1. 抽象类&抽象方法（Abstract Class & Abstract Method）

我们回到我们上一章中举的GeometricObject类的例子中，在那个例子中我们声明了一个抽象类GeometricObject，当时我们在其的子类Cricle和Rectangle中分别有不同的getArea()和getPerimeter()方法，其实我们可以将这两个方法声明为抽象方法，这样我们就可以让Circle和Rectangle作为子类的时候必须实现这两个方法。如下图所示。

我们为什么要这么做呢？这样我们其实可以保证代码的一致性一致性和可预测性，使得其他开发者更容易理解和使用这些类，而且计算面积是对于所有图形都有的，我们可以更好地理解这些类之间的关系。这里抽象类和抽象方法的组合与后面将要介绍地接口类似，但这里我们先专注于抽象类和抽象方法。
下面是上面UML图的代码，上面的UML图中抽象类用斜体表示或带有注释<abstract>，抽象方法同理，且我们使用#表示 protected 关键字，此外在子类的UML图中，超类抽象方法通常被省略。

```
public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    protected GeometricObject() {
        this.dateCreated = new Date();
    }

    protected GeometricObject(String color, boolean filled) {
        this.dateCreated = new 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() {
        super();
    }

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

    public double getRadius() {
        return radius;
    }

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

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

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

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

```
public class Rectangle extends GeometricObject {
    private double width;
    private double height;

    public Rectangle() {
        // super();
    }

    public Rectangle(double width, double height) {
```

```
        this();
        this.width = width;
        this.height = height;
    }

    public Rectangle(double width, double height, String color,
                     boolean filled) {
        super(color, filled);
        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;
    }

    @Override
    public double getArea() {
        return width * height;
    }

    @Override
    public double getPerimeter() {
        return 2 * (width + height);
    }
}
```

```
public class TestGeometricObject1 {
    public static void main(String[] args) {
        // Declare and initialize two geometric objects
        GeometricObject geoObject1 = new Circle(5);
        GeometricObject geoObject2 = new Rectangle(5, 3);
        // Display circle
        displayGeometricObject(geoObject1);
        // Display rectangle
        displayGeometricObject(geoObject2);
        System.out.println("The two objects have the same area? " +
                equalArea(geoObject1, geoObject2));
    }

    /**
     * A method for displaying a geometric object
     */
```

```
    public static void displayGeometricObject(GeometricObject object) {
        System.out.println(object); // Calls object's toString() method
        System.out.println("The area is " + object.getArea());
        System.out.println("The perimeter is " + object.getPerimeter());
    }

    /**
     * A method for comparing the areas of two geometric objects
     */
    public static boolean equalArea(GeometricObject object1,
                                    GeometricObject object2) {
        return object1.getArea() == object2.getArea();
    }
}
```

1.1 抽象方法

前面的例子已经使用了抽象方法，这里强调一下抽象方法是定义在抽象类中的，它们不能存在于普通类中。
它只包含方法签名，没有方法体。它使用 abstract 关键字声明。

在一个非抽象的子类中，如果它继承自一个抽象的父类，那么它必须实现父类中的所有抽象方法，也就是前面的Circle类和Rectangle类必须实现getArea()和getPerimeter()方法，这样可以保证多态性的一致性。
因此对于这样的代码。

```
abstract class A {
    abstract void m();
}
class B extends A {
    void m() {
        // 方法实现
    }
}
```

class B 就必须提供抽象方法 m() 的具体实现。

1.2 抽象类的抽象子类

一个抽象子类继承自另一个抽象超类时，这个子类既可以实现从抽象超类继承的所有抽象方法，也可以将实现抽象方法的责任推迟到非抽象子类。
下图为第一种情况。

代码如下。

```
abstract class A {
    abstract void m();
}
abstract class B extends A {
    void m() {
        // 方法实现
    }
}
class C extends B {
    // 继承 B 的实现，不需要再次实现 m()
}
```

下图是第二种情况。

代码如下。

```
abstract class A {
    abstract void m();
}
abstract class B extends A {
    // 没有实现 m() 方法，所以 B 也必须是抽象类
}
public class C extends B {
    void m() {
        // 提供了 m() 方法的具体实现
    }
}
```

1.3 非抽象父类的子类可以是抽象类

子类可以是抽象的，即使其超类是具体的。
例如，Object 类是具体的，但子类 GeometricObject 是抽象的。

同理，我们其实子类可以覆盖具体超类中的方法，将其定义为抽象方法。这样做有两个目的：
1.可以让其强制其的子类实现该方法。
2.将超累中的方法实现在子类中无效。如果超类中的方法实现对于子类来说不适用或无效，子类可以通过将其重新定义为抽象方法来避免使用超类的实现。

1.4 抽象类

我们现在回到抽象类上，首先我们需要弄清楚一点：我们可以定义不包含抽象方法的抽象类，但可以包含具体方法和属性。
这种抽象类用作定义新子类的基础类，通过继承这种抽象类，子类可以继承基类中的属性和方法，同时也可以添加或修改自己的属性和方法。

抽象类不能直接被实例化，即不能使用 new 关键字来创建抽象类的对象。
所以下面的代码会导致编译错误。

```
GeometricObject o = new GeometricObject();
```

尽管抽象类不能被直接实例化，但仍然可以在抽象类中定义构造函数。这些构造函数在子类的构造过程中被调用，这个过程称为构造函数链（constructor chaining）。
例如，GeometricObject 的构造函数被 Circle 和 Rectangle 类的构造函数调用。

1.4.1 抽象类可以作为数据类型

在声明变量时，可以使用抽象类作为数据类型。例如，可以声明一个 GeometricObject 类型的变量 c，然后将其初始化为 Circle 类型的对象。

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

在这个例子中，变量 c 被声明为 GeometricObject 类型，但它引用的是一个 Circle 对象。

因此我们可以创建一个数组，其元素类型为抽象类。例如，可以创建一个 GeometricObject 类型的数组 geo，该数组可以包含多个几何对象。

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

在数组的元素被具体对象初始化之前，它们都是 null。直到将它们指向具体的对象，如下所示。

```
geo[0] = new Circle();
geo[1] = new Rectangle();
```

1.5 Calendar 抽象类和子类 GregorianCalendar

java.util.Calendar是一个抽象类，用于从 java.util.Date对象中提取详细的日期和时间信息，如年、月、日、时、分和秒。
Calendar 的子类可以实现特定的日历系统，如公历（Gregorian calendar）、农历（Lunar Calendar）和犹太历（Jewish calendar）等。
java.util.GregorianCalendar 正是 Calendar 的一个具体子类，用于实现现代公历（即格里高利历）。
GregorianCalendar的官方文档

下面是 Calendar 和 GregorianCalendar 的UML图。

其中需要注意的是以下几个
1.new GregorianCalendar():
这个构造函数用于创建一个默认的 GregorianCalendar 实例，该实例代表当前的日期和时间。

2.new GregorianCalendar(int year, int month, int dayOfMonth):
这个构造函数允许你创建一个 GregorianCalendar 实例，并指定特定的年份、月份和日期。
需要注意的是，月份参数是基于0的。这意味着：
0 表示一月（January）
1 表示二月（February）
...
11 表示十二月（December）

3.get(int field) 方法定义在 Calendar 类中，用于获取 Calendar 对象中特定字段的值。这些字段代表日期和时间的不同组成部分，如下所示。
YEAR：日历的年份。
MONTH：日历的月份，以0为基准，即0代表一月（January）。
DATE：日历的日期（一个月中的第几天）。
HOUR：日历的小时数（12小时制）。
HOUR_OF_DAY：日历的小时数（24小时制）。
MINUTE：日历的分钟数。
SECOND：日历的秒数。
DAY_OF_WEEK：一周中的第几天，以1为基准，即1代表星期日（Sunday）。

DAY_OF_MONTH：与 DATE 相同，表示一个月中的第几天。
DAY_OF_YEAR：一年中的第几天，以1为基准，即1代表一年的第一天。
WEEK_OF_MONTH：一个月中的第几周。
WEEK_OF_YEAR：一年中的第几周。
AM_PM：上午（AM）或下午（PM）的指示器，0 表示 AM，1 表示 PM。

下面给出一个详细的例子。

```
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:\t" +
calendar.get(Calendar.DAY_OF_YEAR));
        System.out.println("WEEK_OF_MONTH:\t" +
calendar.get(Calendar.WEEK_OF_MONTH));
        System.out.println("WEEK_OF_YEAR:\t" +
calendar.get(Calendar.WEEK_OF_YEAR));
        System.out.println("AM_PM:\t" + 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;
        }
    }
}
```

2. 接口（Interface）

我们现在说接口，接口与前面的抽象类和抽象方法的组合有些类似，但是其有一些不同。
首先，接口是一种类似于类的结构，但它只包含抽象方法和常量（在 Java 8 及以后版本中，接口也可以包含默认方法和静态方法）。
接口的主要目的是为对象指定行为。通过定义一组方法，接口可以描述一个对象应该能够执行哪些操作。
接口可以用来定义对象的契约或能力，例如，可以定义一个接口来指定对象是否可比较（Comparable）、可食用（Edible）、可克隆（Cloneable）等。
虽然接口和抽象类都可以包含抽象方法，但接口的意图更侧重于定义对象的行为和能力，而不是提供部分实现。
与抽象类不同，一个类可以实现多个接口。这种特性称为多重继承，它允许类继承多个接口中定义的行为。这种灵活性使得接口成为定义行为规范的强大工具，因为一个类可以同时遵守多个接口的规范。
所以接口的主要作用更多是规定其的子类需要遵守什么样的规范，提供一个通用的实现框架。
因此其的代码框架如下。

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

下面给出一个例子，我们先规定一个可食用的接口，这个接口会规定一个 hotToEat() 的抽象方法。

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

所以当有子类实现这个接口的时候就需要提供 howToEat() 方法的具体实现，返回一个描述如何食用这个子类的字符串。

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

下面展示了一个较为复杂的例子。

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

这里和前面类似，当一个抽象子类实现了一个接口时，它既可以具体实现继承的抽象方法，也可以将实现的责任交给自己的子类。
这里 Fruit 是一个抽象类，它实现了 Edible 接口。Fruit 类没有具体实现 howToEat() 方法，而是将实现的责任交给了它的子类 Apple 和 Orange。
这里还使用了 instanceof 来检查一个对象是否实现了特定的接口。

2.1 可省略的修饰符

接口中的所有数据字段默认是 public、static 和 final 的。
接口中的所有方法默认是 public 和 abstract 的。
因此对于这些默认的修饰符，我们都是可以省略的。
因此如下所示。

```
public interface T1 {
    public static final int K = 1;
    public abstract void p();
}
```

这个代码等价于下面的代码。

```
public interface T1 {
    int K = 1;
    void p();
}
```

接口中定义的常量（即 public static final 类型的变量）可以通过 InterfaceName.CONSTANT_NAME 的方式访问。
所以这里可以通过 T1.K 访问 T1 接口的常量 K。

2.2 接口的特征

在 Java 中，接口被当作一种特殊的类来处理。
每个接口在编译后都会生成一个独立的字节码文件（.class 文件），就像普通类一样。
接口不能实例化，类似于抽象类，不能使用 new 关键字来创建接口的实例。
接口可以像抽象类一样用作变量的数据类型。

所以接口也可以作为类型转换（casting）的结果。
如下面代码所示。

```
public class TestInterfaces {
    public static void main(String[] args) {
        Edible edible;

        // 使用接口作为变量的数据类型
        edible = new Apple();
        edible.howToEat();

        edible = new Orange();
        edible.howToEat();

        // 使用接口作为类型转换的结果
        Object obj = new Apple();
        Edible edibleObj = (Edible) obj;
        edibleObj.howToEat();
    }
}
```

2.3 Comparable 接口

Comparable 接口定义在 java.lang 包中，并且被 Arrays.sort 使用从而进行排序。

```
package java.lang;

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

许多 Java 标准库中的类（例如 String 和 Date）实现了 Comparable 接口，以定义对象的自然排序顺序。
所以我们可以验证以下。

```
public class TestComparable {
    public static void main(String[] args) {
        String str = new String();
        Date date = new java.util.Date();

        // 检查 String 对象是否实现了 Comparable 接口
        System.out.println("new String() instanceof Comparable " + (str
instanceof Comparable));

        // 检查 Date 对象是否实现了 Comparable 接口
        System.out.println("new java.util.Date() instanceof Comparable " + (date
instanceof Comparable));
    }
}
```

即如图所示。

在 UML 类图中，接口和接口中的方法用斜体表示，虚线和三角形表示类实现接口的关系。

2.3.1 实现 Comparable 接口

我们现在可以写一个通用的 max 方法，用于找出两个对象中的最大值。
有两种方案，一种是接受 Comparable 接口的对象作为参数，另一种是接受Object 类型的对象作为参数，但这种方案需要通过类型转换将 Object 类型的对象转换为 Comparable 类型，再使用 compareTo 方法进行比较。
方案一：

```
public class Max {
    public static Comparable max(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

方案二：

```
public class Max {
    public static Object max(Object o1, Object o2) {
        if (((Comparable)o1).compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}
```

需要注意的是，第二个版本仍需要保证传入的对象实际上是 Comparable 类型的，否则在运行时会抛出 ClassCastException。
还需要注意第一个方法返回的是 Comparable 类型，第二个返回的是 Object 类型，所以使用返回值时，可能需要进行类型转换以匹配调用方法时的期望类型。

就像我们前面说的，我们需要保证传入的对象实际上是 Comparable 类型的，所以我们现在无法使用 max 方法去找到两个 Rectangle 实例中较大的一个，因为 Rectangle 类没有实现 Comparable 接口。我们可以定义一个新的 Rectangle 类，命名为 ComparableRectangle，并让它实现 Comparable 接口。这样，ComparableRectangle 类的实例就可以进行比较了。

代码如下。

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

    /** Implement the compareTo method defined in Comparable */
    @Override
    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));
    }
}
```

2.4 Cloneable 接口

Cloneable 是一个空接口，它不包含任何常量（constants）或方法（methods）。它的主要作用是作为一个标记（marker），向编译器（compiler）和 Java 虚拟机（JVM）表明实现该接口的类具有某些理想的属性。

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

当一个类实现了 Cloneable 接口，这意味着这个类被标记为“可克隆”（cloneable）。
在 Java 中，Object 类（所有类的根类）定义了一个 clone() 方法，这个方法可以用来创建当前对象的一个副本（shallow copy）。如果一个类实现了 Cloneable 接口，那么它的实例可以使用 clone() 方法进行克隆。
当然开发者可以在自己的类中重写（override）Object 类的 clone() 方法，以提供自定义的克隆行为。

例如我们前面提到的 Calendar 类就实现了 Cloneable，这意味着 Calendar 对象可以被克隆。

```
Calendar calendar = new GregorianCalendar(2022, 1, 1);
Calendar calendarCopy = (Calendar)(calendar.clone());
System.out.println("calendar == calendarCopy is " + (calendar == calendarCopy));
```

这里需要显式转换的原因是 clone() 方法在 Object 类中被定义，所以返回的是 Object 类型，以确保兼容性。
而上面代码的结果如下。

```
calendar == calendarCopy is false
```

因为克隆操作创建了一个新的对象，但它与原始对象在内存中有不同的引用（地址），所以这里直接比较引用的结果是false。
如果我们使用equals()方法进行比较，我们就会获得true。

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

结果如下。

```
calendar.equals(calendarCopy) is true
```

2.4.1 实现 Cloneable 接口

我们前面说 clone() 方法是在 Object 类下的，那我们如果如果尝试克隆一个没有实现 Cloneable 接口的类的对象会发生什么呢?
结果将会抛出 CloneNotSupportedException 异常，从而防止未经允许的对象被克隆。
Object 类中的 clone() 方法用于创建当前对象的一个新实例，并且初始化新对象的所有字段，使其内容与原对象的相应字段完全相同，就像通过赋值操作（使用一种称为反射的技术）一样。
需要注意的是，引用数据字段的内容不会被克隆，即如果对象中包含对其他对象的引用，克隆后的新对象将与原对象共享这些引用。
再次强调一下clone() 方法返回一个 Object 类型的对象，因此使用返回值时需要注意类型转换。
我们可以通过重写 Object 类的 clone() 方法来创建自定义的克隆行为。

现在给出一些实现 Cloneable 的例子。

```
public class SomethingCloneable implements Cloneable {
    public boolean equals(Object 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));
        // 应该输出 false，因为 s1 和 s2 是不同的对象引用
        System.out.println("s1.equals(s2) is " + s1.equals(s2));
        // 应该输出 true，因为 equals 方法总是返回 true
    }
}
```

这里或许可能会想 SomethingCloneable 没有继承 Object 类，怎么能使用 clone() 方法的?
在 Java 中，所有的类都隐式地继承自 Object 类，除非显式地继承自另一个类，所以能使用 clone() 方法。

```
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;
        this.whenBuilt = new java.util.Date();
    }

    public int getId() {
        return id;
    }

    public double getArea() {
        return area;
    }

    public java.util.Date getWhenBuilt() {
        return whenBuilt;
```

```
    }

    // Override the clone method defined in the Object class
    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 (this.area > ((House)o).area)
            return 1;
        else if (this.area < ((House)o).area)
            return -1;
        else
            return 0;
    }
}
```

这里的 House 类就实现了两个接口，并且重写了来自 Object 类的 clone() 方法。

2.4.2 浅拷贝（Shallow Copy）和深拷贝（Deep Copy）

对于前面的代码，如果我们使用 clone() 方法。

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

由于这里 whenBuilt 是引用类型，所以这里克隆直接复制引用本身，而不复制引用的对象，所以现在 house1 和 house2 共享同一个 Date 对象， 因此如果修改 house 1 的 whenBuilt 字段的任何修改都会影响 house2 的 whenBuilt 字段，反之亦然。

这就是浅拷贝（Shallow Copy），如果字段是引用类型（如 Date 对象），则只复制引用本身，而不复制引用的对象。

而在在深拷贝（Deep Copy）中，引用类型字段会被递归地复制。这意味着不仅复制引用，还复制引用的对象。
这样，house1 和 house2 将拥有独立的 Date 对象，修改一个不会影响到另一个。
我们将 House 类里重写的 clone 方法中加入对引用类型的克隆。

```
 @Override
   public Object clone() {
       try {
           House h = (House) super.clone(); // Perform a shallow copy
           h.whenBuilt = (Date) whenBuilt.clone(); // Perform a deep copy for
Date object
           return h;
       } catch (CloneNotSupportedException ex) {
           return null;
        }
    }
```

2.4.3 接口和抽象类的对比

2.4.3.1 组成部分

1. 变量（Variables）：
   接口：接口中的所有变量都必须是 public、static 和 final 的常量。这意味着接口中定义的变量是全局常量，不能被修改。
   抽象类：抽象类可以有普通的变量，即实例变量或类变量，这些变量可以被修改。
2. 构造函数（Constructors）：
   接口：接口没有构造函数。由于接口不能被实例化，它不需要构造函数。
   抽象类：抽象类有构造函数，这些构造函数通过子类构造链（constructor chaining）被子类调用。尽管抽象类不能直接实例化，但子类可以创建抽象类的实例。
3. 方法（Methods）：
   接口：接口中的每个方法都必须是 public 和 abstract 的，即它们只定义了方法签名，没有实现。接口不能包含具体方法的实现。
   抽象类：抽象类可以包含具体方法（即有实现的方法）。抽象类中的方法可以是 public、protected 或 private，并且可以包含方法体（实现）。

下表也是这几点的对比。

2.4.3.2 继承（Inheritance）

在继承（Inheritance）上，接口又有以下几种特性：

1. 接口可以扩展任意数量的其他接口：
   一个接口可以继承（或称为“扩展”）任意数量的其他接口。这意味着接口可以组合多个接口中定义的行为，从而为实现类提供更多的功能。
2. 接口没有根：
   与类继承体系不同，接口没有根接口。类继承体系中，所有类最终都继承自 Object 类，但接口可以独立存在，不需要继承自其他接口。
3. 类可以实现任意数量的接口：
   一个类可以实现（或称为“实现”）任意数量的接口。这允许类具有多个接口定义的行为，从而支持多重继承，这是 Java 实现多重继承的一种方式。
   如图所示。

   由于第三点所以当一个类实现了两个或多个接口，而这些接口中存在冲突的信息时，就会发生接口冲突。
   冲突的信息可能包括：
   两个相同的常量（constants）具有不同的值。例如，两个接口都定义了一个名为 MAX_VALUE 的常量，但赋予了不同的数值。

两个具有相同签名（signature）但返回类型（return type)不同的方法。例如，两个接口都有一个名为 getValue 的方法，但一个返回 int 类型，另一个返回 double 类型。

为了解决这种冲突，类需要明确指定它将使用哪个接口的常量或方法。例如，类可以重新定义一个新的常量或方法来解决冲突，如下所示。

```
public class MyClass implements InterfaceA, InterfaceB {
    // 使用 InterfaceA 的 MAX_VALUE
    int valueA = InterfaceA.MAX_VALUE;

    // 使用 InterfaceB 的 MAX_VALUE
    int valueB = InterfaceB.MAX_VALUE;

    // 实现两个接口的方法
    public int getValueA() {
        return InterfaceA.MAX_VALUE;
    }

    public double getValueB() {
        return InterfaceB.MAX_VALUE;
    }
}
```

2.4.3.3 什么时候该使用类/接口呢?

强关系（Strong）：
当存在一个清晰的“是一个”（is-a）关系时，应该使用类继承（class inheritance）。
这种关系描述了一种父子关系，例如，一个学生是一个人，这表示学生继承自人。
在这种情况下，类继承是合适的，因为它允许子类继承父类的所有属性和方法，并可以添加或修改这些属性和方法。

弱关系（Weak）：
当存在一种“具有”（has-a 或 is-kind-of）关系时，表明对象具有某种属性或行为。
例如，所有的字符串（String）都是可比较的（Comparable），所有的日期（Date）也都是可比较的。
在这种情况下，接口是合适的选择，因为接口允许类声明它们实现了某种行为或属性，而不需要继承自其他类。

使用接口来绕过单继承限制：
在 Java 中，类不能直接继承自多个类（即 Java 不支持多重继承），但可以实现多个接口。
接口允许类实现多个行为或属性，从而绕过 Java 的单继承限制，实现类似多重继承的效果。所以我们可以使用接口来绕过单继承限制。

3. 包装类（Wrapper Classes）

包装类（Wrapper Classes）是用于将基本数据类型（primitive data types）封装为对象。
Java 中的基本数据类型（如 int, double, boolean 等）不是对象，它们是存储在栈上的简单值。因此它们提供更好的性能，因为它们直接存储值，没有对象开销。
包装类是这些基本数据类型的类版本，它们是对象，可以存储在堆上。比如数据结构（如 ArrayList）期望对象作为元素，而不是直接存储基本数据类型的值，我们就可以使用包装类。
包装类包括：Boolean, Character, Short, Byte, Integer, Long, Float, Double。

从这里我们发现这些类还都实现了 Comparable 接口，所以它们的实例可以通过 compareTo 方法进行

比较。
而且包装类的对象一旦创建，其值就不能被改变。这种不可变性使得包装类的对象在多线程环境中更安全，因为它们的状态不会意外地被修改。
每个包装类还都重写了 Object 类中定义的 toString 和 equals 方法。

3.1 Number 类

每个数值包装类（如 Integer、Double、Float 等）都继承自抽象的 Number 类。
Number 类是一个抽象类，它定义了一组方法，这些方法用于将包装类的对象转换为对应的基本数据类型值。
Number 类包含一组方法，如 doubleValue()、floatValue()、intValue()、longValue()、shortValue() 和 byteValue()，这些方法的目的是将对象转换为基本数据类型值。
这些方法中，doubleValue()、floatValue()、intValue() 和 longValue() 是抽象方法，需要在每个具体的包装类中实现。
byteValue() 和 shortValue() 方法不是抽象的，它们直接返回对象值的 byte 和 short 类型，即 (byte)intValue() 和 (short)intValue()。
每个具体的数值包装类（如 Integer、Double、Float 等）都实现了 Number 类中的抽象方法。
下图给出了 Integer 和 Double 类的例子，它们继承 Number 类，又都实现了 Comparable 类。

3.2 创建 Wrapper 类对象

我们可以通过基本数据类型值或表示数值的字符串创建包装对象。
例子如下。

```
// 使用基本数据类型值创建包装对象
Integer integerObject = new Integer(10);
Double doubleObject = new Double(3.14);

// 使用字符串创建包装对象
Integer integerFromString = new Integer("20");
Double doubleFromString = new Double("4.15");
```

后面我们可以使用静态方法创建 Wrapper 类对象，将在后文进行介绍。

3.3 MAX_VALUE 和 MIN_VALUE

在数值包装类（Numeric Wrapper Classes）中有两个常量—— MAX_VALUE 和 MIN_VALUE。
每个数值包装类（如 Integer、Double、Float 等）都定义了两个常量：MAX_VALUE 和 MIN_VALUE。
MAX_VALUE 表示对应基本数据类型的最大值。
MIN_VALUE 表示对应基本数据类型的最小正值（对于 Float 和 Double 类型，MIN_VALUE 表示最小的正浮点数）。
其中Integer.MAX_VALUE：表示 int 类型的最大值，即$2,147,483,647$。
Float.MIN_VALUE：表示 float 类型的最小正值，即$1.4E-45$（即1.4×10^{-45}）。
Double.MAX_VALUE：表示 double 类型的最大值，即 $1.79769313486231570e+308$（即 $1.7976931348623157\times10^{308}$）。
也可以由下面的代码展示出来。

```
public class WrapperConstantsExample {
    public static void main(String[] args) {
        System.out.println("Integer.MAX_VALUE: " + Integer.MAX_VALUE);
        System.out.println("Float.MIN_VALUE: " + Float.MIN_VALUE);
        System.out.println("Double.MAX_VALUE: " + Double.MAX_VALUE);
    }
}
```

3.4 静态方法和解析方法

数值包装类（如 Double 和 Integer）提供了一个名为 valueOf 的静态方法，该方法用于根据指定的字符串创建一个新的对象，并将其初始化为字符串所表示的数值。
例子如下。

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

通过静态方法我们能将数值字符串创建为对象，要是想获得对应的数据类型，我们可以使用解析方法，将数值字符串解析为相应的数值类型。
例子如下。

```
double d = Double.parseDouble("12.4");
int i = Integer.parseInt("12");
```

3.5 自动装箱（boxing）和拆箱（unboxing）

自 Java 1.5 版本起，Java 允许基本数据类型（primitive types）和包装类类型（wrapper class types）之间自动转换。
装箱（boxing）：当需要对象时，基本数据类型会自动转换为包装类型。
拆箱（unboxing）：当需要基本数据类型时，包装类型会自动转换为基本数据类型。
示例如下。

```
public class BoxingUnboxingExample {
    public static void main(String[] args) {
        // 装箱：基本数据类型转换为包装类类型
        Integer[] intArray = {2, 4, 3};
        System.out.println("intArray: " + java.util.Arrays.toString(intArray));

        // 拆箱：包装类类型转换为基本数据类型
        int n = intArray[0] + intArray[1] + intArray[2];
        System.out.println("Sum: " + n);
    }
}
```

4. Arrays 对象

在 Java 中，数组是对象，它们是 Object 类的实例。
我们可以使用 instanceof 来验证这一点。

```
new int[10] instanceof Object
```

结果是 true，所以 数组的确是 Object 类的实例。

如果 A 是 B 的子类，那么 A[] 的每个实例也是 B[] 的实例。

```
new GregorianCalendar[10] instanceof Calendar[]
new Calendar[10] instanceof Object[]
new Calendar[10] instanceof Object
```

这里的结果都是 true。

尽管我们可以将 int 数值直接赋值给 double 类型变量， 但是 int[] 和 double[] 是两种不兼容的类型，因为它们不是类。
所以我们无法将 int[] 数组赋值给 double[]，如下面的代码会造成编译错误。

```
double[] a = new int[10];
```

4.1 排序方法

java.util.Arrays 类提供了一个静态方法 sort，用于对对象数组进行排序。
这个方法接受一个对象数组作为参数，并使用 Comparable 接口来确定排序顺序。
下面的代码定义了一个 sort() 方法实现排序， 当然也可以直接使用 Arrays.sort() 方法进行排序。

```
public class GenericSort {
    public static void main(String[] args) {
        Integer[] intArray = {new Integer(2), new Integer(4), new Integer(3)};
        sort(intArray); // 或者使用 Arrays.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;
            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] + " ");
        }
    }
}
```

5. BigInteger 和 BigDecimal 类

BigInteger 和 BigDecimal 类位于 java.math 包中，专门用于处理非常大的整数或高精度浮点数值的计算。
BigInteger 用于计算非常大的整数，它可以表示任意大小的整数，不受 Java 基本数据类型（如 int 或 long）大小的限制。
BigDecimal 用于处理高精度的浮点数，它没有精度限制（只要数值是有限的、可终止的）。
需要注意的是它们都是不可变的（immutable）对象，且都继承自 Number 类，并实现了 Comparable 接口。
下面分别给出它们的示例。

```
BigInteger a = new BigInteger("92233720368547758O7");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c); // 输出 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); // 输出 0.33333333333333333334
```

下面的代码展示了其的作用。

```
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;
   }
}
```

它打印了 50 的阶乘，这个数字很大，所以使用 BigInteger 。

6. 练习

6.1 基础练习

1.运行下列代码会发生什么?
A. 编译错误，因为Y不是抽象类。
B. 编译错误，因为Z不是抽象类。
C. 运行错误，因为抽象类被实例化。
D. 对象被创建。

```
abstract class X {
```

```
    abstract void methodX();
}

class Y extends X {
}

class Z extends Y {
    void methodX() {
    }
}

public class Test {
    public static void main(String[] args) {
        Z obj = new Z();
    }
}
```

运行后会出现报错：java: Y不是抽象的, 并且未覆盖X中的抽象方法methodX()
因此应该是选项A。

2.下面的代码有哪些问题?
A. 使用 extends 代替 implements。
B. howToEat()必须是 public。
C. howToEat()必须返回一个字符串。
D. 以上所有。

```
interface Edible {
    public abstract String howToEat();
}
public class Banana extends Edible {
    String howToEat() {
        return 'Peel and eat';
    }
}
```

比较简单，选D。

3.下面哪一个选项中的代码可以替换下面的代码?
A. return this.id + ((Student)o).id;
B. return this.id - ((Student)o).id;
C. return this.id * ((Student)o).id;
D. return this.id / ((Student)o).id;

```
public class Student implements Comparable {
    int id;

    public int compareTo(Object o) {
        if (this.id > ((Student)o).id)
            return 1;
        if (this.id == ((Student)o).id)
            return 0;
        return -1;
    }
}
```

在 compareTo 方法中，只要返回值满足以下条件即可：
如果当前对象小于参数对象，返回负整数；
如果当前对象等于参数对象，返回 0；
如果当前对象大于参数对象，返回正整数。
所以选B。

4.下列代码会报什么错?
A. 第三行编译错误。
B. 第三行运行错误。
C. 第四行编译错误。
D. 第四行运行错误。

```
public class AutomaticUnboxing {
    public static void main(String[] args) {
       Integer num = null;
       int x = num; // 这里尝试将 Integer 类型的对象自动拆箱为 int 类型
       System.out.println(x);
   }
}
```

运行时抛出异常 NullPointerException，自动拆箱（unboxing）要求包装类的对象引用必须不为 null，因此选D。

6.2 实例展示

6.2.1 有理数类

现在定义一个有理数类，其UML图如下。

实现的代码如下。

```
import java.lang.*;
import java.math.*;

public class Rational extends Number implements Comparable<Rational> {
    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;
    }

    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;
    }

    public Rational add(Rational secondRational) {
        long n = numerator * secondRational.getDenominator() +
                 denominator * secondRational.getNumerator();
        long d = denominator * secondRational.getDenominator();
        return new Rational(n, d);
    }

    public Rational subtract(Rational secondRational) {
        return add(secondRational.negate());
    }

    public Rational negate() {
        return new Rational(-numerator, denominator);
    }

    public Rational multiply(Rational secondRational) {
        long n = numerator * secondRational.getNumerator();
        long d = denominator * secondRational.getDenominator();
        return new Rational(n, d);
    }

    public Rational divide(Rational secondRational) {
        return multiply(secondRational.reciprocal());
    }

    public Rational reciprocal() {
        return new Rational(denominator, numerator);
    }

    @Override
    public int intValue() { return (int)doubleValue(); }
    public double double doubleValue() { return ((double)numerator)/denominator;
}

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

    @Override
    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;
    }

    public static void main(String[] args) {
        Rational r1 = new Rational(1, 2);
        Rational r2 = new Rational(2, 3);
        System.out.println(r1 + " + " + r2 + " = " + r1.add(r2));
    }
}
```

6.3 进阶练习

6.3.1 ArrayList的洗牌算法

用下列的代码写一个洗牌算法出来。

```
public static void shuffle(ArrayList<Number> list)
```

提示如下。
我们可以使用Fisher-Yates洗牌算法来打乱 ArrayList<Number> 中数字。
Fisher-Yates洗牌算法的步骤如下：

1. 初始化：从列表的最后一个元素开始，向前遍历列表。
2. 随机选择：对于列表中的每个元素，在它和它之前的所有元素（包括自己）中随机选择一个元素。
3. 交换：将当前元素与随机选中的元素进行交换。
4. 重复：继续向前遍历列表，重复步骤2和3，直到到达列表的开始。

它的特点如下：

1. 该算法的时间复杂度为 O(n)，其中 n 是列表的长度。这是因为每个元素只被访问一次。
2. 每个元素都有相同的机会出现在列表的任何位置，确保了洗牌的随机性和公平性。
3. 算法不需要额外的存储空间，直接在原列表上进行操作，空间复杂度为 O(1)。
   最后的代码如下。

```
import java.util.ArrayList;
import java.util.Random;

public class ShuffleArrayList {
    public static void main(String[] args) {
        // 创建一个包含数字的ArrayList
        ArrayList<Number> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.add(6);
        list.add(7);
        list.add(8);
        list.add(9);
        list.add(10);

        // 打印原始列表
        System.out.println("Original list: " + list);

        // 调用shuffle方法打乱列表
        shuffle(list);

        // 打印打乱后的列表
        System.out.println("Shuffled list: " + list);
    }

    public static void shuffle(ArrayList<Number> list) {
        Random random = new Random(); // 创建一个随机数生成器

        // Fisher-Yates洗牌算法
```

```
        for (int i = list.size() - 1; i > 0; i--) {
            // 生成一个从0到i（包含i）的随机索引
            int j = random.nextInt(i + 1);

            // 交换索引i和j处的元素
            Number temp = list.get(i);
            list.set(i, list.get(j));
            list.set(j, temp);
        }
    }
}
```

6.3.2 ComparableCircle 类

创建一个名为 ComparableCircle 的类，该类继承自 Circle 类并实现了 Comparable 接口。然后，你需要实现 compareTo 方法，以便根据圆的面积来比较两个 ComparableCircle 对象的大小。最后，你需要编写一个测试类来验证 ComparableCircle 类的功能，通过比较两个 ComparableCircle 对象来找出较大的一个。
比较简单的示例如下。

```
// 可比较的圆形类 ComparableCircle
public class ComparableCircle extends Circle implements
Comparable<ComparableCircle> {
    public ComparableCircle() {
        super();
    }

    public ComparableCircle(double radius) {
        super(radius);
    }

    @Override
    public int compareTo(ComparableCircle other) {
        double thisArea = this.getArea();
        double otherArea = other.getArea();
        return Double.compare(thisArea, otherArea);
    }
}

// 测试类 TestComparableCircle
public class TestComparableCircle {
    public static void main(String[] args) {
        ComparableCircle circle1 = new ComparableCircle(5);
        ComparableCircle circle2 = new ComparableCircle(3);

        if (circle1.compareTo(circle2) > 0) {
            System.out.println("Circle1 is larger");
        } else if (circle1.compareTo(circle2) < 0) {
            System.out.println("Circle2 is larger");
        } else {
            System.out.println("Both circles are equal");
        }
    }
}
```

当然由于圆的特性，所以其实我们可以使用半径去比较。
相关的代码如下。

```
// 可比较的圆形类 ComparableCircle
public class ComparableCircle extends Circle implements
Comparable<ComparableCircle> {
    public ComparableCircle() {
        super();
    }

    public ComparableCircle(double radius) {
        super(radius);
    }

    @Override
    public int compareTo(ComparableCircle other) {
        return Double.compare(this.getRadius(), other.getRadius());
    }
}

// 测试类 TestComparableCircle
public class TestComparableCircle {
    public static void main(String[] args) {
        ComparableCircle circle1 = new ComparableCircle(5);
        ComparableCircle circle2 = new ComparableCircle(3);

        if (circle1.compareTo(circle2) > 0) {
            System.out.println("Circle1 is larger");
        } else if (circle1.compareTo(circle2) < 0) {
            System.out.println("Circle2 is larger");
        } else {
            System.out.println("Both circles are equal");
        }
    }
}
```

当然我们还可以使用前面的 max 方法去比较这里的面积。
示例如下。

```
import java.util.Date;

// 基类 GeometricObject
public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    protected GeometricObject() {
        this.dateCreated = new Date();
    }

    protected GeometricObject(String color, boolean filled) {
        this.dateCreated = new 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();
}

// 圆形类 Circle
public class Circle extends GeometricObject {
    private double radius;

    public Circle() {
        super();
    }

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

    public double getRadius() {
        return radius;
    }

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

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

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

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

// 可比较的圆形类 ComparableCircle
public class ComparableCircle extends Circle implements
Comparable<ComparableCircle> {
    public ComparableCircle() {
        super();
    }

    public ComparableCircle(double radius) {
        super(radius);
    }

    @Override
    public int compareTo(ComparableCircle other) {
        return Double.compare(this.getRadius(), other.getRadius());
    }
}

// 比较类 Max
public class Max {
    public static Comparable max(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}

// 测试类 TestComparableCircle
public class TestComparableCircle {
    public static void main(String[] args) {
        ComparableCircle circle1 = new ComparableCircle(5);
        ComparableCircle circle2 = new ComparableCircle(3);

        ComparableCircle maxCircle = (ComparableCircle) Max.max(circle1,
circle2);

        System.out.println("The larger circle has radius: " +
maxCircle.getRadius());
    }
```

```
}
```

书本示例如下。

```
import java.util.Comparator;

public class GeometricObjectComparator implements Comparator<GeometricObject>,
java.io.Serializable {
    public int compare(GeometricObject o1, GeometricObject o2) {
       double area1 = o1.getArea();
       double area2 = o2.getArea();
       if (area1 < area2)
           return -1;
       else if (area1 == area2)
           return 0;
       else
           return 1;
   }
}

import java.util.Comparator;

public class TestComparator {
    public static void main(String[] args) {
       GeometricObject g1 = new Rectangle(5, 5);
       GeometricObject g2 = new Circle(5);

       GeometricObject g = max(g1, g2, new GeometricObjectComparator());
       System.out.println("The area of the larger object is " + g.getArea());
   }

   public static GeometricObject max(GeometricObject g1, GeometricObject g2,
Comparator<GeometricObject> c) {
       if (c.compare(g1, g2) > 0)
           return g1;
       else
           return g2;
   }
}
```

6.3.3 MyStack 类的深拷贝

重写 MyStack 类以实现深拷贝。

```
public class MyStack {
    private java.util.ArrayList list = new java.util.ArrayList();

    public void push(Object o) {
        list.add(o);
    }

    public Object pop() {
        Object o = list.get(getSize() - 1);
        list.remove(getSize() - 1);
```

```
        return o;
    }

    public Object peek() {
        return list.get(getSize() - 1);
    }

    public int search(Object o) {
        return list.lastIndexOf(o);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    public String toString() {
        return "stack: " + list.toString();
    }
}
```

示例代码如下。

```
import java.util.ArrayList;

public class MyStack implements Cloneable {
    private ArrayList<Object> list = new ArrayList<>();

    public void push(Object o) {
        list.add(o);
    }

    public Object pop() {
        if (isEmpty()) {
            return null;
        }
        Object o = list.remove(getSize() - 1);
        return o;
    }

    public Object peek() {
        if (isEmpty()) {
            return null;
        }
        return list.get(getSize() - 1);
    }

    public int search(Object o) {
        return list.lastIndexOf(o);
    }
```

```
    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    public String toString() {
        return "stack: " + list.toString();
    }

    @Override
    public MyStack clone() {
        try {
            MyStack cloned = (MyStack) super.clone();
            cloned.list = new ArrayList<>(list);
            return cloned;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError(); // Can't happen
        }
    }
}
```

我们这里重写了 MyStack 类下的 clone() 方法，先调用 super.clone() 方法创建当前对象的一个浅拷贝。这会复制对象的所有字段，但对于对象引用类型的字段，只复制引用，不复制引用的对象。再通过调用 new ArrayList<>(list) 来创建了一个新的 ArrayList 对象，并将原 list 中的所有元素添加到这个新对象中。

6.3.4 判断代码对错

判断下列代码是否会成功编译？如果不能，原因是什么？

6.3.4.1 Animal 类

```
public interface Animal {
    String name; // Data field representing the name of the animal
    void makeSound(); // Abstract method to make the animal sound
}
```

这段代码不会成功编译。原因是在Java接口中，你不能直接声明实例字段。

6.3.4.2 MyInterface2 接口

```
public interface MyInterface2 {
    void method1(); // Abstract method without implementation
    void method2(); // Abstract method without implementation
    void method3() {
        // Concrete method with implementation
        System.out.println("Method 3 implementation");
    }
}
```

这段代码不会成功编译。原因是在Java接口中的方法默认是抽象的，不能有方法体。

6.3.4.3 MyClass 类

```
public abstract class MyClass {
    public MyClass() {
        System.out.println("Abstract class constructor");
    }

    public static void main(String[] args) {
        MyClass obj = new MyClass(); // Error here
    }
}
```

这段代码不会成功编译。原因是在Java中不能实例化一个抽象类。