Chapter 1 Introduction to Computers, Programs, and Java

Chapter 14 Abstract Classes and Interfaces

Motivations

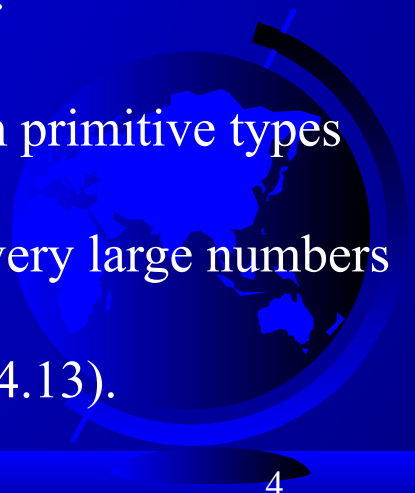
You learned how to write simple programs to display GUI components. Can you write the code to respond to user actions such as clicking a button?

HandleEvent

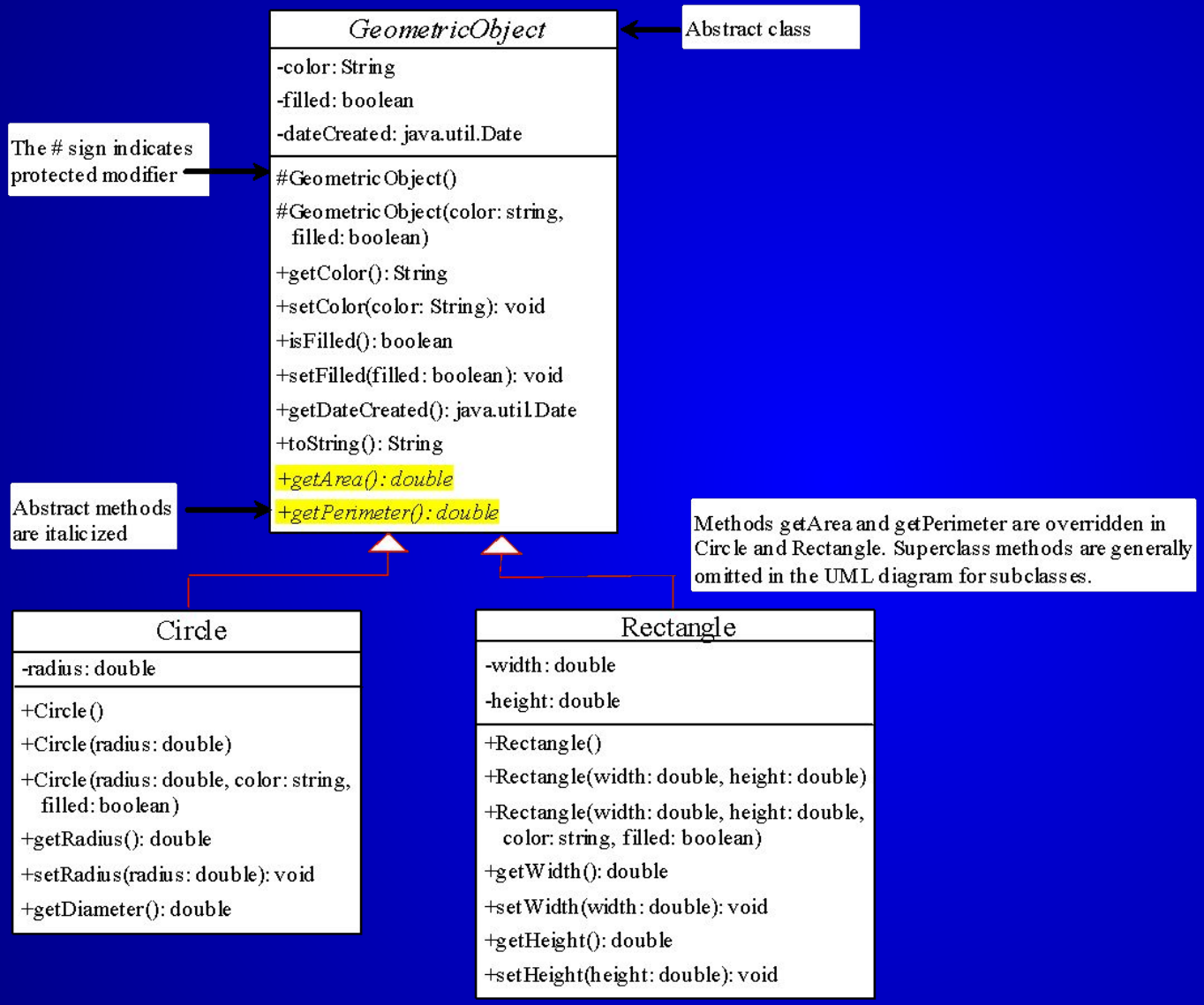
Run

Objectives

- To design and use abstract classes (§14.2).
- To process a calendar using the Calendar and GregorianCalendar classes (§14.3).
- To specify common behavior for objects using interfaces (§14.4).
- To define interfaces and define classes that implement interfaces (§14.4).
- To define a natural order using the Comparable interface (§14.5).
- To enable objects to listen for action events using the ActionListener interface (§14.6).
- To make objects cloneable using the Cloneable interface (§14.7).
- To explore the similarities and differences between an abstract class and an interface (§14.8).
- To create objects for primitive values using the wrapper classes (Byte, Short, Integer, Long, Float, Double, Character, and Boolean) (§14.9).
- To create a generic sort method (§14.10).
- To simplify programming using automatic conversion between primitive types and wrapper class types (§14.11).
- To use the BigInteger and BigDecimal classes for computing very large numbers with arbitrary precisions (§14.12).
- To design the Rational class for defining the Rational type (§14.13).



Abstract Classes and Abstract Methods



GeometricObject

Circle

Rectangle

TestGometricObject

Run

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. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.

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

abstract class without abstract method

A class that contains abstract methods must be abstract. However, 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. This class is used as a base class for defining a new subclass.

superclass of abstract class may be concrete

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.

concrete method overridden to be abstract

A subclass can override a method from its superclass to define it abstract. This is rare, but useful when the implementation of the method 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 a data type. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

GeometricObject[] geo = new GeometricObject[10];

The Abstract Calendar Class and Its GregorianCalendar subclass

java.util.Calendar	
#Calendar()	Constructs a default calendar.
+get(field: int): int	Returns the value of the given calendar field.
+set(field: int, value: int): void	Sets the given calendar to the specified value.
+set(year: int, month: int, dayOfMonth: int): void	Sets the calendar with the specified year, month, and date. The month parameter is 0-based, that is, 0 is for January.
+getActualMaximum(field: int): int	Returns the maximum value that the specified calendar field could have.
+add(field: int, amount: int): void	Adds or subtracts the specified amount of time to the given calendar field.
+getTime(): java.util.Date	Returns a Date object representing this calendar's time value (million second offset from the Unix epoch).
+setTime(date: java.util.Date): void	Sets this calendar's time with the given Date object.

java.util.GregorianCalendar	
+GregorianCalendar()	Constructs a GregorianCalendar for the current time.
+GregorianCalendar(year: int, month: int, dayOfMonth: int)	Constructs a GregorianCalendar for the specified year, month, and day of month.
+GregorianCalendar(year: int, month: int, dayOfMonth: int, hour:int, minute: int, second: int)	Constructs a GregorianCalendar for the specified year, month, day of month, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

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

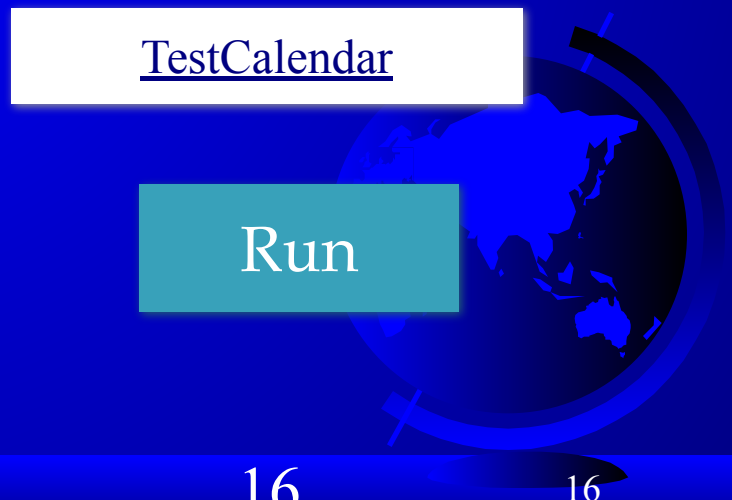
You can use new GregorianCalendar() to construct a default GregorianCalendar with the current time and use new GregorianCalendar(year, month, date) to construct a GregorianCalendar with the specified year, month, and date. The month parameter is 0-based, i.e., 0 is for January.

The get Method in Calendar Class

The get(int field) method defined in the Calendar class is useful to extract the date and time information from a Calendar object. The fields are defined as constants, as shown in the following.

Constant	Description
YEAR	The year of the calendar.
MONTH	The month of the calendar with 0 for January.
DATE	The day of the calendar.
HOUR	The hour of the calendar (12-hour notation).
HOUR_OF_DAY	The hour of the calendar (24-hour notation).
MINUTE	The minute of the calendar.
SECOND	The second of the calendar.
DAY_OF_WEEK	The day number within the week with 1 for Sunday.
DAY_OF_MONTH	Same as DATE.
DAY_OF_YEAR	The day number in the year with 1 for the first day of the year.
WEEK_OF_MONTH	The week number within the month.
WEEK_OF_YEAR	The week number within the year.
AM_PM	Indicator for AM or PM (0 for AM and 1 for PM).

Getting Date/Time Information from Calendar



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 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;
  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 in Java. Each interface is compiled into a separate bytecode file, just like a regular class. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a variable, as the result of casting, and so on.

Example

You can now 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 (See TestEdible).

Edible

TestEdible

Run

Omitting Modifiers in Interfaces

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

```
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 syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

Example: The Comparable Interface

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable {
   public int compareTo(Object o);
}
```

String and Date Classes

Many classes (e.g., String and Date) in the Java library implement Comparable to define a natural order for the objects. If you examine the source code of these classes, you will see the keyword implements used in the classes, as shown below:

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

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

```
new String() instanceof String
new String() instanceof Comparable
new java.util.Date() instanceof java.util.Date
new java.util.Date() instanceof Comparable
```

Generic `max` Method

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Comparable max
      (Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(a)

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Object max
      (Object o1, Object o2) {
    if ((Comparable)o1).compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(b)

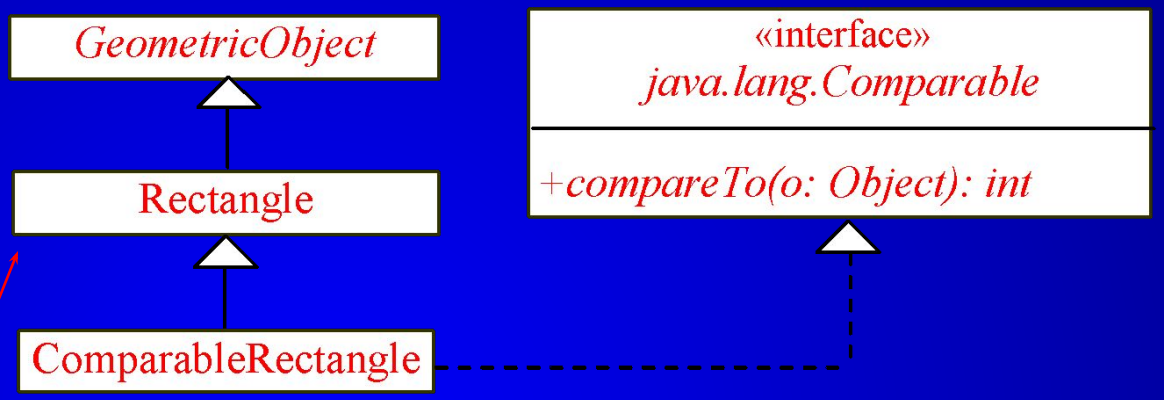
```
String s1 = "abcdef";
String s2 = "abcdee";
String s3 = (String)Max.max(s1, s2);
```

```
Date d1 = new Date();
Date d2 = new Date();
Date d3 = (Date)Max.max(d1, d2);
```

The return value from the max method is of the Comparable type. So, you need to cast it to String or Date explicitly.

Defining Classes to Implement Comparable

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

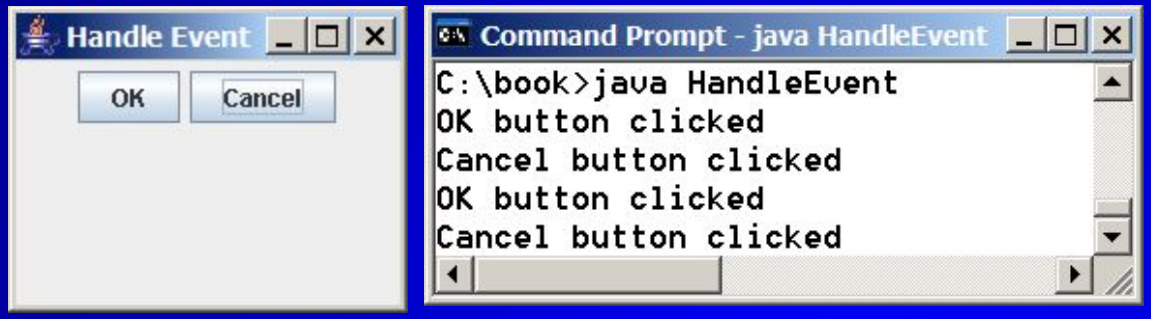


ComparableRectangle

You cannot use the max method to find the larger of two instances of Rectangle, because Rectangle does not implement Comparable. However, you can define a new rectangle class that implements Comparable. The instances of this new class are comparable. Let this new class be named ComparableRectangle.

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```

The `ActionListener` Interfaces



HandleEvent

Run

Handling GUI Events

Source object (e.g., button)

Listener object contains a method for processing the event.

Trace Execution

```
public class HandleEvent extends JFrame {
  public HandleEvent() {
    …
   OKListenerClass listener1 = new OKListenerClass();
    jbtOK.addActionListener(listener1);
    …
  }

  public static void main(String[] args) {
    …
  }
}

class OKListenerClass implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

1. Start from the main method to create a window and display it

animation

Trace Execution

```
public class HandleEvent extends JFrame {
  public HandleEvent() {
    …
   OKListenerClass listener1 = new OKListenerClass();
    jbtOK.addActionListener(listener1);
    …
  }

  public static void main(String[] args) {
    …
  }
}

class OKListenerClass implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

2. Click OK

Trace Execution

```
public class HandleEvent extends JFrame {
  public HandleEvent() {
    …
   OKListenerClass listener1 = new OKListenerClass();
    jbtOK.addActionListener(listener1);
    …
  }

  public static void main(String[] args) {
    …
  }
}

class OKListenerClass implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}
```

3. Click OK. The JVM invokes the listener's actionPerformed method

The `Cloneable` Interfaces

Marker Interface: An empty interface.

A marker interface does not contain constants or methods. It is used to denote that a class possesses certain desirable 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.

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

Examples

Many classes (e.g., Date and Calendar) in the Java library implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code

```
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

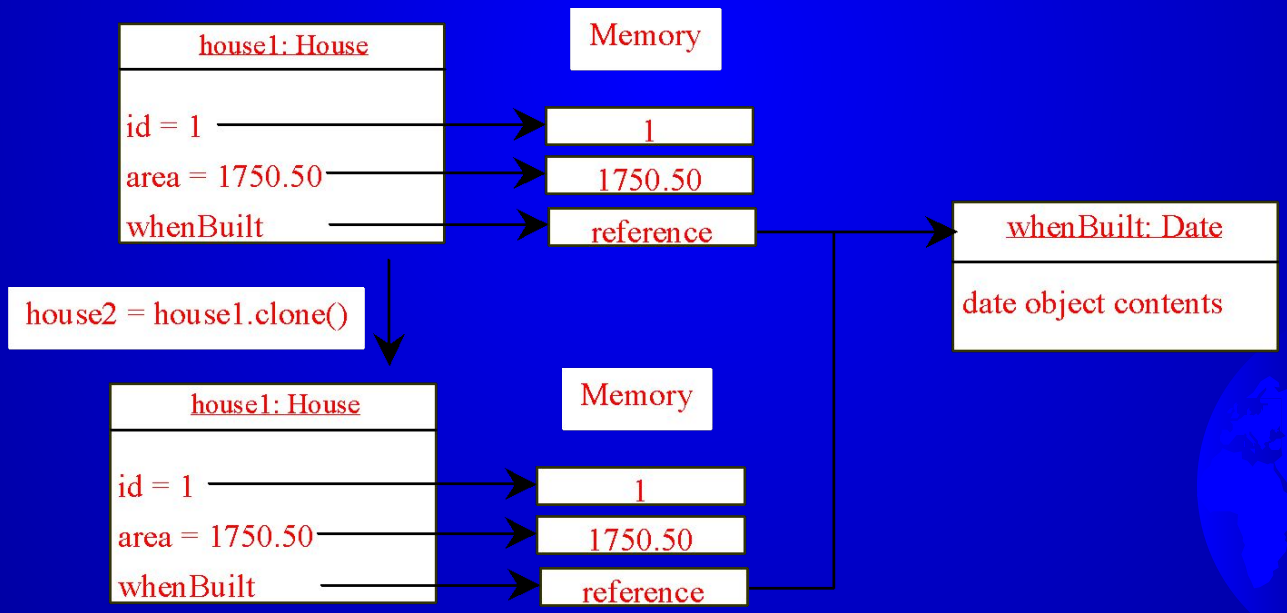
To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class. The following code defines a class named House that implements Cloneable and Comparable.

House

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

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



Interfaces vs. Abstract Classes

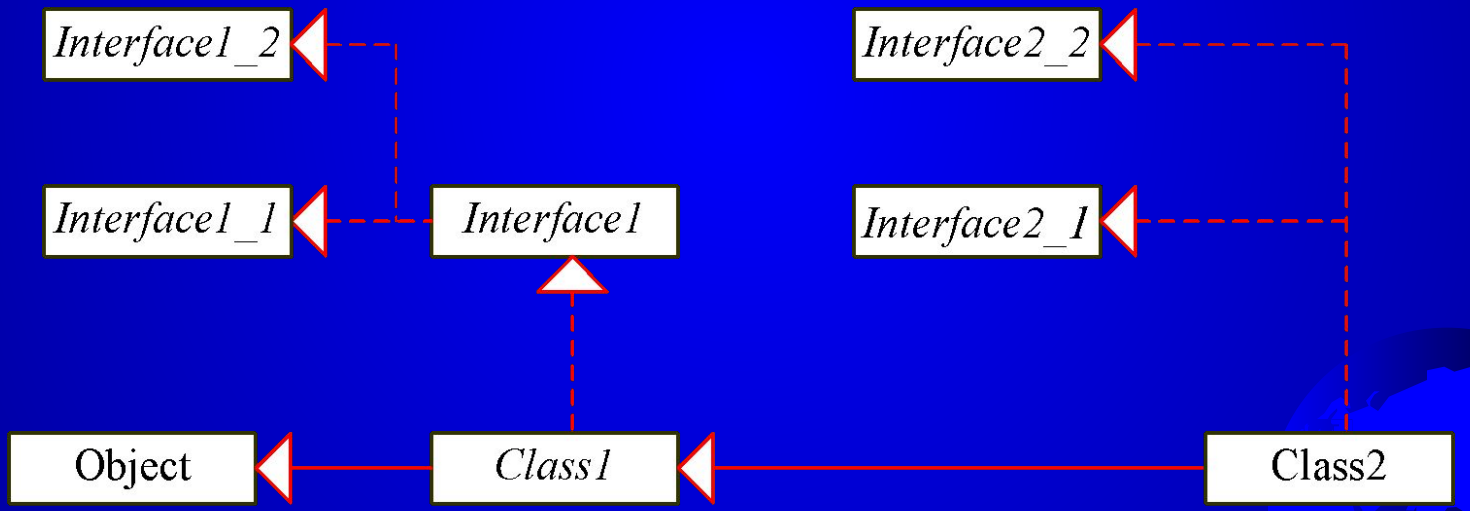
In an interface, the data must be constants; an abstract class can have all types of data.

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.

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 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.



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.

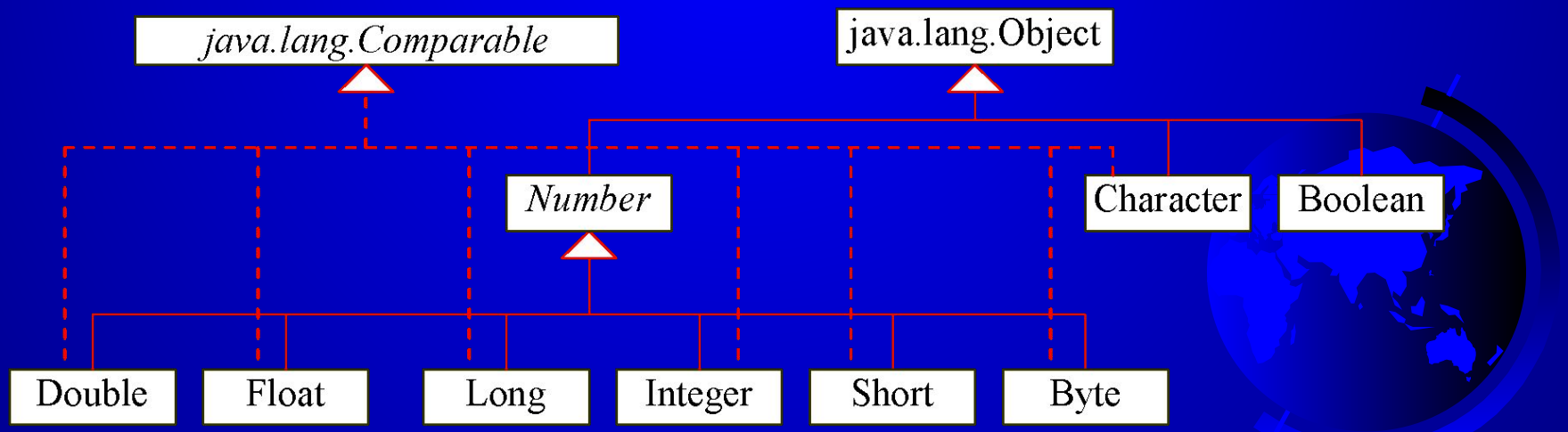
Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features. 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. So their relationship should be modeled using class inheritance. 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. See Chapter 10, "Object-Oriented Modeling," for more discussions.

Wrapper Classes

- Boolean
- Character
- Short
- Byte
- Integer
- Long
- Float
- Double

NOTE: (1) The wrapper classes do not have no-arg constructors. (2) The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.



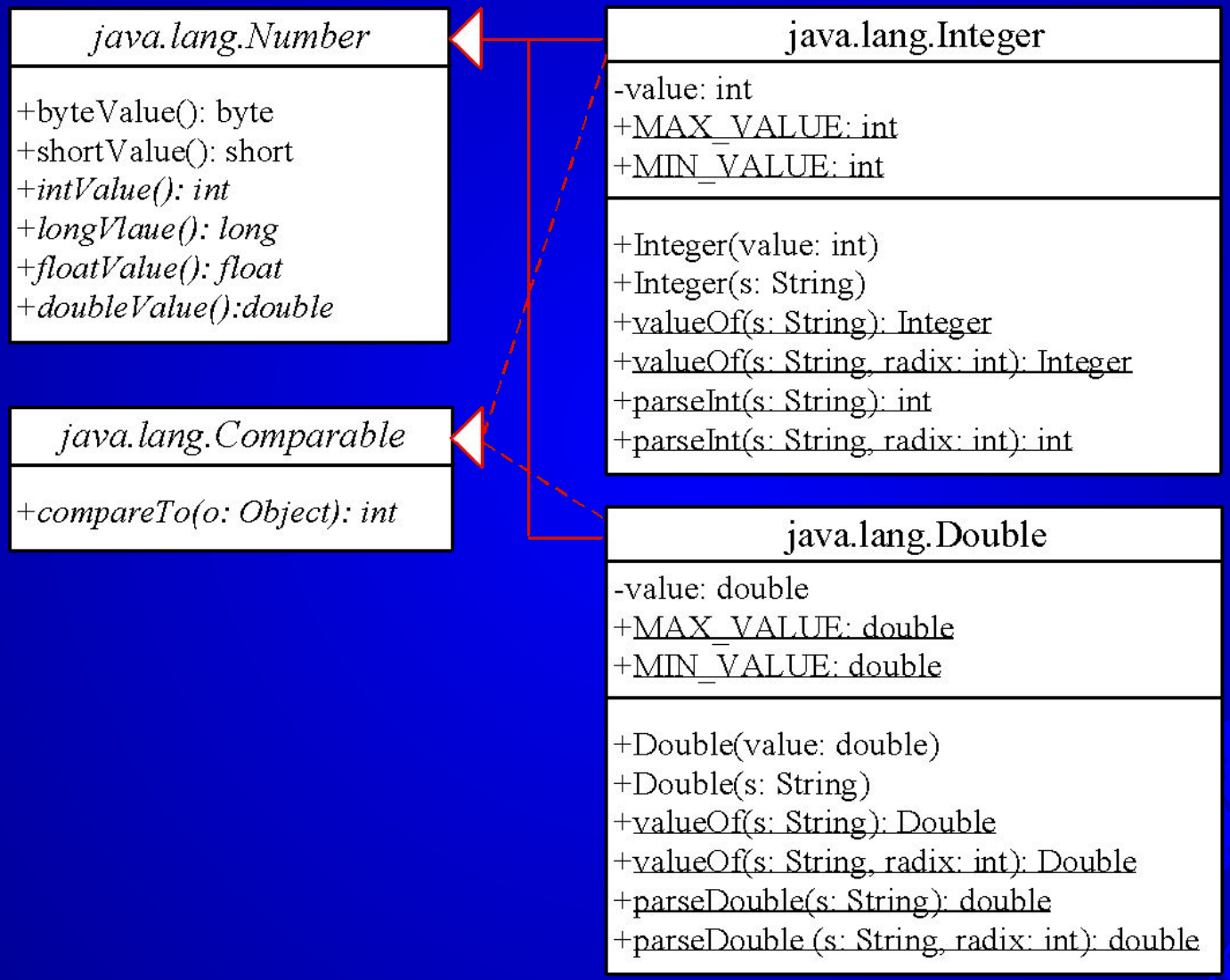
The toString, equals, and hashCode Methods

Each wrapper class overrides the toString, equals, and hashCode methods defined in the Object class. Since all the numeric wrapper classes and the Character class implement the Comparable interface, the compareTo method is implemented in these classes.

The Number Class

Each numeric wrapper class extends the abstract Number class, which contains the methods doubleValue, floatValue, intValue, longValue, shortValue, and byteValue. These methods "convert" objects into primitive type values. The methods doubleValue, floatValue, intValue, longValue are abstract. The methods byteValue and shortValue are not abstract, which simply return (byte)intValue() and (short)intValue(), respectively.

The `Integer` and `Double` Classes



The `Integer` Class and the `Double` Class

- Constructors
- Class Constants `MAX_VALUE`, `MIN_VALUE`
- Conversion Methods

Numeric Wrapper Class Constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value. The constructors for Integer and Double are:

public Integer(int value)

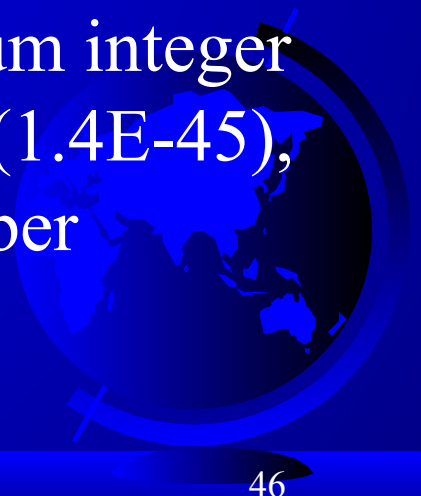
public Integer(String s)

public Double(double value)

public Double(String s)

Numeric Wrapper Class Constants

Each numerical wrapper class has the constants MAX_VALUE and MIN_VALUE. MAX_VALUE represents the maximum value of the corresponding primitive data type. For Byte, Short, Integer, and Long, MIN_VALUE represents the minimum byte, short, int, and long values. For Float and Double, MIN_VALUE represents the minimum *positive* float and double values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45), and the maximum double floating-point number (1.79769313486231570e+308d).



Conversion Methods

Each numeric wrapper class implements the abstract methods doubleValue, floatValue, intValue, longValue, and shortValue, which are defined in the Number class. These methods "convert" objects into primitive type values.

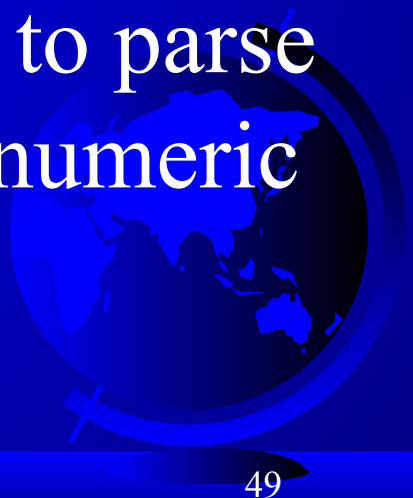
The Static valueOf Methods

The numeric wrapper classes have a useful class method, valueOf(String s). This method creates a new object initialized to the value represented by the specified string. For example:

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

The Methods for Parsing Strings into Numbers

You have used the parseInt method in the Integer class to parse a numeric string into an int value and the parseDouble method in the Double class to parse a numeric string into a double value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.



Sorting an Array of Objects

Objective: The example presents a generic method for sorting an array of objects. The objects are instances of the Comparable interface and they are compared using the compareTo method.

GenericSort

Run

TIP

Java provides a static sort method for sorting an array of Object in the java.util.Arrays class. So you can use the following code to sort arrays in this example:

```
java.util.Arrays.sort(intArray);
java.util.Arrays.sort(doubleArray);
java.util.Arrays.sort(charArray);
java.util.Arrays.sort(stringArray);
```

NOTE

Arrays are objects. An array is an instance of the Object class. Furthermore, if A is a subclass of B, every instance of A[] is an instance of B[]. Therefore, the following statements are all true:

new int[10] instanceof Object
new GregorianCalendar[10] instanceof Calendar[];
new Calendar[10] instanceof Object[]
new Calendar[10] instanceof Object

CAUTION

Although an int value can be assigned to a double type variable, int[] and double[] are two incompatible types. Therefore, you cannot assign an int[] array to a variable of double[] or Object[] type.

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):

```
Integer[] intArray = {new Integer(2),
  new Integer(4), new Integer(3)};
```

(a)

Equivalent

```
Integer[] intArray = {2, 4, 3};
```

New JDK 1.5 boxing

(b)

```
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Unboxing

BigInteger and BigDecimal

If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package. Both are *immutable*. Both extend the Number class and implement the Comparable interface.

BigInteger and BigDecimal

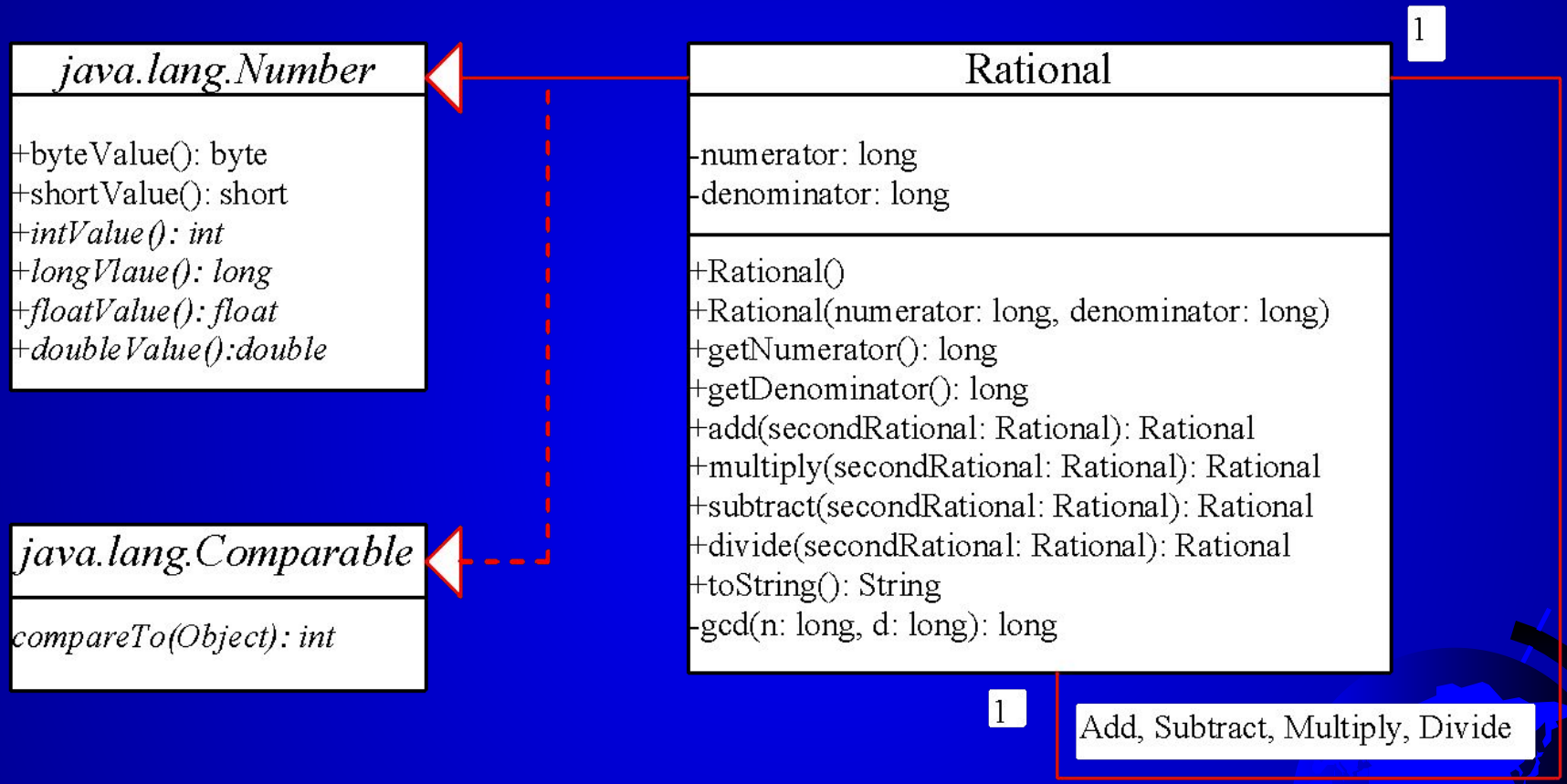
```
BigInteger a = new BigInteger("92233720368547758O7");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 92233720368547758O7 * 2
System.out.println(c);
```

LargeFactorial

Run

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

The `Rational` Class



Rational | TestRationalClass | Run