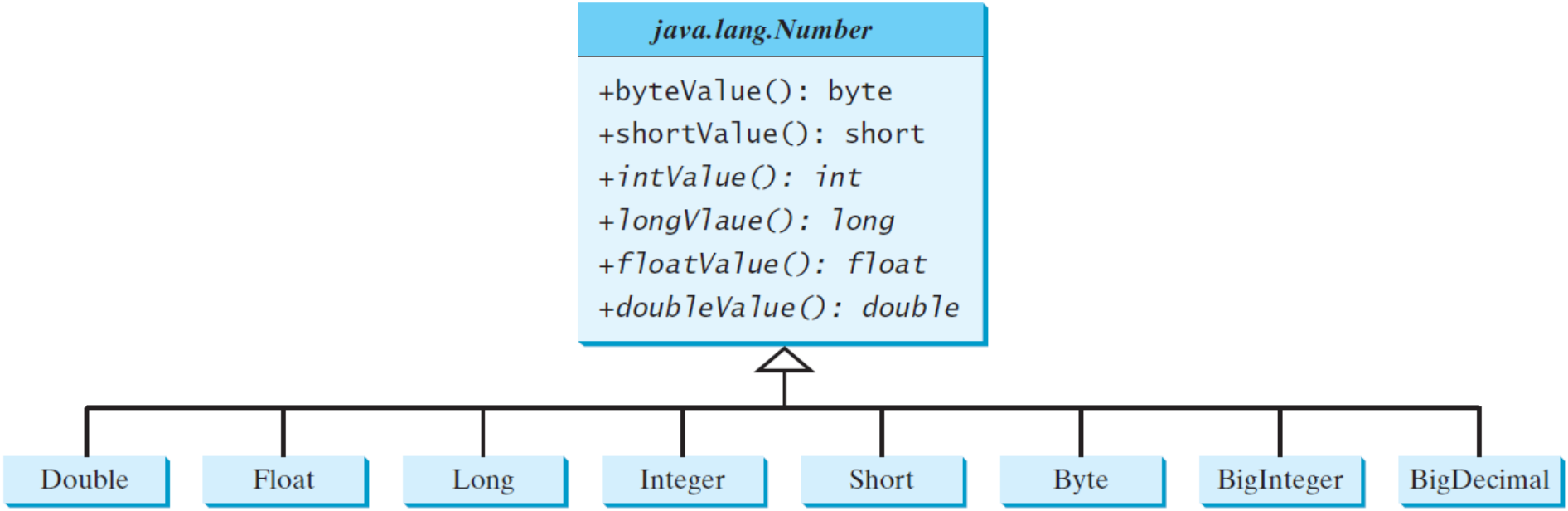
```
GeometricObject[] geo = new  GeometricObject[10];
```

- You can then create an instance of GeometricObject and assign its reference to the array like this:

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

Case Study: the Abstract Number Class

* The Number class is an abstract superclass for numeric wrapper classes (Double, Float, Long, Integer, Short, Byte), BigInteger and BigDecimal.



Listing 13.5 LargestNumbers.java

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

public class LargestNumbers {
  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("34323232344344343101"));
    // 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;
  }
}
```

```
The largest number is 343232323434434310 1
```

Explanation: the Abstract Number Class

* These classes have common methods:

  byteValue(), shortValue(), intValue(), longValue(),

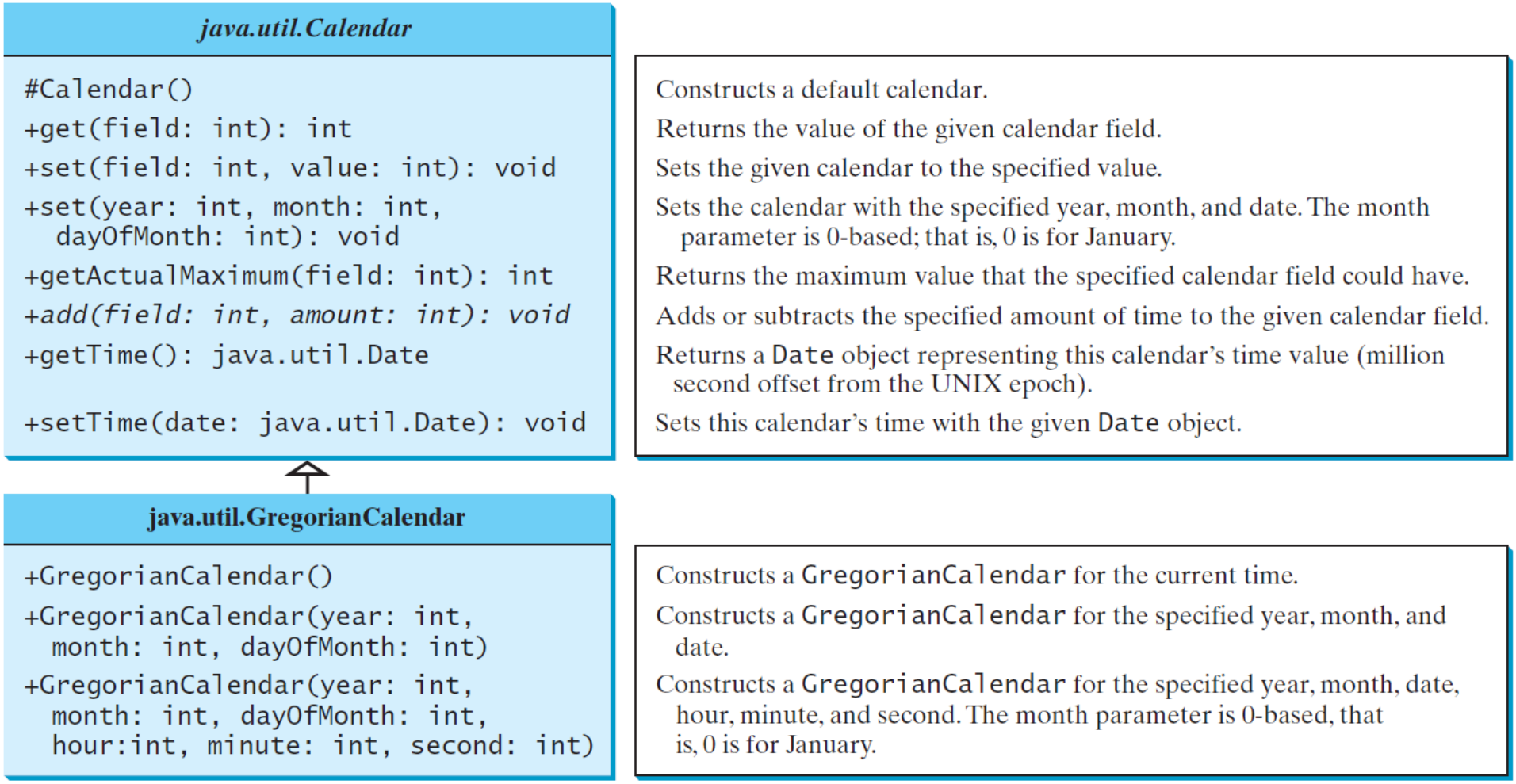
  floatValue(), and doubleValue()

* These methods are actually defined in the Number class.

* Since these methods cannot be implemented in the Number class, they are defined as abstract methods in the Number class.

* The Number class is therefore an abstract class.

The Abstract Calendar Class and Its GregorianCalendar subclass



The Abstract Calendar Class and Its GregorianCalendar subclass

* An instance of *java.util.Date* represents a specific instant in time with millisecond precision.

* *java.util.Calendar* is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a Date object.

* Subclasses of Calendar can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar.

* Currently, *java.util.GregorianCalendar* for the Gregorian calendar is supported in the Java API.

The GregorianCalendar Class

* You can use new GregorianCalendar() to construct a default GregorianCalendar with the current time

* and use new GregorianCalendar(year, month, date) to construct a GregorianCalendar with the specified year, month, and date.

Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?

What is an interface? Why is an interface useful?

* An interface is a classlike construct that contains only constants and abstract methods.

* In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects.

  - For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.

Define an Interface

* To distinguish an interface from a class, Java uses the following syntax to define an interface:

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

Example:

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

Interface is a Special Class

* An interface is treated like a special class.

- Each interface is compiled into a separate bytecode file, just like a regular class.

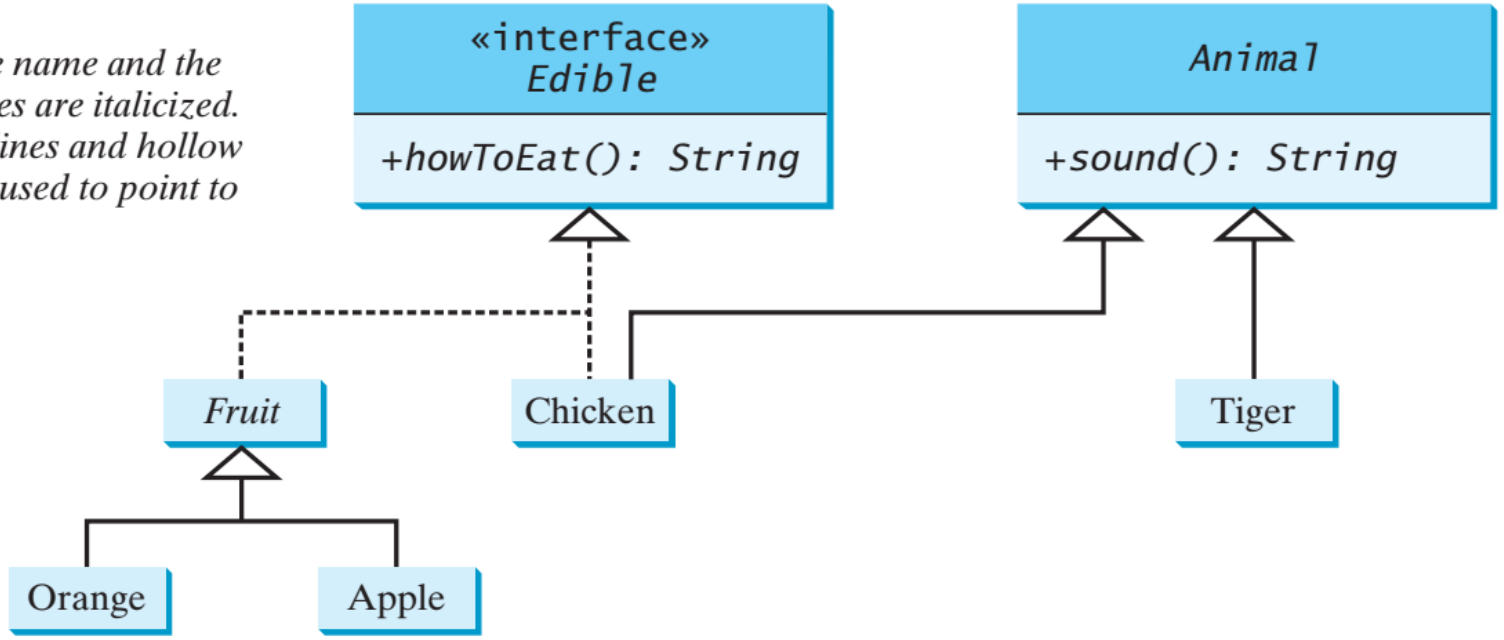
- in most cases you can use an interface more or less the same way you use an abstract class.

• Like an abstract class, you cannot create an instance from an interface using the new operator.

• you can use an interface as a data type for a variable, as the result of casting, and so on.

Example

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




```
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());
      }
    }
  }
}
```

```
Tiger: RROOAARR
Chicken: Fry it
Chicken: cock-a-doodle-doo
Apple: Make apple cider
```

* You can use the Edible interface to specify whether an object is edible.

* This is accomplished by letting the class for the object implement this interface using the *implements* keyword.

* For example, the classes Chicken and Fruit implement the Edible interface.

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

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

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

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

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

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

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

* All data fields are *public final static*
* all methods are *public abstract* in an interface.
* So, these modifiers can be omitted, as shown below:

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

  public abstract void p();
}
```

Equivalent

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

  void p();
}
```

* A constant defined in an interface can be accessed using syntax *InterfaceName.CONSTANT_NAME*.

The Comparable Interface

* Suppose you want to design a generic method to find the larger of two objects of the same type,

  - such as two students, two dates, two circles, or two squares.

  - In order to accomplish this, the two objects must be comparable, so the common behavior for the objects must be comparable.

* Java provides the Comparable interface for this purpose.

  - The Comparable interface defines the *compareTo* method for comparing objects.

  - The Comparable interface is a generic interface.

  - The generic type E is replaced by a concrete type when implementing this interface.

Example: The Comparable Interface

```
// This interface is defined in java.lang package
package java.lang
public interface Comparable<E> {
  public int compareTo(E o);
}
```

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

  - Since all the numeric wrapper classes and the Character class implement the Comparable interface, the compareTo method is implemented in these classes.

Integer and 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 and 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
  }
}
```

Generic `sort` Method

* the java.util.Arrays.sort(Object[]) method uses compareTo method to compare and sorts the objects in an array.

Listing 13.8 SortComparableObjects.java

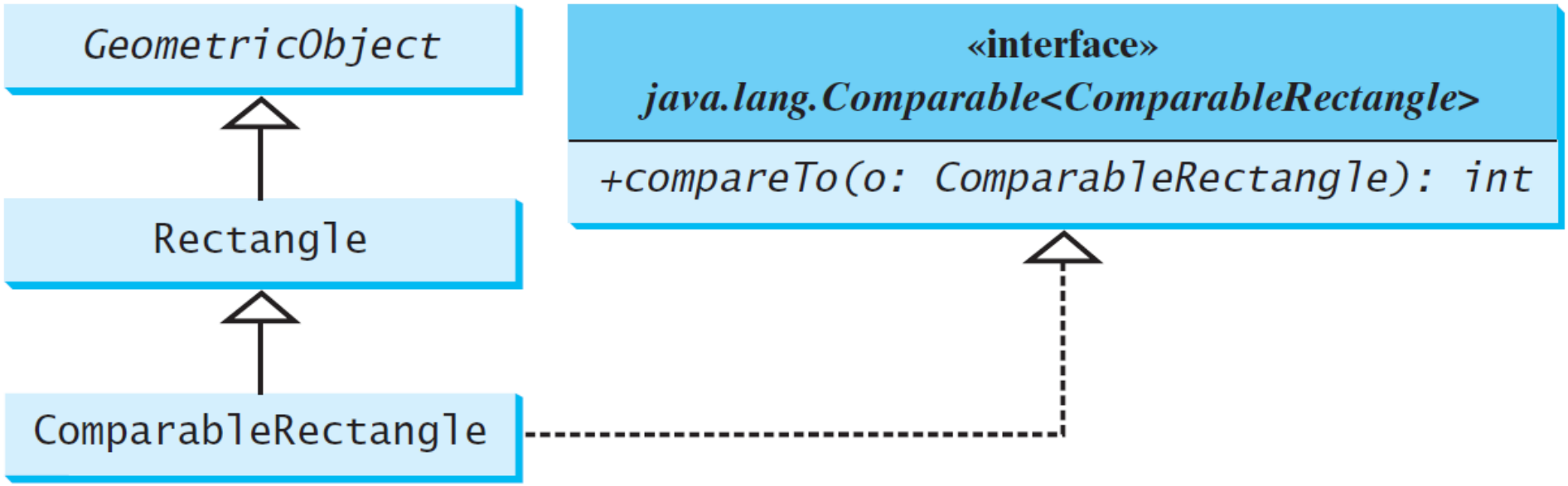
```
import java.math.*;

public class SortComparableObjects {
  public static void main(String[] args) {
    String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
    java.util.Arrays.sort(cities);
    for (String city: cities)
      System.out.print(city + " ");
    System.out.println();

    BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
      new BigInteger("432232323239292"),
      new BigInteger("54623239292")};
    java.util.Arrays.sort(hugeNumbers);
    for (BigInteger number: hugeNumbers)
      System.out.print(number + " ");
  }
}
```

```
Atlanta Boston Savannah Tampa
54623239292 432232323239292 2323231092923992
```

Defining Classes to Implement Comparable



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

  @Override // Implement the compareTo method defined in Comparable
  public int compareTo(ComparableRectangle o) {
    if (getArea() > o.getArea())
      return 1;
    else if (getArea() < o.getArea())
      return -1;
    else
      return 0;
  }

  @Override // Implement the toString method in GeometricObject
  public String toString() {
    return super.toString() + " Area: " + getArea();
  }
}
```

```
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);
    for (Rectangle rectangle: rectangles) {
      System.out.print(rectangle + " ");
      System.out.println();
    }
  }
}
```

```
Width: 3.4 Height: 5.4 Area: 18.36
Width: 1.4 Height: 25.4 Area: 35.559999999999995
Width: 7.4 Height: 35.4 Area: 261.96
Width: 13.24 Height: 55.4 Area: 733.496
```

The `Cloneable` Interfaces

* the Cloneable interface is a special case.

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

* Marker Interface: An empty interface.
  - does not contain constants or methods.
  - used to denote that a class possesses certain properties.

* A class that implements the Cloneable interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

Examples

* Many classes (e.g., Date and Calendar) in the Java library implement Cloneable.

* 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

* To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class.

```
public class House implements Cloneable, Comparable<House> {
  private int id;
  private double area;
  private java.util.Date whenBuilt;

  public House(int id, double area) {
    this.id = id;
    this.area = area;
    whenBuilt = new java.util.Date();
  }
  @Override /** Override the protected clone method defined in
    the Object class, and strengthen its accessibility */
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
}
```

Listing 13.11 House.java

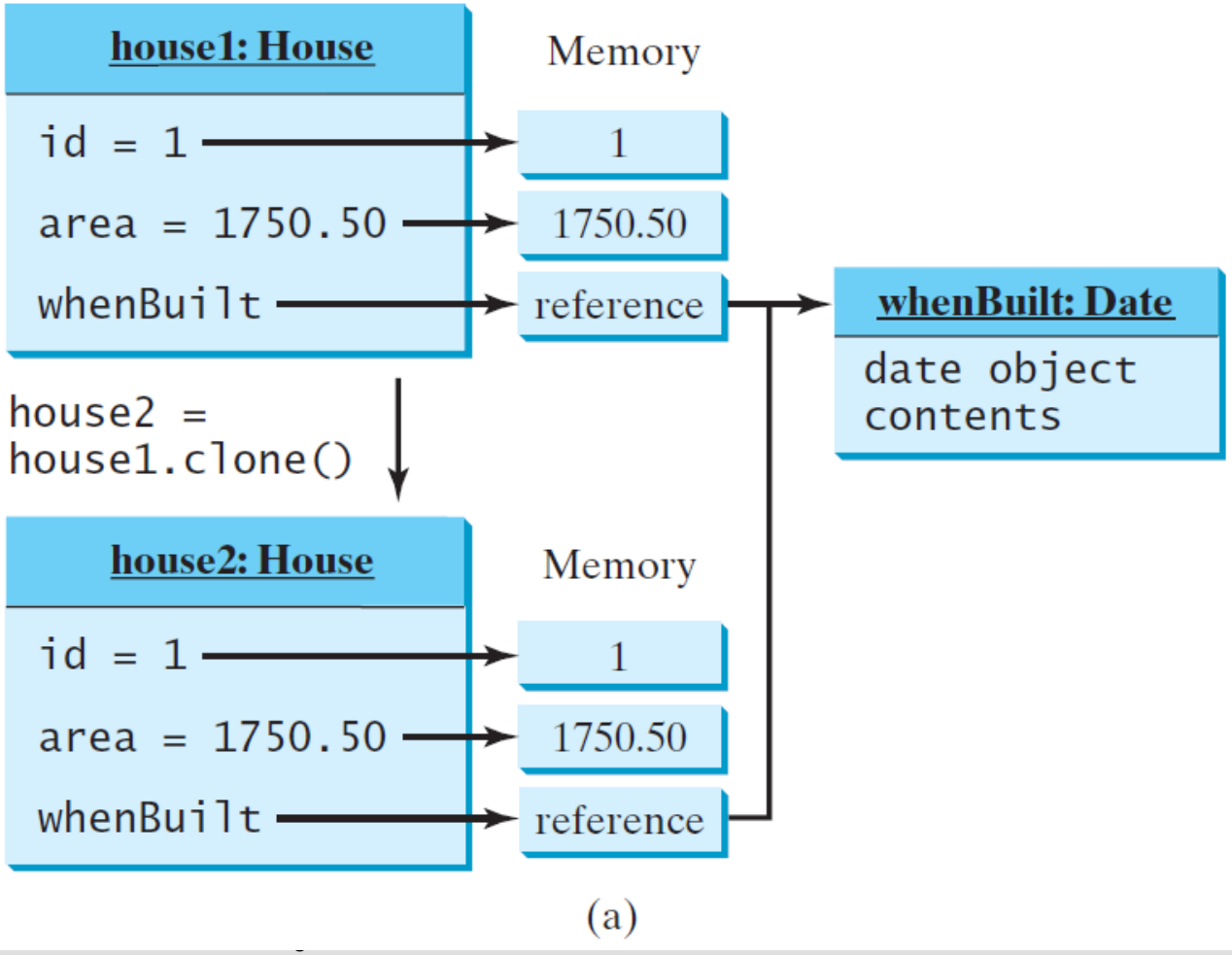
* The clone method in the Object class copies each field from the original object to the target object.

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

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

Shallow Copy

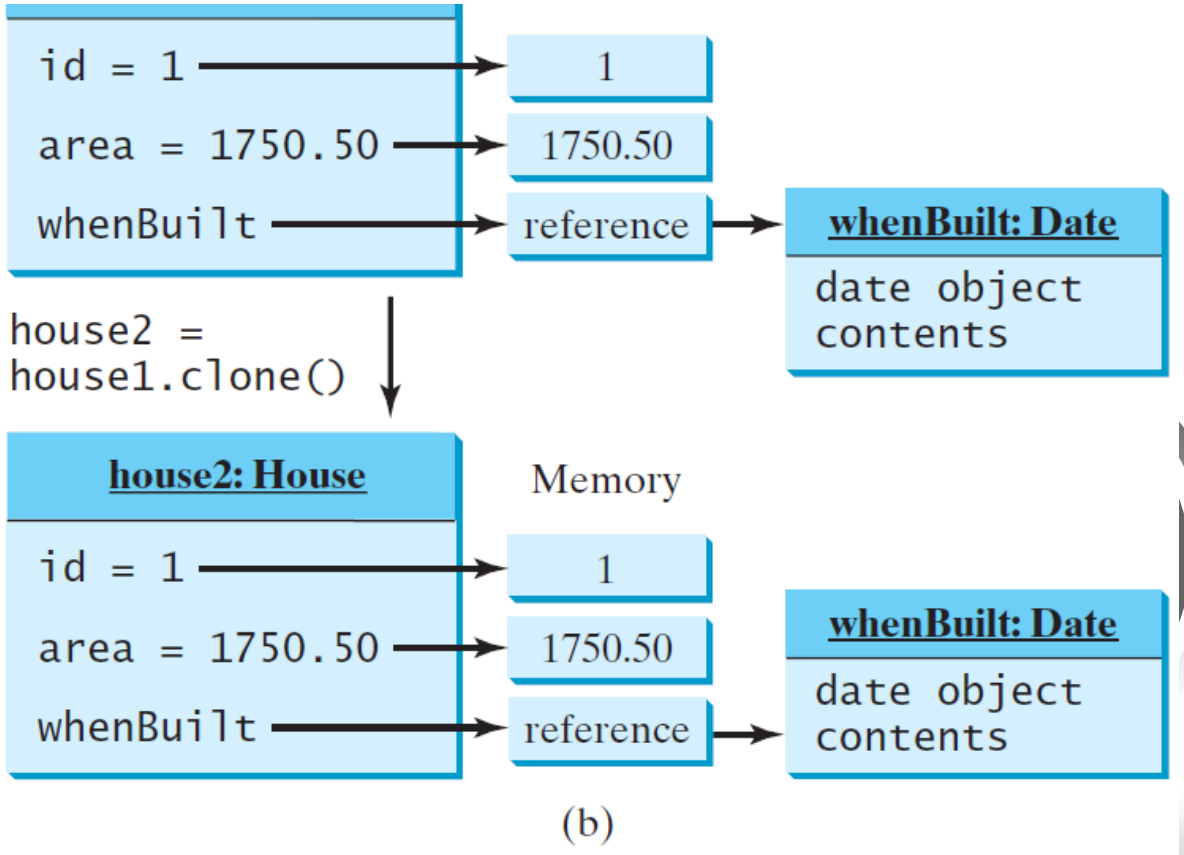


(a) The default clone method performs a shallow copy.

Shallow vs. Deep Copy

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

Deep Copy



(b) The custom clone method performs a deep copy.

Interfaces vs. Abstract Classes

* An interface can be used more or less the same way as an abstract class, but defining an interface is different from defining an abstract class.

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

- 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.

* Java allows only single inheritance for class extension but allows multiple extensions for interfaces.

* For example,

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

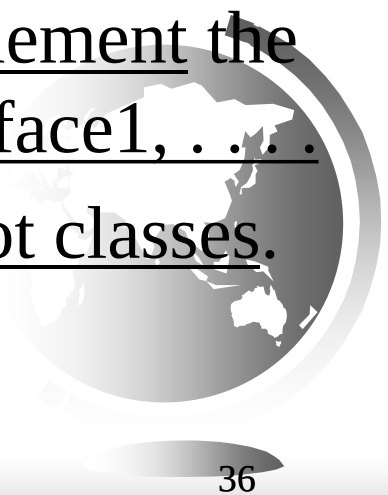
Interfaces vs. Abstract Classes, cont.

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

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

* A class implementing NewInterface must implement the abstract methods defined in NewInterface, Interface1, ...

* An interface can extend other interfaces but not classes.



Interfaces vs. Abstract Classes, cont.

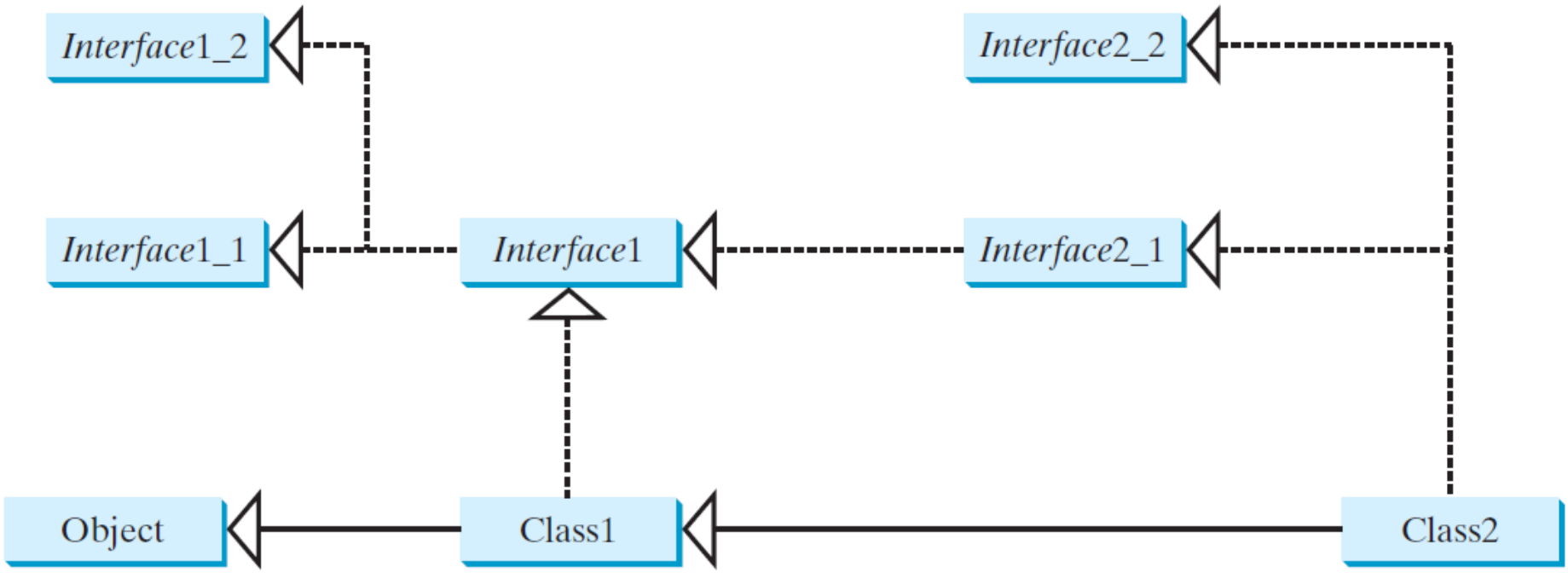
* All classes share a single root, the *Object* class, but there is no single root for interfaces.

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

* If a class extends an interface, this interface plays the same role as a superclass.

* You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.

Interfaces vs. Abstract Classes, cont.



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

Caution: conflict interfaces

* In rare occasions, a class may implement two interfaces with conflict information

  - e.g., two same constants with different values

  - or two methods with same signature but different return type).

* This type of errors will be detected by the compiler.

Whether to use an interface or a class?

* Abstract classes and interfaces can both be used to model common features. Which to use?

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

- 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.

- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. You have to design one as a superclass, and others as interface.

The `Rational` Class

* rational number:

- has the form a/b, where a is the numerator and b the denominator. b can not be 0.

- are used in exact computations.

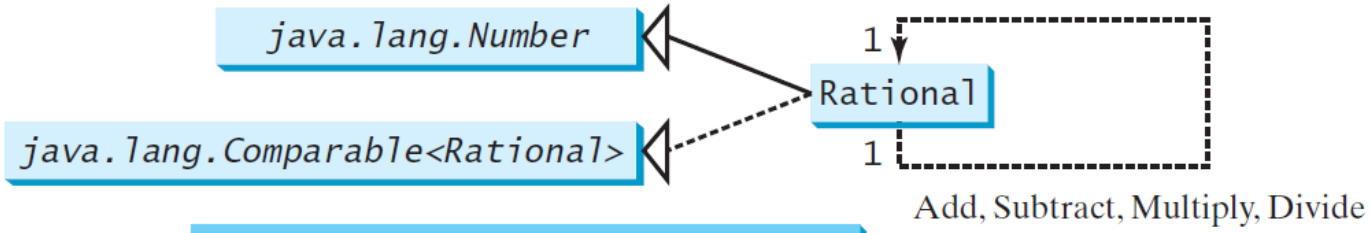
- Java does not provide for rational numbers.

- Since it shares many common features with numbers, and Number is the root class for numeric wrapper classes, it is appropriate to define Rational as a subclass of Number.

- Since comparable, the Rational class should also implement the Comparable interface.

- many equivalent numbers, 1/3 = 2/6 = 3/9 = 4/12, where 1/3 is in lowest terms, GCD is used to achieve it.

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

The numerator of this rational number.
The denominator of this rational number.
Creates a rational number with numerator 0 and denominator 1.
Creates a rational number with a specified numerator and denominator.
Returns the numerator of this rational number.
Returns the denominator of this rational number.
Returns the addition of this rational number with another.
Returns the subtraction of this rational number with another.
Returns the multiplication of this rational number with another.
Returns the division of this rational number with another.
Returns a string in the form "numerator/denominator." Returns the numerator if denominator is 1.
Returns the greatest common divisor of n and d.

Listing 13.12 TestRationalClass.java

```
public class TestRationalClass {
  /** Main method */
  public static void main(String[] args) {
    // Create and initialize two rational numbers r1 and r2
    Rational r1 = new Rational(4, 2);
    Rational r2 = new Rational(2, 3);

    // Display results

    System.out.println(r1 + " + " + r2 + " = " + r1.add(r2));
    System.out.println(r1 + " - " + r2 + " = " + r1.subtract(r2));
    System.out.println(r1 + " * " + r2 + " = " + r1.multiply(r2));
    System.out.println(r1 + " / " + r2 + " = " + r1.divide(r2));
    System.out.println(r2 + " is " + r2.doubleValue());
  }
}
```

```
2 + 2/3 = 8/3
2 - 2/3 = 4/3
2 * 2/3 = 4/3
2 / 2/3 = 3
2/3 is 0.6666666666666666
```

Listing 13.13 Rational.java

```
public class Rational extends Number implements Comparable<Rational> {
  // Data fields for numerator and denominator
  private long numerator = 0;
  private long denominator = 1;

  /** Construct a rational with default properties */
  public Rational() {
    this(0, 1);
  }

  /** Construct a rational with specified numerator and denominator */
  public Rational(long numerator, long denominator) {
    long gcd = gcd(numerator, denominator);
    this.numerator = ((denominator > 0) ? 1 : -1) * numerator / gcd;
    this.denominator = Math.abs(denominator) / gcd;
  }

  /** Find GCD of two numbers */
  private static long gcd(long n, long d) {
    long n1 = Math.abs(n);
    long n2 = Math.abs(d);
    int gcd = 1;

    for (int k = 1; k <= n1 && k <= n2; k++) {
      if (n1 % k == 0 && n2 % k == 0)
        gcd = k;
    }

    return gcd;
  }
```

Listing 13.13 Rational.java

```
/** Return numerator */
public long getNumerator() {
  return numerator;
}

/** Return denominator */
public long getDenominator() {
  return denominator;
}

/** Add a rational number to this rational */
public Rational add(Rational secondRational) {
  long n = numerator * secondRational.getDenominator() +
    denominator * secondRational.getNumerator();
  long d = denominator * secondRational.getDenominator();
  return new Rational(n, d);
}

/** Subtract a rational number from this rational */
public Rational subtract(Rational secondRational) {
  long n = numerator * secondRational.getDenominator()
    - denominator * secondRational.getNumerator();
  long d = denominator * secondRational.getDenominator();
  return new Rational(n, d);
}

/** Multiply a rational number by this rational */
public Rational multiply(Rational secondRational) {
  long n = numerator * secondRational.getNumerator();
  long d = denominator * secondRational.getDenominator();
  return new Rational(n, d);
}
```

```
/** Divide a rational number by this rational */
public Rational divide(Rational secondRational) {
  long n = numerator * secondRational.getDenominator();
  long d = denominator * secondRational.numerator;
  return new Rational(n, d);
}
```

Listing 13.13 Rational.java

```
@Override
public String toString() {
  if (denominator == 1)
    return numerator + "";
  else
    return numerator + "/" + denominator;
}

@Override // Override the equals method in the Object class
public boolean equals(Object other) {
  if ((this.subtract((Rational)(other))).getNumerator() == 0)
    return true;
  else
    return false;
}

@Override // Implement the abstract intValue method in Number
public int intValue() {
  return (int)doubleValue();
}
```

Listing 13.13 Rational.java

```
  @Override // Implement the abstract floatValue method in Number
  public float floatValue() {
    return (float)doubleValue();
  }

  @Override // Implement the doubleValue method in Number
  public double doubleValue() {
    return numerator * 1.0 / denominator;
  }

  @Override // Implement the abstract longValue method in Number
  public long longValue() {
    return (long)doubleValue();
  }

  @Override // Implement the compareTo method in Comparable
  public int compareTo(Rational o) {
    if (this.subtract(o).getNumerator() > 0)
      return 1;
    else if (this.subtract(o).getNumerator() < 0)
      return -1;
    else
      return 0;
  }
}
```

Class Design Guidelines

(Coherence: coherent purpose)

* A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.

- You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.

Class Design Guidelines, cont.

(Coherence: Separating responsibilities)

* A single entity with too many responsibilities can be broken into several classes to separate responsibilities.

* The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities.

- The String class deals with immutable strings,

- the StringBuilder class is for creating mutable strings,

- and the StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.

Class Design Guidelines, cont.

(Consistency: naming conventions)

* Follow standard Java programming style and naming conventions.

* Choose informative names for classes, data fields, and methods.

* Always place the data declaration before the constructor, and place constructors before methods.

Class Design Guidelines, cont.

(Consistency: naming consistency)

* Make the names consistent.

* It is not a good practice to choose different names for similar operations.

  - For example, the length() method returns the size of a String, a StringBuilder, and a StringBuffer.

  - It would be inconsistent if different names were used for this method in these classes.

Class Design Guidelines, cont.

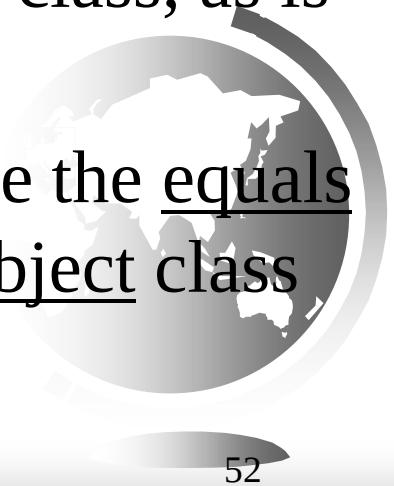
(Consistency: no-arg constructor)

* In general, you should consistently provide a public no-arg constructor for constructing a default instance.

* If a class does not support a no-arg constructor, document the reason. If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.

* If you want to prevent users from creating an object for a class, you can declare a private constructor in the class, as is the case for the Math class.

* Provide a public no-arg constructor and override the equals method and the toString method defined in the Object class whenever possible.



Class Design Guidelines, cont.

(Encapsulation: encapsulate data fields)

* A class should use the private modifier to hide its data from direct access by clients. This makes the class easy to maintain.

* Provide a getter method only if you want the data field to be readable, and provide a setter method only if you want the data field to be updateable.

* For example, the Rational class provides a getter method for numerator and denominator, but no setter method, because a Rational object is immutable.

Class Design Guidelines, cont.

(Clarity: easy to explain, independent methods)

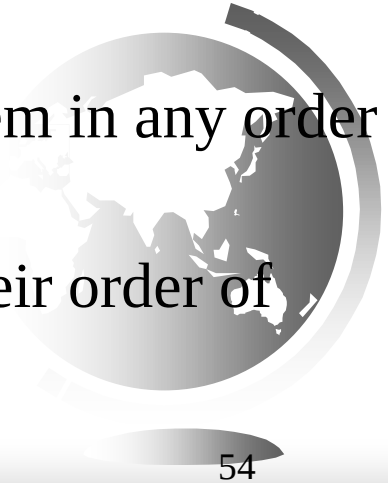
* In addtion to Cohesion, consistency, and encapsulation, a class should have a clear contract that is easy to explain and easy to understand.

* Users can incorporate classes in different combinations, orders, and environments. Therefore, you should,

- design a class that imposes no restrictions on how or when the user can use it,

- design the properties in a way that lets the user set them in any order and with any combination of values,

- and design methods that function independently of their order of occurrence.



Class Design Guidelines, cont.

(Clarity: intuitive meaning, independent properties)

* Methods should be defined intuitively without causing confusion.

* You should not declare a data field that can be derived from other data fields.

  - For example, the following Person class has two data fields: birthDate and age.

  - Since age can be derived from birthDate, age should not be declared as a data field.

Class Design Guidelines, cont.

(Completeness)

* Classes are designed for use by many different customers.

* In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through properties and methods.

* For example, the String class contains more than 40 methods that are useful for a variety of applications.

Class Design Guidelines, cont.

(Instance vs. Static)

*A variable or method that is dependent on a specific instance of the class must be an instance variable or method.

* A variable that is shared by all the instances of a class should be declared static. A method that is not dependent on a specific instance should be defined as a static method.

* Always reference static variables and methods from a class name to improve readability and avoid errors.

* It is better to use a setter method to change the static data field in stead of from a constructor.

* A constructor is always instance, because it is used to create a specific instance.

* A static variable or method can be invoked from an instance method, but an instance variable or method cannot be invoked from a static method.

Class Design Guidelines, cont.

(Inheritance vs. Aggregation)

* The difference between inheritance and aggregation is the difference between an is-a and a has-a relationship.

* For example, an apple is a fruit; thus, you would use inheritance to model the relationship between the classes Apple and Fruit.

* A person has a name; thus, you would use aggregation to model the relationship between the classes Person and Name.

Class Design Guidelines, cont.

(Interfaces vs. Abstract Classes)

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

* weak is-a relationship, indicating that an object possesses a certain property, can be modeled using interfaces.

* Interfaces are more flexible than abstract classes, because a subclass can extend only one superclass but can implement any number of interfaces. However, interfaces cannot contain concrete methods.

* The virtues of interfaces and abstract classes can be combined so that you can use the interface or the abstract class, whichever is convenient.

Class Design Guidelines, cont.

(Using Visibility Modifiers)

* Each class can present two contracts – one for the users of the class and one for the extenders of the class.

* Make the fields private and accessor methods public if they are intended for the users of the class.

* Make the fields or method protected if they are intended for extenders of the class.

* A class should use the private modifier to hide its data from direct access by clients.

  - You can use get methods and set methods to provide users with access to the private data.

  - A class should also hide methods not intended for client use.