9. 抽象类和接口

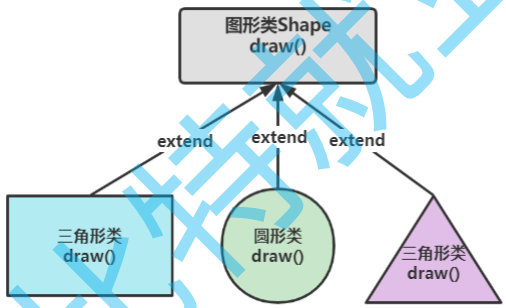
【本节目标】

1. **抽象类**
2. **接口**
3. **Object类**

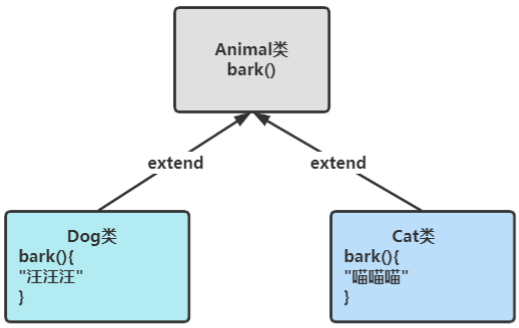
1. 抽象类

1.1 抽象类概念

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，**如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。** 比如：



说明：
1. 矩形、三角形、圆形都是图形，因此和Shape类的惯性应该是继承关系
2. 虽然图形图Shape中也存在draw的方法，但由于Shape类并不是具体的图形，因此其内部的draw方法实际是没有办法实现的
3. 由于Shape类没有办法描述一个具体的图形，导致其draw()方法无法具体实现，因此可以将Shape类设计为“抽象类”



说明：
1. Animal是动物类，每个动物都有叫的方法，但由于Animal不是一个具体的动物，因此其内部bark()方法无法具体实现
2. Dog是狗类，首先狗是动物，因此与Animal是继承关系，其次狗是一种具体的动物，狗叫：汪汪汪，其bark()可以实现
3. Cat是猫类，首先猫是动物，因此与Animal是继承关系，其次猫是一种具体的动物，猫叫：喵喵喵，其bark()可以实现
4. 因此：Animal可以设计为"抽象类"

在打印图形例子中, 我们发现, 父类 Shape 中的 draw 方法好像并没有什么实际工作, 主要的绘制图形都是由 Shape 的各种子类的 draw 方法来完成的. 像这种没有实际工作的方法, 我们可以把它设计成一个 **抽象方法(abstract method)**, 包含抽象方法的类我们称为 **抽象类(abstract class)**.

1.2 抽象类语法

在Java中，一个类如果被`abstract`修饰称为抽象类，抽象类中被`abstract`修饰的方法称为抽象方法，抽象方法不用给出具体的实现体。

```
// 抽象类：被abstract修饰的类
public abstract class Shape {
    // 抽象方法：被abstract修饰的方法，没有方法体
    abstract public void draw();
    abstract void calcArea();

    // 抽象类也是类，也可以增加普通方法和属性
    public double getArea(){
        return area;
    }

    protected double area;   // 面积
}
```

注意：抽象类也是类，内部可以包含普通方法和属性，甚至构造方法

1.3 抽象类特性

1. 抽象类不能直接实例化对象

```
Shape shape = new Shape();

// 编译出错
Error:(30, 23) java: Shape是抽象的; 无法实例化
```

2. 抽象方法不能是 private 的

```
abstract class Shape {
    abstract private void draw();
}

// 编译出错
Error:(4, 27) java: 非法的修饰符组合: abstract和private
```

注意：抽象方法没有加访问限定符时，默认是public.

3. 抽象方法不能被final和static修饰，因为抽象方法要被子类重写

```
public abstract class Shape {
    abstract final void methodA();
    abstract public static void methodB();
}

// 编译报错:
// Error:(20, 25) java: 非法的修饰符组合: abstract和final
// Error:(21, 33) java: 非法的修饰符组合: abstract和static
```

4. 抽象类必须被继承，并且继承后子类要重写父类中的抽象方法，否则子类也是抽象类，必须要使用`abstract`修饰

```
// 矩形类
public class Rect extends Shape {
    private double length;
    private double width;

    Rect(double length, double width){
        this.length = length;
        this.width = width;
    }

    public void draw(){
        System.out.println("矩形: length= "+length+" width= " + width);
    }

    public void calcArea(){
        area = length * width;
    }
}

// 圆类:
public class Circle extends Shape{
    private double r;
    final private static double PI = 3.14;
    public Circle(double r){
        this.r = r;
    }

    public void draw(){
        System.out.println("圆：  r = "+r);
    }

    public void calcArea(){
        area = PI * r * r;
    }
}

// 三角形类:
public abstract class Triangle extends Shape {
    private double a;
    private double b;
```

```
    private double c;

    @Override
    public void draw() {
        System.out.println("三角形：a = "+a + " b = "+b+" c = "+c);
    }

    // 三角形：直角三角形、等腰三角形等，还可以继续细化
    //@Override
    //double calcArea();  // 编译失败：要么实现该抽象方法，要么将三角形设计为抽象类
}
```

5. 抽象类中不一定包含抽象方法，但是有抽象方法的类一定是抽象类
6. 抽象类中可以有构造方法，供子类创建对象时，初始化父类的成员变量

1.4 抽象类的作用

抽象类本身不能被实例化, 要想使用, 只能创建该抽象类的子类. 然后让子类重写抽象类中的抽象方法.

> 有些同学可能会说了, 普通的类也可以被继承呀, 普通的方法也可以被重写呀, 为啥非得用抽象类和抽象方法呢?

确实如此. 但是使用抽象类相当于多了一重编译器的校验.

使用抽象类的场景就如上面的代码, 实际工作不应该由父类完成, 而应由子类完成. 那么此时如果不小心误用成父类了, 使用普通类编译器是不会报错的. 但是父类是抽象类就会在实例化的时候提示错误, 让我们尽早发现问题.

> 很多语法存在的意义都是为了 "预防出错", 例如我们曾经用过的 final 也是类似. 创建的变量用户不去修改, 不就相当于常量嘛? 但是加上 final 能够在不小心误修改的时候, 让编译器及时提醒我们.
>
> 充分利用编译器的校验, 在实际开发中是非常有意义的.

2. 接口

2.1 接口的概念

在现实生活中，接口的例子比比皆是，比如：笔记本上的USB口，电源插座等。

电脑的USB口上，可以插：U盘、鼠标、键盘...所有符合USB协议的设备

电源插座插孔上，可以插：电脑、电视机、电饭煲...所有符合规范的设备

通过上述例子可以看出：**接口就是公共的行为规范标准，大家在实现时，只要符合规范标准，就可以通用。**

在Java中，接口可以看成是：多个类的公共规范，是一种引用数据类型。

2.2 语法规则

接口的定义格式与定义类的格式基本相同，将class关键字换成`interface`关键字，就定义了一个接口。

```
public interface 接口名称{
    // 抽象方法
    public abstract void method1();  // public abstract 是固定搭配，可以不写
    public void method2();
    abstract void method3();
    void method4();

    // 注意：在接口中上述写法都是抽象方法，跟推荐方式4，代码更简洁
}
```

> 提示:
>
> 1. 创建接口时, 接口的命名一般以大写字母 `I` 开头.
> 2. 接口的命名一般使用 "形容词" 词性的单词.
> 3. 阿里编码规范中约定, 接口中的方法和属性不要加任何修饰符号, 保持代码的简洁性.

2.3 接口使用

接口不能直接使用，必须要有一个"实现类"来"实现"该接口，实现接口中的所有抽象方法。

```
public  class 类名称 implements 接口名称{
    // ...
}
```

注意：子类和父类之间是extends 继承关系，类与接口之间是 implements 实现关系。

```
请实现笔记本电脑使用USB鼠标、USB键盘的例子
1. USB接口：包含打开设备、关闭设备功能
2. 笔记本类：包含开机功能、关机功能、使用USB设备功能
3. 鼠标类：实现USB接口，并具备点击功能
4. 键盘类：实现USB接口，并具备输入功能
```

```
// USB接口
public interface USB {
    void openDevice();
    void closeDevice();
}

// 鼠标类，实现USB接口
public class Mouse implements USB {
    @Override
    public void openDevice() {
```

```
        System.out.println("打开鼠标");
    }

    @Override
    public void closeDevice() {
        System.out.println("关闭鼠标");
    }

    public void click(){
        System.out.println("鼠标点击");
    }
}

// 键盘类，实现USB接口
public class KeyBoard implements USB {
    @Override
    public void openDevice() {
        System.out.println("打开键盘");
    }

    @Override
    public void closeDevice() {
        System.out.println("关闭键盘");
    }

    public void inPut(){
        System.out.println("键盘输入");
    }
}

// 笔记本类：使用USB设备
public class Computer {
    public void powerOn(){
        System.out.println("打开笔记本电脑");
    }

    public void powerOff(){
        System.out.println("关闭笔记本电脑");
    }

    public void useDevice(USB usb){
        usb.openDevice();
        if(usb instanceof Mouse){
            Mouse mouse = (Mouse)usb;
            mouse.click();
        }else if(usb instanceof KeyBoard){
            KeyBoard keyBoard = (KeyBoard)usb;
            keyBoard.inPut();
        }
        usb.closeDevice();
    }
}
```

```
// 测试类:
public class TestUSB {
    public static void main(String[] args) {
        Computer computer = new Computer();
        computer.powerOn();

        // 使用鼠标设备
        computer.useDevice(new Mouse());

        // 使用键盘设备
        computer.useDevice(new KeyBoard());

        computer.powerOff();
    }
}
```

2.4 接口特性

1. 接口类型是一种引用类型，但是不能直接new接口的对象

```
public class TestUSB {
    public static void main(String[] args) {
        USB usb = new USB();
    }
}

// Error:(10, 19) java: day20210915.USB是抽象的; 无法实例化
```

2. 接口中每一个方法都是public的抽象方法, 即接口中的方法会被隐式的指定为 **public abstract**（只能是public abstract，其他修饰符都会报错）

```
public interface USB {
    // Error:(4, 18) java: 此处不允许使用修饰符private
    private void openDevice();
    void closeDevice();
}
```

3. 接口中的方法是不能在接口中实现的，只能由实现接口的类来实现

```
public interface USB {
    void openDevice();

    // 编译失败：因为接口中的方式默认为抽象方法
    // Error:(5, 23) java: 接口抽象方法不能带有主体
    void closeDevice(){
        System.out.println("关闭USB设备");
    }
}
```

4. 重写接口中方法时，不能使用default访问权限修饰

```
public interface USB {
    void openDevice();    // 默认是public的
    void closeDevice();   // 默认是public的
}

public class Mouse implements USB {
    @Override
    void openDevice() {
        System.out.println("打开鼠标");
    }

    // ...
}

// 编译报错，重写USB中openDevice方法时，不能使用默认修饰符
// 正在尝试分配更低的访问权限; 以前为public
```

5. 接口中可以含有变量，但是接口中的变量会被隐式的指定为 **public static final** 变量

```
public interface USB {
    double brand = 3.0;  // 默认被：final public static修饰
    void openDevice();
    void closeDevice();
}

public class TestUSB {
    public static void main(String[] args) {
        System.out.println(USB.brand);  // 可以直接通过接口名访问，说明是静态的

        // 编译报错：Error:(12, 12) java: 无法为最终变量brand分配值
        USB.brand = 2.0;      // 说明brand具有final属性
    }
}
```

6. 接口中不能有静态代码块和构造方法

```
public interface USB {
    // 编译失败
    public USB(){

    }

    {}   // 编译失败

    void openDevice();
    void closeDevice();
}
```

7. 接口虽然不是类，但是接口编译完成后字节码文件的后缀格式也是.class
8. 如果类没有实现接口中的所有的抽象方法，则类必须设置为抽象类
9. jdk8中：接口中还可以包含default方法。

2.5 实现多个接口

在Java中，类和类之间是单继承的，一个类只能有一个父类，即**Java中不支持多继承**，但是**一个类可以实现多个接口**。下面通过类来表示一组动物.

```
class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }
}
```

另外我们再提供一组接口, 分别表示 "会飞的", "会跑的", "会游泳的".

```
interface IFlying {
    void fly();
}

interface IRunning {
    void run();
}

interface ISwimming {
    void swim();
}
```


接下来我们创建几个具体的动物

猫, 是会跑的.

```
class Cat extends Animal implements IRunning {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println(this.name + "正在用四条腿跑");
    }
}
```

鱼, 是会游的.

```
class Fish extends Animal implements ISwimming {
    public Fish(String name) {
        super(name);
    }

    @Override
    public void swim() {
        System.out.println(this.name + "正在用尾巴游泳");
    }
}
```

青蛙, 既能跑, 又能游(两栖动物)

```
class Frog extends Animal implements IRunning, ISwimming {
    public Frog(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println(this.name + "正在往前跳");
    }

    @Override
    public void swim() {
        System.out.println(this.name + "正在蹬腿游泳");
    }
}
```

注意：一个类实现多个接口时，每个接口中的抽象方法都要实现，否则类必须设置为抽象类。

> 提示, IDEA 中使用 ctrl + i 快速实现接口

还有一种神奇的动物, 水陆空三栖, 叫做 "鸭子"

```
class Duck extends Animal implements IRunning, ISwimming, IFlying {
    public Duck(String name) {
        super(name);
    }

    @Override
    public void fly() {
        System.out.println(this.name + "正在用翅膀飞");

    }

    @Override
    public void run() {
        System.out.println(this.name + "正在用两条腿跑");
    }
```

```
    @Override
    public void swim() {
        System.out.println(this.name + "正在漂在水上");
    }
}
```

上面的代码展示了 Java 面向对象编程中最常见的用法: 一个类继承一个父类, 同时实现多种接口.

继承表达的含义是 `is - a` 语义, 而接口表达的含义是 `具有 xxx 特性` .

> 猫是一种动物, 具有会跑的特性.
>
> 青蛙也是一种动物, 既能跑, 也能游泳
>
> 鸭子也是一种动物, 既能跑, 也能游, 还能飞

这样设计有什么好处呢? 时刻牢记多态的好处, 让程序猿**忘记类型**. 有了接口之后, 类的使用者就不必关注具体类型, 而只关注某个类是否具备某种能力.

例如, 现在实现一个方法, 叫 "散步"

```
public static void walk(IRunning running) {
    System.out.println("我带着伙伴去散步");
    running.run();
}
```

在这个 walk 方法内部, 我们并不关注到底是哪种动物, 只要参数是会跑的, 就行

```
Cat cat = new Cat("小猫");
walk(cat);

Frog frog = new Frog("小青蛙");
walk(frog);

// 执行结果
我带着伙伴去散步
小猫正在用四条腿跑
我带着伙伴去散步
小青蛙正在往前跳
```

甚至参数可以不是 "动物", 只要会跑!

```
class Robot implements IRunning {
    private String name;
    public Robot(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println(this.name + "正在用轮子跑");

    }
```

```
}

Robot robot = new Robot("机器人");
walk(robot);

// 执行结果
机器人正在用轮子跑
```

2.6 接口间的继承

在Java中，类和类之间是单继承的，一个类可以实现多个接口，接口与接口之间可以多继承。即：用接口可以达到多继承的目的。

接口可以继承一个接口, 达到复用的效果. 使用 extends 关键字.

```
interface IRunning {
    void run();
}

interface ISwimming {
    void swim();
}

// 两栖的动物, 既能跑, 也能游
interface IAmphibious extends IRunning, ISwimming {

}

class Frog implements IAmphibious {
    ...
}
```


通过接口继承创建一个新的接口 IAmphibious 表示 "两栖的". 此时实现接口创建的 Frog 类, 就继续要实现 run 方法, 也需要实现 swim 方法.

> 接口间的继承相当于把多个接口合并在一起.

2.7 接口使用实例

给对象数组排序

```
class Student {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }


    @Override
```

```
    public String toString() {
        return "[" + this.name + ":" + this.score + "]";
    }
}
```

再给定一个学生对象数组, 对这个对象数组中的元素进行排序(按分数降序).

```
Student[] students = new Student[] {
    new Student("张三", 95),
    new Student("李四", 96),
    new Student("王五", 97),
    new Student("赵六", 92),
};
```

按照我们之前的理解, 数组我们有一个现成的 sort 方法, 能否直接使用这个方法呢?

```
Arrays.sort(students);
System.out.println(Arrays.toString(students));

// 运行出错, 抛出异常.
Exception in thread "main" java.lang.ClassCastException: Student cannot be cast to java.lang.Comparable
```

仔细思考, 不难发现, 和普通的整数不一样, 两个整数是可以直接比较的, 大小关系明确. 而两个学生对象的大小关系怎么确定? 需要我们额外指定.

让我们的 Student 类实现 Comparable 接口, 并实现其中的 compareTo 方法

```
class Student implements Comparable {
    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public String toString() {
        return "[" + this.name + ":" + this.score + "]";
    }

    @Override
    public int compareTo(Object o) {
        Student s = (Student)o;
        if (this.score > s.score) {
            return -1;
        } else if (this.score < s.score) {
            return 1;
        } else {

            return 0;
```

```
    }
  }
}
```

在 sort 方法中会自动调用 compareTo 方法. compareTo 的参数是 Object , 其实传入的就是 Student 类型的对象.

然后比较当前对象和参数对象的大小关系(按分数来算).

- 如果当前对象应排在参数对象之前, 返回小于 0 的数字;
- 如果当前对象应排在参数对象之后, 返回大于 0 的数字;
- 如果当前对象和参数对象不分先后, 返回 0;

再次执行程序, 结果就符合预期了.

```
// 执行结果
[[王五:97], [李四:96], [张三:95], [赵六:92]]
```

注意事项: 对于 sort 方法来说, 需要传入的数组的每个对象都是 "可比较" 的, 需要具备 compareTo 这样的能力. 通过重写 compareTo 方法的方式, 就可以定义比较规则.

为了进一步加深对接口的理解, 我们可以尝试自己实现一个 sort 方法来完成刚才的排序过程(使用冒泡排序)

```
public static void sort(Comparable[] array) {
  for (int bound = 0; bound < array.length; bound++) {
    for (int cur = array.length - 1; cur > bound; cur--) {
      if (array[cur - 1].compareTo(array[cur]) > 0) {
        // 说明顺序不符合要求, 交换两个变量的位置
        Comparable tmp = array[cur - 1];
        array[cur - 1] = array[cur];
        array[cur] = tmp;
      }
    }
  }
}
```

再次执行代码

```
sort(students);
System.out.println(Arrays.toString(students));

// 执行结果
[[王五:97], [李四:96], [张三:95], [赵六:92]]
```

2.8Clonable 接口和深拷贝

Java 中内置了一些很有用的接口, Clonable 就是其中之一.

Object 类中存在一个 clone 方法, 调用这个方法可以创建一个对象的 "拷贝". 但是要想合法调用 clone 方法, 必须要先实现 Clonable 接口, 否则就会抛出 CloneNotSupportedException 异常.

```
class Animal implements Cloneable {
    private String name;

    @Override
    public Animal clone() {
        Animal o = null;
        try {
            o = (Animal)super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return o;
    }
}

public class Test {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal animal2 = animal.clone();
        System.out.println(animal == animal2);
    }
}

// 输出结果
// false
```

浅拷贝 VS 深拷贝

Cloneable 拷贝出的对象是一份 "浅拷贝"

观察以下代码:

```
class Money {
    public double m = 99.99;
}

class Person implements Cloneable{
    public Money money = new Money();

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class TestDemo3 {
    public static void main(String[] args) throws CloneNotSupportedException {
        Person person1 = new Person();
        Person person2 = (Person) person.clone();
        System.out.println("通过person2修改前的结果");
        System.out.println(person1.money.m);
```

```
        System.out.println(person2.money.m);
        person2.money.m = 13.6;
        System.out.println("通过person2修改后的结果");
        System.out.println(person1.money.m);
        System.out.println(person2.money.m);
    }
}

// 执行结果
通过person2修改前的结果
99.99
99.99
通过person2修改后的结果
13.6
13.6
```

如上代码，我们可以看到，通过clone，我们只是拷贝了Person对象。但是Person对象中的Money对象，并没有拷贝。通过person2这个引用修改了m的值后，person1这个引用访问m的时候，值也发生了改变。这里就是发生了浅拷贝。那么同学们想一下如何实现深拷贝呢?

2.9 抽象类和接口的区别

抽象类和接口都是 Java 中多态的常见使用方式. 都需要重点掌握. 同时又要认清两者的区别(重要!!! 常见面试题).

核心区别: 抽象类中可以包含普通方法和普通字段, 这样的普通方法和字段可以被子类直接使用(不必重写), 而接口中不能包含普通方法, 子类必须重写所有的抽象方法.

如之前写的 Animal 例子. 此处的 Animal 中包含一个 name 这样的属性, 这个属性在任何子类中都是存在的. 因此此处的 Animal 只能作为一个抽象类, 而不应该成为一个接口.

```
class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }
}
```

再次提醒:

抽象类存在的意义是为了让编译器更好的校验, 像 Animal 这样的类我们并不会直接使用, 而是使用它的子类. 万一不小心创建了 Animal 的实例, 编译器会及时提醒我们.

No	区别	抽象类(abstract)	接口(interface)
1	结构组成	普通类+抽象方法	抽象方法+全局常量
2	权限	各种权限	public
3	子类使用	使用extends关键字继承抽象类	使用implements关键字实现接口
4	关系	一个抽象类可以实现若干接口	接口不能继承抽象类，但是接口可以使用extends关键字继承多个父接口
5	子类限制	一个子类只能继承一个抽象类	一个子类可以实现多个接口

3. Object类

Object是Java默认提供的一个类。Java里面除了Object类，所有的类都是存在继承关系的。默认会继承Object父类。即所有类的对象都可以使用Object的引用进行接收。

范例：使用Object接收所有类的对象

```
class Person{}
class Student{}
public class Test {
    public static void main(String[] args) {
        function(new Person());
        function(new Student());
    }
    public static void function(Object obj) {
        System.out.println(obj);
    }
}
//执行结果：
Person@1b6d3586
Student@4554617c
```

所以在开发之中，Object类是参数的最高统一类型。但是Object类也存在有定义好的一些方法。如下：

Modifier and Type	Method and Description
protected Object	clone() 创建并返回此对象的副本。
boolean	equals(Object obj) 指示一些其他对象是否等于此。
protected void	finalize() 当垃圾收集确定不再有对该对象的引用时，垃圾收集器在对象上调用该对象。
类<?>	getClass() 返回此 Object的运行时类。
int	hashCode() 返回对象的哈希码值。
void	notify() 唤醒正在等待对象监视器的单个线程。
void	notifyAll() 唤醒正在等待对象监视器的所有线程。
String	toString() 返回对象的字符串表示形式。
void	wait() 导致当前线程等待，直到另一个线程调用该对象的 notify()方法或 notifyAll()方法。
void	wait(long timeout) 导致当前线程等待，直到另一个线程调用 notify()方法或该对象的 notifyAll()方法，或者指定的时间已过。
void	wait(long timeout, int nanos) 导致当前线程等待，直到另一个线程调用该对象的 notify()方法或 notifyAll()方法，或者某些其他线程中断当前线程，或一定量的实时时间。

对于整个Object类中的方法需要实现全部掌握。

本小节当中，我们主要来熟悉这几个方法：toString()方法，equals()方法，hashcode()方法

2.2 获取对象信息

如果要打印对象中的内容，可以直接重写Object类中的toString()方法，之前已经讲过了，此处不再累赘。

```
// Object类中的toString()方法实现：
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

2.3 对象比较equals方法

在Java中，==进行比较时：

a.如果==左右两侧是基本类型变量，比较的是变量中值是否相同

b.如果==左右两侧是引用类型变量，比较的是引用变量地址是否相同

c.如果要比较对象中内容，必须重写Object中的equals方法，因为equals方法默认也是按照地址比较的：

```
// Object类中的equals方法
public boolean equals(Object obj) {
    return (this == obj);  // 使用引用中的地址直接来进行比较
}
```

```
class Person{
    private String name ;
    private int age ;
    public Person(String name, int age) {
```

```
        this.age = age ;
        this.name = name ;
    }
}
public class Test {
    public static void main(String[] args) {
        Person p1 = new Person("gaobo", 20) ;
        Person p2 = new Person("gaobo", 20) ;
        int a = 10;
        int b = 10;
        System.out.println(a == b);           // 输出true
        System.out.println(p1 == p2);         // 输出false
        System.out.println(p1.equals(p2));    // 输出false
    }
}
```

Person类重写equals方法后，然后比较：

```
class Person{
    ...
    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false ;
        }
        if(this == obj) {
            return true ;
        }
        // 不是Person类对象
        if (!(obj instanceof Person)) {
            return false ;
        }

        Person person = (Person) obj ; // 向下转型，比较属性值
        return this.name.equals(person.name) && this.age==person.age ;
    }
}
```


结论：比较对象中内容是否相同的时候，一定要重写equals方法。

2.4 hashcode方法

回忆刚刚的toString方法的源码：

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

我们看到了hashCode()这个方法，他帮我算了一个具体的**对象位置**，这里面涉及数据结构，但是我们还没学数据结构，没法讲述，所以我们只能说它是个内存地址。然后调用Integer.toHexString()方法，将这个地址以16进制输出。

hashcode方法源码：

```
public native int hashCode();
```

该方法是一个native方法，底层是由C/C++代码写的。我们看不到。

我们认为两个名字相同，年龄相同的对象，将存储在同一个位置，如果不重写hashcode()方法，我们可以来看示例代码：

```
class Person {
    public String name;
    public int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
public class TestDemo4 {
    public static void main(String[] args) {
        Person per1 = new Person("gaobo", 20) ;
        Person per2 = new Person("gaobo", 20) ;
        System.out.println(per1.hashCode());
        System.out.println(per2.hashCode());
    }
}
//执行结果
460141958
1163157884
```

注意事项：两个对象的hash值不一样。

像重写equals方法一样，我们也可以重写hashcode()方法。此时我们再来看看。

```
class Person {
    public String name;
    public int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }
}
public class TestDemo4 {
    public static void main(String[] args) {
        Person per1 = new Person("gaobo", 20) ;
        Person per2 = new Person("gaobo", 20) ;

        System.out.println(per1.hashCode());
```

```
        System.out.println(per2.hashCode());
    }
}
//执行结果
460141958
460141958
```

注意事项：哈希值一样。

结论：

> 1、hashcode方法用来确定对象在内存中存储的位置是否相同
>
> 2、事实上hashCode() 在散列表中才有用，在其它情况下没用。在散列表中hashCode() 的作用是获取对象的散列码，进而确定该对象在散列表中的位置。