Abstract Classes and Interfaces

CSE 114, Computer Science 1
Stony Brook University
http://www.cs.stonybrook.edu/~cse114
Abstract Classes and Abstract Methods

Abstract classes are italicized or have the annotation `<abstract>`.

Methods `getArea` and `getPerimeter` are overridden in `Circle` and `Rectangle`. Superclass abstract methods are generally omitted in the UML diagram for subclasses.

The `#` sign indicates protected modifier.

Abstract methods are italicized or have the annotation `<abstract>`.

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

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

The `#` sign indicates protected modifier.
public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;
    protected GeometricObject() {
        dateCreated = new java.util.Date();
    }
    protected GeometricObject(String color, boolean filled) {
        dateCreated = new java.util.Date();
        this.color = color;
        this.filled = filled;
    }
    public String getColor() {   return color;  }
    public void setColor(String color) {  this.color = color;  }
    public boolean isFilled() {   return filled;  }
    public void setFilled(boolean filled) {  this.filled = filled;  }
    public java.util.Date getDateCreated() {   return dateCreated;  }
    public String toString() {
        return "created on " + dateCreated + ", color: " + color + " and filled: " + filled;
    }
    /** Abstract method getArea */
    public abstract double getArea();
    /** Abstract method getPerimeter */
    public abstract double getPerimeter();
}
public class Circle extends GeometricObject {
    private double radius;

    public Circle() {
    }

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    public double getArea() {
        return radius * radius * Math.PI;
    }

    public double getDiameter() {
        return 2 * radius;
    }

    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }

    /* Print the circle info */
    public void printCircle() {
        System.out.println("The circle is created " + getDateCreated() + " and the radius is " + radius);
    }
}
public class Rectangle extends GeometricObject {
    private double width;
    private double height;
    public Rectangle() {  }
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public double getWidth() {    return width;  }
    public void setWidth(double width) {     this.width = width;  }
    public double getHeight() {    return height;  }
    public void setHeight(double height) {    this.height = height;  }
    public double getArea() {
        return width * height;
    }
    public double getPerimeter() {
        return 2 * (width + height);
    }
}

public class TestGeometricObject {
    public static void main(String[] args) {
        // Declare and initialize two geometric objects
        GeometricObject geoObject1 = new Circle(5);
        GeometricObject geoObject2 = new Rectangle(5, 3);
        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();
    }

    /** A method for displaying a geometric object */
    public static void displayGeometricObject(GeometricObject object) {
        System.out.println();
        System.out.println("The area is " + object.getArea());
        System.out.println("The perimeter is " + object.getPerimeter());
    }
}
abstract method in abstract class

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

• 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.

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

- An object cannot be created from abstract class
- An abstract class cannot be instantiated using the `new` operator
- We 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 classes

- A class that contains abstract methods must be abstract
- An abstract class without abstract method:
  - It is possible to define an abstract class that contains no abstract methods.
  - We cannot create instances of the class using the `new` operator.
- This class is used as a base class for defining new subclasses
abstract classes

- A subclass can be abstract even if its superclass is concrete.
- For example, the `Object` class is concrete, but a subclass, `GeometricObject`, is abstract.
abstract classes

• A subclass can override a method from its superclass to define it abstract
  • rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass.
• the subclass must be defined abstract
abstract classes as types

- You cannot create an instance from an abstract class using the `new` operator, but an abstract class can be used as a data type:

  ```java
  GeometricObject c = new Circle(2);
  ```

- The following statement, which creates an array whose elements are of `GeometricObject` type, is correct:

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

- There are only `null` elements in the array!!!
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 for a specific calendar.

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

- `java.util.GregorianCalendar` is for the Gregorian calendar.
The GregorianCalendar Class

- Java API for the GregorianCalendar class: http://docs.oracle.com/javase/8/docs/api/java/util/GregorianCalendar.html
- `new GregorianCalendar()` constructs a default `GregorianCalendar` with the current time
- `new GregorianCalendar(year, month, date)` constructs a `GregorianCalendar` with the specified `year`, `month`, and `date`
  - The `month` parameter is 0-based, i.e., 0 is for January.
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 day of month. 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 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.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>The year of the calendar.</td>
</tr>
<tr>
<td>MONTH</td>
<td>The month of the calendar with 0 for January.</td>
</tr>
<tr>
<td>DATE</td>
<td>The day of the calendar.</td>
</tr>
<tr>
<td>HOUR</td>
<td>The hour of the calendar (12-hour notation).</td>
</tr>
<tr>
<td>HOUR_OF_DAY</td>
<td>The hour of the calendar (24-hour notation).</td>
</tr>
<tr>
<td>MINUTE</td>
<td>The minute of the calendar.</td>
</tr>
<tr>
<td>SECOND</td>
<td>The second of the calendar.</td>
</tr>
<tr>
<td>DAY_OF_WEEK</td>
<td>The day number within the week with 1 for Sunday.</td>
</tr>
<tr>
<td>DAY_OF_MONTH</td>
<td>Same as DATE.</td>
</tr>
<tr>
<td>DAY_OF_YEAR</td>
<td>The day number in the year with 1 for the first day of the year.</td>
</tr>
<tr>
<td>WEEK_OF_MONTH</td>
<td>The week number within the month.</td>
</tr>
<tr>
<td>WEEK_OF_YEAR</td>
<td>The week number within the year.</td>
</tr>
<tr>
<td>AM_PM</td>
<td>Indicator for AM or PM (0 for AM and 1 for PM).</td>
</tr>
</tbody>
</table>
import java.util.*;
public class TestCalendar {
    public static void main(String[] args) {
        // Construct a Gregorian calendar for the current date and time
        Calendar calendar = new GregorianCalendar();
        System.out.println("Current time is " + new Date());
        System.out.println("YEAR:\t" + calendar.get(Calendar.YEAR));
        System.out.println("MONTH:\t" + calendar.get(Calendar.MONTH));
        System.out.println("DATE:\t" + calendar.get(Calendar.DATE));
        System.out.println("HOUR:\t" + calendar.get(Calendar.HOUR));
        System.out.println("HOUR_OF_DAY:\t" + calendar.get(Calendar.HOUR_OF_DAY));
        System.out.println("MINUTE:\t" + calendar.get(Calendar.MINUTE));
        System.out.println("SECOND:\t" + calendar.get(Calendar.SECOND));
        System.out.println("DAY_OF_WEEK:\t" + calendar.get(Calendar.DAY_OF_WEEK));
        System.out.println("DAY_OF_MONTH:\t" + calendar.get(Calendar.DAY_OF_MONTH));
        System.out.println("DAY_OF_YEAR: " + calendar.get(Calendar.DAY_OF_YEAR));
        System.out.println("WEEK_OF_MONTH: " + calendar.get(Calendar.WEEK_OF_MONTH));
        System.out.println("WEEK_OF_YEAR: " + calendar.get(Calendar.WEEK_OF_YEAR));
        System.out.println("AM_PM: " + calendar.get(Calendar.AM_PM));
        // Construct a calendar for January 1, 2020
        Calendar calendar1 = new GregorianCalendar(2020, 0, 1);
        System.out.println("January 1, 2020 is a " +
                            dayNameOfWeek(calendar1.get(Calendar.DAY_OF_WEEK)));
    }
    public static String dayNameOfWeek(int dayOfWeek) {
        switch (dayOfWeek) {
            case 1: return "Sunday";
            case 2: return "Monday";
            case 3: return "Tuesday";
            ... case 7: return "Saturday";
            default: return null;
        }
    }
}
Interfaces

- What is an interface?
  - An interface is a class-like construct that contains only constants and abstract methods.

- Why is an interface useful?
  - An interface is similar to an abstract class, but the intent of an interface is to specify behavior for objects.
    - For example: specify that the objects are comparable, edible, cloneable, …
  - Allows multiple inheritance.
Define an Interface

• Declaration:

```java
public interface InterfaceName {
    // constant declarations;
    // method signatures;
}
```

• Example:

```java
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

- Uses:
  - as a data type for a variable
  - as the result of casting
Interface Example

- The **Edible** interface specifies whether an object is edible

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

- The class **Chicken** implements the **Edible** interface:

```java
class Chicken extends Animal implements Edible {
    public String howToEat() {
        return "Chicken: Fry it";
    }
}
```
interface Edible {
    public abstract String howToEat(); /** Describe how to eat */
}
class Animal {
}
class Chicken extends Animal implements Edible {
    public String howToEat() {
        return "Chicken: Fry it";
    }
}
class Tiger extends Animal {
    /** Does not extend Edible */
}
abstract class Fruit implements Edible {
}
class Apple extends Fruit {
    public String howToEat() {
        return "Apple: Make apple cider";
    }
}
class Orange extends Fruit {
    public String howToEat() {
        return "Orange: Make orange juice";
    }
}
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());
    }
}
Omitting Modifiers in Interfaces

- All data fields are `public static final` in an interface
- All methods are `public abstract` in an interface
- These modifiers can be omitted:

  ```java
  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 `InterfaceName.CONSTANT_NAME`, for example: `T1.K`
Example: The Comparable Interface

// This interface is defined in
// the java.lang package

package java.lang;
public interface Comparable {
    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.

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

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

```java
new String() instanceof String true
new String() instanceof Comparable true
new java.util.Date() instanceof java.util.Date true
new java.util.Date() instanceof Comparable true
```
In UML, the interface and the methods are italicized. Dashed lines and triangles are used to point to the interface.
The return value from the `max` method is of the `Comparable` type. So, we need to cast it to `String` or `Date` explicitly.
Defining Classes to Implement Comparable

- We cannot use the `max` method to find the larger of two instances of `Rectangle`, because `Rectangle` does not implement `Comparable`.
- We can define a new rectangle class `ComparableRectangle` that implements `Comparable`: the instances of this new class are comparable.
public class ComparableRectangle extends Rectangle implements Comparable {
    /** Construct a ComparableRectangle with specified properties */
    public ComparableRectangle(double width, double height) {
        super(width, height);
    }

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

    public static void main(String[] args) {
        ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
        ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
        System.out.println(Max.max(rectangle1, rectangle2));
    }
}
The Cloneable Interface

- Marker Interface: an empty 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: its objects can be cloned using the `clone()` method defined in the Object class

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

- Calendar (in the Java library) implements Cloneable:

```java
Calendar calendar = new GregorianCalendar(2020, 1, 1);
Calendar calendarCopy = (Calendar)(calendar.clone());
System.out.println("calendar == calendarCopy is " + (calendar == calendarCopy));
```

Displays:

calendar == calendarCopy is false

```java
System.out.println("calendar.equals(calendarCopy) is " + calendar.equals(calendarCopy));
```

calendar.equals(calendarCopy) is true
### The Cloneable Interface

- **Throws checked exception:**

```java
public class SomethingCloneable implements Cloneable {
    public boolean equals(Object o) {
        SomethingCloneable s = (SomethingCloneable) o;
        return true;
    }

    public static void main(String[] args) throws CloneNotSupportedException {
        SomethingCloneable s1 = new SomethingCloneable();
        SomethingCloneable s2 = (SomethingCloneable) s1.clone();
        System.out.println("s1 == s2 is "+ (s1 == s2));
        System.out.println("s1.equals(s2) is "+ (s1.equals(s2)));
    }
}
```
Implementing the **Cloneable** Interface

- If we try to create a clone of an object instance of a class that does not implement the Cloneable interface, it throws `CloneNotSupportedException`
- We can override the `clone()` method from the `Object` class to create custom clones.
  - The `clone()` method in the `Object` class creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment; the contents of the fields are not themselves cloned.
  - The `clone()` method returns an `Object` that needs to be casted.
public class House implements Cloneable, Comparable {
    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();}
    public double getId() { return id;}
    public double getArea() { return area;}
    public java.util.Date getWhenBuilt() { return whenBuilt;}
    /** 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;
        }
    }
    /** Implement the compareTo method defined in Comparable */
    public int compareTo(Object o) {
        if (area > ((House)o).area)
            return 1;
        else if (area < ((House)o).area)
            return -1;
        else
            return 0;
    }
}
Shallow vs. Deep Copy

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

Shallow copy: if the field is of reference type, the object’s reference is copied rather than its content.
For deep copying, we can override the clone method with custom object creation

```java
public class House implements Cloneable {
    ...
    public Object clone() { // deep copy
        try {
            House h = (House)(super.clone());
            h.whenBuilt = (Date)(whenBuilt.clone());
            return h;
        } catch (CloneNotSupportedException ex) {
            return null;
        }
    }
    ...
}
```
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.

<table>
<thead>
<tr>
<th></th>
<th>Variables</th>
<th>Constructors</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract class</td>
<td>No restrictions</td>
<td>Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.</td>
<td>No restrictions.</td>
</tr>
<tr>
<td>Interface</td>
<td>All variables must be <strong>public static final</strong></td>
<td>No constructors. An interface cannot be instantiated using the new operator.</td>
<td>All methods must be <strong>public abstract</strong> methods</td>
</tr>
</tbody>
</table>
Interfaces vs. Abstract Classes

- A class can implement any number of interfaces
- An interface can extend another interface
- There is no root for interfaces
Caution: conflict interfaces

- Errors detected by the compiler:
  - If a class implements two interfaces with conflict information:
    - two same constants with different values, or
    - two methods with same signature but different return type
Whether to use an interface or a class?

- **Strong is-a**: a relationship that clearly describes a parent-child relationship - **should be modeled using classes and class inheritance**
  - For example: a staff member is a person

- **Weak is-a (is-kind-of)**: indicates that an object possesses a certain property - **should 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
Wrapper Classes

- Primitive data types in Java ⇒ Better performance
- Each primitive has a wrapper class: Boolean, Character, Short, Byte, Integer, Long, Float, Double
  - The wrapper classes do not have no-arg constructors
  - The instances of all wrapper classes are immutable: their internal values cannot be changed once the objects are created
Wrapper Classes

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

```
java.lang.Object
   - Double
   - Float
   - Long
   - Integer
   - Short
   - Byte
   - Character
   - Boolean
```

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
The **Number Class**

- Each numeric wrapper class extends the abstract Number class:
  - The abstract Number class contains the methods `doubleValue`, `floatValue`, `intValue`, `longValue`, `shortValue`, and `byteValue` to “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
  - Each numeric wrapper class implements the abstract methods `doubleValue`, `floatValue`, `intValue` and `longValue`
The **Integer** and **Double** Classes

**java.lang.Number**
- byteValue(): byte
- shortValue(): short
- intValue(): int
- longValue(): long
- floatValue(): float
- doubleValue(): double

**java.lang.Comparable**
- compareTo(o: Object): int

**java.lang.Integer**
- value: int
- MAX_VALUE: int
- MIN_VALUE: int
- Integer(value: int)
- Integer(s: String)
- valueOf(s: String): Integer
- valueOf(s: String, radix: int): Integer
- parseInt(s: String): int
- parseInt(s: String, radix: int): int

**java.lang.Double**
- value: double
- MAX_VALUE: double
- MIN_VALUE: double
- Double(value: double)
- Double(s: String)
- valueOf(s: String): Double
- valueOf(s: String, radix: int): Double
- parseDouble(s: String): double
- parseDouble(s: String, radix: int): double

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
Wrapper Classes

• 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 `Float` and `Double`, `MIN_VALUE` represents the minimum \textit{positive} float and double values

- The maximum integer: 2,147,483,647
- The minimum positive float: 1.4E-45
- The maximum double floating-point number: 1.79769313486231570e+308d
The Static `valueOf` Methods

- The numeric wrapper classes have a static method `valueOf(String s)` to create a new object initialized to the value represented by the specified string:

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

- Each numeric wrapper class has overloaded parsing methods to parse a numeric string into an appropriate numeric value:

  ```java
  double d = Double.parseDouble("12.4");
  int i = Integer.parseInt("12");
  ```
public class GenericSort {
    public static void main(String[] args) {
        Integer[] intArray={new Integer(2),new Integer(4),new Integer(3)};
        sort(intArray);
        printList(intArray);
    }
    public static void sort(Object[] list) {
        Object currentMax;
        int currentMaxIndex;
        for (int i = list.length - 1; i >= 1; i--) {
            currentMax = list[i];
            currentMaxIndex = i; // Find the maximum in the list[0..i]
            for (int j = i - 1; j >= 0; j--) {
                if (((Comparable)currentMax).compareTo(list[j]) < 0) {
                    currentMax = list[j];
                    currentMaxIndex = j;
                }
            }
            list[currentMaxIndex] = list[i];
            list[i] = currentMax;
        }
    }
    public static void printList(Object[] list) {
        for (int i=0;i<list.length;i++) System.out.print(list[i]+" ");
    }
}

The objects are instances of the Comparable interface and they are compared using the compareTo method.
Sorting an Array of Objects

- Java provides a static `sort` method for sorting an array of `Object` in the `java.util.Arrays` class:
  ```java
  java.util.Arrays.sort(intArray);
  ```
Arrays of Objects

- Arrays are objects:
  - An array is an instance of the `Object` class
  - If $A$ is a subclass of $B$, every instance of $A[]$ is an instance of $B[]$

```java
new int[10] instanceof Object true
new GregorianCalendar[10] instanceof Calendar[]; true
new Calendar[10] instanceof Object[] true
new Calendar[10] instanceof Object true
```

- Although an int value can be assigned to a double type variable, `int[]` and `double[]` are two incompatible types:
  - We cannot assign an `int[]` array to a variable of `double[]` array
Wrapper Classes

- Automatic Conversion Between Primitive Types and Wrapper Class Types:
  - JDK 1.5 allows primitive type and wrapper classes to be converted automatically = boxing

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

(a) Equivalent

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

(b) New JDK 1.5 boxing
```

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

Unboxing
BigInteger and BigDecimal

- **BigInteger** and **BigDecimal** classes in the *java.math* package:
  - For computing with very large integers or high precision floating-point values
    - **BigInteger** can represent an integer of any size
    - **BigDecimal** has no limit for the precision (as long as it’s finite=terminates)
  - Both are *immutable*
  - Both extend the *Number* class and implement the *Comparable* interface.
BigInteger and BigDecimal

BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
18446744073709551614

BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
0.33333333333333333334
import java.math.*;
public class LargeFactorial {
    public static void main(String[] args) {
        System.out.println("50! is \n" + factorial(50));
    }
    public static BigInteger factorial(long n) {
        BigInteger result = BigInteger.ONE;
        for (int i = 1; i <= n; i++)
            result = result.multiply(new BigInteger(i + ""));
        return result;
    }
    
    
    }
Case Study: The **Rational** Class

```
java.lang.Number
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longValue(): long
+floatValue(): float
+doubleValue(): double
```

```
java.lang.Comparable
compareTo(Object): int
```

```
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
```

add, subtract, multiply, divide
public class Rational extends Number implements Comparable{
    private long numerator = 0;
    private long denominator = 1;
    public Rational() { this(0, 1); }
    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;
    }
    public Rational add(Rational secondRational) {
        long n = numerator * secondRational.getDenominator() +
            denominator * secondRational.getNumerator();
        long d = denominator * secondRational.getDenominator();
        return new Rational(n, d);
    }
    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;
    }
}
/** Override the abstract intValue method in java.lang.Number */
public int intValue() { return (int)doubleValue(); }

// ... Override all the abstract *Value methods in java.lang.Number

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

public static void main(String[] args) {
    Rational r1 = new Rational(4, 2);
    Rational r2 = new Rational(2, 3);
    System.out.println(r1 + " + " + r2 + " = " + r1.add(r2));
}