Abstract Classes and Interfaces

CSE 114, Computer Science 1
SUNY Korea

Pradipta De
pradipta.de@sunykorea.ac.kr

Acknowledgement for the slides: Dr. Paul Fodor (CS Stony Brook)
Abstract Classes and Abstract Methods

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

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

The # sign indicates protected modifier.
Abstract methods are italicized.

Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass abstract methods are generally omitted in the UML diagram for subclasses.
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 + "ncolor: " + 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!!!
Legal Definitions?

(a) `class A {
    abstract void unfinished() {
    }
}
`

(b) `public class abstract A {
    abstract void unfinished();
}
`

c. Incorrect

d. Incorrect

e. Correct

(f) `abstract class A {
    abstract void unfinished();
}
`

(f) `abstract class A {
    abstract int unfinished();
}
`

c. Incorrect

d. Incorrect

e. Correct

f. Correct
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.
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](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 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: 	" + calendar.get(Calendar.YEAR));
        System.out.println("MONTH: 	" + calendar.get(Calendar.MONTH));
        System.out.println("DATE: 	" + calendar.get(Calendar.DATE));
        System.out.println("HOUR: 	" + calendar.get(Calendar.HOUR));
        System.out.println("HOUR_OF_DAY: 	" + calendar.get(Calendar.HOUR_OF_DAY));
        System.out.println("MINUTE: 	" + calendar.get(Calendar.MINUTE));
        System.out.println("SECOND: 	" + calendar.get(Calendar.SECOND));
        System.out.println("DAY_OF_WEEK: 	" + calendar.get(Calendar.DAY_OF_WEEK));
        System.out.println("DAY_OF_MONTH: 	" + 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 September 11, 2001
        Calendar calendar1 = new GregorianCalendar(2001, 8, 11);
        System.out.println("September 11, 2001 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 classlike 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, ...**
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();
}
```

All data fields in Interface are “public static final”
All methods are “public abstract”
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.
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";
    }
}
```
Edible interface defines common behavior for edible objects

- all edible objects must define “howToEat” method”

Chicken and Fruit are unrelated classes with respect to Animal class

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.
In UML, the interface and the methods are italicized dashed lines and empty triangles are used to point to the interface

```java
interface java.lang.Comparable

+compareTo(o: Object): int
```

```
Object

String

«interface»

String
```

```
java.lang.Comparable

+compareTo(o: Object): int
```
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 final static` 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
```java
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`
Which of the following are correct?

(a) `interface A { 
    void print() { };
}`

(b) `abstract interface A extends I1, I2 { 
    abstract void print() { };
}`

(c) `abstract interface A { 
    print();
}`

(d) `interface A { 
    void print();
}`

- a. Incorrect (syntax error)
- b. Incorrect (Syntax error at print; Interface can extend multiple interfaces; class cannot extend multiple classes)
- c. Correct
- d. Correct
Error in code?

```java
interface A {
    void m1();
}

class B implements A {
    void m1() {
        System.out.println("m1");
    }
}
```

Default access modifier for method m1 in Class B restricts it to package level.

Modifier of m1 in interface is “public abstract”

Cannot reduce the visibility while implementing the method m1 in class B
Example: The Comparable Interface

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

```
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
```
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.
Sort array of rectangles by area

```java
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();
    }
}
```
The GUI Interfaces

- Handling GUI Events
  - Source object (e.g., button)
  - Listener object contains a method for processing the event
import javax.swing.*;
import java.awt.event.*;

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

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

public class HandleEvent extends JFrame {
    public HandleEvent() {
        JButton jbtOK = new JButton("OK");
        JButton jbtCancel = new JButton("Cancel");
        JPanel panel = new JPanel();
        panel.add(jbtOK);
        panel.add(jbtCancel);
        add(panel); // Add panel to the frame
        OKListenerClass listener1 = new OKListenerClass();
        CancelListenerClass listener2 = new CancelListenerClass();
        jbtOK.addActionListener(listener1); // Register listeners
        jbtCancel.addActionListener(listener2);
    }

    public static void main(String[] args) {
        JFrame frame = new HandleEvent();
        frame.setSize(200, 150);
        frame.setLocation(200, 100);
        frame.setVisible(true);
    }
}
Trace Execution

```java
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
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");
    }
}
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
  - 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 Interfaces

- Calendar (in the Java library) implements Cloneable:

```java
Calendar calendar = new GregorianCalendar(2014, 4, 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
```
To define a custom class that implements the Cloneable interface, the class must override the clone() method. It throws CloneNotSupportedException if it is not supported.
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() throws CloneNotSupportedException {
        return super.clone();
    }
    /** 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

Note: For deep copying, we can override the clone method with custom object creation.
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;
}
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 <strong>constructor chaining</strong>. 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 methods</strong></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.
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`
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:

```java
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 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");
  ```
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

```java
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 
" + 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;
    }
}

30414093201713378043612608166064768844377641
568960512000000000000