第六天：

教学任务：

第六章 高级特性

目标：

第五章： Advanced Language Features (132-171)

知识点：一. static修饰符
static修饰的方法或者属性,可以使用类.方法或者类.属性访问.
还可以使用引用.方法或者引用.属性 但是不建议使用
引用.的方式访问

. 用static修饰的成员变量表示静态变量，可以直接通过类名来访问；
. 用static修饰的成员方法表示静态方法，可以直接通过类名来访问；
. 用static修饰的程序代码表示静态代码块，当Java虚似机加载类时，就会执行该代码块；

被static所修饰的成员变量和成员方法表明归某个类所有，它不依赖于类的特定实例，被类的所有实例共享。只要这个类被
加载，Java虚拟机就能根据类名在运行时数据区的方法区内定位到它们。

1. static 变量

. Java虚拟机在加载类的过程中为static变量分配内存，实例变量在加载完类后创建对象时分配内存；
static变量存在方法区，实例变量存在堆区；

2. static 方法

```
public class Sample1 {
    public static int add(int x, int y) {
        return x+y;
    }
}

public class Sample2 {
    public void method() {
        int result = Sample1.add(1,2);
        System.out.println("result= " + result);
```

```
    }
}
```

a. static方法 main可以直接访问所属类的引用的实例变量和
类的引用的实例方法，直接访问所属类的静态变量和静态方法；

注：1) 不能使用this关键字，super关键字
2) 静态方法必须被实现。静态方法用来表示某个类所特有的功能，这种功能的实现不依赖于类的具体实例，也不
依赖于它的子类。既然如此，当前类必须为静态方法提供实现。

b. 父类的静态方法不能被子类覆为非静态方法。以下代码编译出错。

```
public class Base {
    public static void method() {}
}

public class Sub extends Base {
    public void method() {}//编译出错
}
```

子类可以定义与父类的静态方法同名的静态方法，以便在子类中隐藏父类的静态方法。子类的静态方法也要满足覆
盖条件。 名字 返回类型，参数

子类隐藏父类的静态方法和子类覆盖父类的实例方法，区别在于：运行时，JVM把静态方法和所属的类绑定，而把实例
方法和所属的实例绑定。

c. 父类的非静态方法不能被子类覆盖为静态方法；

```
class Dog {
        public static void bark() {
                System.out.print("woof ");
        }
    }
    class Basenji extends Dog {
        public static void bark() { }
    }
    public class Bark {
        public static void main(String args[]) {
        Dog woofer = new Dog();
        Dog nipper = new Basenji();
        woofer.bark();
    nipper.bark();
        }
    }
```

static方法只考虑编译类型,不考虑运行时类型.换言之子类继承父类时,父类static方法在调用时,子类即使重写了也不能被执行.如果想要执行子类的static

方法,只能用子类的名称.static方法
一个程序调用了一个静态方法时，要被调用的方法都是在编译时刻被选定的，而这种选定是基于修饰符的编译期类型而做出的
两个方法调用的修饰符分别是变量woofer和nipper，它们都被声明为Dog类型。因为它们具有相同的编译期类型，所以编译器使得它们调用的是相同的方法：Dog.bark。这也就解释了为什么程序打印出woof woof。尽管nipper的运行期类型是Basenji，但是编译器只会考虑其编译器类型

3. static 代码块

类中可以包含静态代码块，它不存于任何方法中。在Java虚拟机中加载类时会执行这些静态代码块。如果类中包含多个静

态代码块，那么Java虚拟机将按照它们在类中出现的顺序依次执行它们，每个静态代码块只会被执行一次。

```
public class Sample {
   static int i = 5;
   static {//第一个静态代码块
        System.out.println("First Static code i="+i++);
   }
   static {//第二个静态代码块
        System.out.println("Second Static code i="+i++);
   }
   public static void main(String[] args) {
        Sample s1 = new Sample();
        Sample s2 = new Sample();
        System.out.println("At last, i= "+i);
   }
}
```

输出：i=7

类的构造方法用于初始化类的实例，而类的静态代码块则可用于初始化类，给类的静态变量赋初始值。

静态代码块与静态方法一样，也不能直接访问类的实例变量和实例方法，而必须通过实例的引用来访问它们。

二. final修改符

父类中用private修饰的方法.不能被子类的方法覆盖，因此private类型的方法默认是final类型的。

1. final类

继承关系的弱点是打破封装，子类能够访问父类的实现细节，而且能以方法覆盖的方式修改实现细节。在以下情况下，

可以考虑把类定义为final类型，使得这个类不能被继承。

. 子类有可能会错误地修改父类的实现细节；
. 出于安全，类的实现细节不允许有任何改动；
. 在创建对象模型时，确信这个类不会再被扩展；

例如JDK中java.lang.String类被定义为final类型；

2. final方法；

某些情况下，出于安全原因，父类不允许子类覆盖某个方法， 此时可以把这个方法声明为final类型。例如在

java.lang.Object类中，getClass()方法为final类型。

3. final变量：

a. final可以修饰静态变量、实例变量、局部变量；
b. final变量都必须显示初始化，否则会导致编译错误；
1) 静态变量，如果没有static代码块只能在定义变量时进行初始化
如果有static代码块可以讲初始化放入到static代码块中.
必须要初始化.只能初始化一次
2) 实例变量，可以在定义变量时，或者在构造方法中进行初始化；
有一个构造方法就要实例化一次

final修饰的成员变量必须显示初始化,如果没有初始化
可以在构造器进行初始化.需要注意,此时构造方法必须对变量初始化
.换言之调用了没有初始化变量构造方法,违反了final原则.

三. abstract修改符

可用来修饰类和成员方法。
. 用abstract修饰的类表示抽象类，抽象类不能实例化
. 用abstract修饰的方法表示抽象方法，抽象方法没有方法体。抽象方法用来描述系统具有什么功能，但不提供具体的实现。
子类继承抽象的父类.需要注意
如果父类中包含抽象方法,子类在继承时又没有重写父类的

抽象方法.此时子类必须也声明称抽象类.否则报错.

abstract不能与final static 以及构造方法并存

语法规则；

1) 抽象类中可以没有抽象方法，但包含了抽象方法的类必须被定义为抽象类；

2) 没有抽象构造方法，也没有抽象静态方法；

abstract和static 不允许同时存在

3) 抽象类中可以有非抽象的构造方法；

4) 抽象类及抽象方法不能被final修饰符修饰。

四. 接口

1. 接口是抽象类的另外一种形式

接口是抽象类的抽象，抽象类可存在有方法体的方法，接口中的方法全部为抽象方法；

2. 接口中的所有方法均是抽象方法， 默认都是public、abstract类型的；

3. 接口中的成员变量默认都是public, static, final类型，必须被显式初始化；

4. 接口中只能包含public, static, final类型成员变量和public、abstract类型的成员方法；

5. 接口中没有构造方法，不能被实例化。

接口和抽象类

1. 相同点：

a. 都不能被实例化；

b. 都能包含抽象方法；

2. 不同点；

a. 抽象类中可以为部分方法提供默认的实现，从而避免子类中重复实现它们，提高代码的可重用性，

而接口中只能包含抽象方法；

b. 一个类只能继承一个直接的父类，这个父类有可能是抽象类；但一个类可以实现多个接口，这是接口的优势所在。

五. 访问控制

通过修饰符的控制来判断哪些值用户可以访问,哪些值用户不能访问.

面向对象的基本思想之一是封装实现细节并且公开方法。Java语言采用访问控制修饰符来控制类及类的方法和变量的访问

权限，从而只向使用者暴露方法，但隐藏实现细节。访问控制分4种级别。

访问级别　访问控制修饰符　同类　同包　子类(不同包 import)　不同的包
公开级别：　public　y　y　y　y
受保护　protected　y　y　y
默认　default没有访问控制符　y　y
私有　private　y

子类继承父类,如果调用时.使用的是父类的引用指向子类的实力对象,此时体现的java的多态,主要的是用来调用方法.没有体现继承关系.
如果想要体现继承关系,需要直接构建子类的引用指向子类的实例对象,此时体现的java的继承关系.
e.g
Student extends Person
Person p = new Student(此时体现多态,调用方法)
Student s = new Student(此时体现继承,可以调用父类变量)

成员变量、成员方法和构造方法可以处于4个访问级别中的一个；
类的访问控制符有两种：public, default(默认的，什么都不用写)　default就是包内访问控制符来

六. ==和equalas
java中的数据类型，可分为两类：

2.复合数据类型(类)

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

那有的同学就问了，那equals方法还有什么用呢?

虽然每个对象都有自己的内存地址，但是每个对象也有自己的一些特有的特征。
比如学生对象有自己的姓名和年龄，希望根据姓名和年龄的相同来判断学生对象是否相同。

这时使用Object类的equals就不能满足需求了，
就需要通过覆盖equals的方式，建立学生对象比较相同的具体内容
两个对象进行比较,如果使用equals.同样是比较两个对象的地址.
此时如果调用的两个对象分别赋予相同的名字
e.g
Student stu1 = new Student(“tom”)
Student stu2 = new Student(“tom”);
由于equals比较的是对象的地址.返回为false表示两个不相同对象。实际当前名称相同的对象时,希望返回相同的对象。此时equals没办法完成职能需要进行重写

```
    重写后比较对象所对应的值
    public boolean equals(Object obj) {
        Student student = (Student)obj;
    return name.equals(student.name);
}
```

七. 内部类

在一个类的内部定义的类称为内部类。内部类允许把一些逻辑相关的类组织在一起，并且控制内部类代码的可视性。对于初

学者而言，内部类似乎有多条，但是随着内部类的逐步了解，就会发现它有独到的用途。它能够让程序结构变得更优雅。

变量按照作用域可分为：

1) 成员变量: 实例变量、静态变量；
2) 局部变量；

同样，内部类按照作用域可分为；

1) 成员内部类: 实例内部类、静态内部类；
2) 局部内部类；匿名内部类

顶层类只能处于public和默认访问级别，而成员内部类可以处于public, protected, private和默认这4种访问级别；

1. 静态内部类；

是成员内部类的一种，用static修饰。静态内部类具有以下特点：

1) 静态内部类的实例不会自动持有外部类的特定实例的引用，在创建内部类的实例时，不必创建外部类的实例。

```
class A {
    public static class B{
        int v;
                static int v2;
    }
}

class Tester {
    public void test() {
        A.B b = new A.B();
        b.v = 1;
                A.B.v2 = 2;
    }
}
```

2) 静态内部类可以直接访问外部类的静态成员，如果访问外部类的实例成员，就必须通过外部类的实例去访问。

```
class A {
    private int a1;             //实例变量a1
    private static int a2;      //静态变量a2

    public static class B {
        int b1 = a1;        //编译错误，不能直接访问外部类A的实例变量a1
        int b2 = a2;        //合法，可以直接访问外部类A的静态变量a2
        int b3 = new A().a1; //合法，可以通过类A的实例访问变量a1
    }
}
```

3) 在静态内部类中可以定义静态成员和实例成员。

```
class A {
    public static class B {
        int v1;                 //实例变量
        static int v2;          //静态变量

        public static class C {
            static int v3;      //静态内部类
        }
    }
}
```

4) 可以通过完整的类名直接访问静态内部类的静态成员。

```
class A {
    public static class B {
        int v1;                 //实例变量
        static int v2;          //静态变量

        public static class C {
            static int v3;      //静态内部类
            int v4;
        }
    }
}

public class Tester {
    public void test() {
        A.B b = new A.B();
        A.B.C c = new A.B.C();
        b.v1 = 1;
        b.v2 = 1;
```

```
        A.B.v1 = 1;          //编译错误
        A.B.v2 = 1;          //合法
        A.B.C.v3 = 1;        //合法
                c.v4 = 1;
        }
}
```

2. 实例内部类；

成员内部类的一种，没有static修饰符。特点：

1) 在创建实例内部类的实例时，外部类的实例必须已经存在。

```
Outer.InnerTool tool = new Outer().new InnerTool();
```

等价于：

```
Outer outer = new Outer();
Outer.InnerTool tool = outer.new InnerTool();
```

以下代码会导致编译错误：

```
Outer.InnerTool tool = new Outer.InnerTool();
```

2) 实例内部类的实例自动持有外部类的实例的引用。在内部类中,可以直接访问外部类的所有成员，包括
成员变量和成员方法。

```
public class A {
    private int a1;
    public int a2;
    static int a3;
    public A(int a1, int a2) {
        this.a1 = a1;
        this.a2 = a2;
    }
    protected int methodA() {
        return a1*a2;
    }

    class B {
        int b1 = a1;            //直接访问private的a1
        int b2 = a2;            //直接访问public的a2
        int b3 = a3;            //直接访问static的a3
        int b4 = new A(3,4).a1;  //访问一个新建的实例A的a1
        int b5 = methodA();      //访问methodA()方法
    }
```

```
    public static void main(String args[]) {
        A.B b = new A(1,2).new B();
        System.out.println("b.b1="+b.b1);   //打印b.b1=1;
        System.out.println("b.b2="+b.b2);   //打印b.b2=2;
        System.out.println("b.b3="+b.b3);   //打印b.b3=0;
        System.out.println("b.b4="+b.b4);   //打印b.b4=3;
        System.out.println("b.b5="+b.b5);   //打印b.b5=2;
    }
}
```

3) 外部类实例与内部类实例之间是一对多的关系，一个内部类实例只会引用一个外部类实例，而一个外部类实例

对应零个或多个内部类实例。在外部类中不能直接访问内部类的成员，必须通过内部类的实例去访问。

```
class A {
    class B {
        private int b1 = 1;
        public int b2 = 2;
        class C{}
    }

    public void test() {
        int v1 = b1;                    //invalid
        int v2 = b2;                    //invalid
        B.C c1 = new C();               //invalid

        B b = new B();
        int v3 = b.b1;                  //valid
        int v4 = b.b2;                  //valid
        B.C c2 = b.new C();             //valid
        B.C c3 = new B().new C();       //valid
    }
}
```

4) 实例内部类中不能定义静态成员，而只能定义实例成员。

5) 如果实例内部类B与外部类A包含同名的成员，那么在类B中， this.v表示类B的成员， A.this.v表示类A的成员。

3. 局部内部类；

在一个方法中定义的内部类，它的可见范围是当前方法。和局部变量一样，局部内部类不能用访问控制修饰符

(public, private和protected)及static修饰符来修饰。特点：

1) 局部内部类只能在当前方法中使用。

```
class A {
```

```
        B b = new B();              //编译错误；
        public void method() {
            class B{
                int v1;
                int v2;

                class C {
                    int v3;
                }
            }
            B b = new B();      构建B()对象比较位于class B{}的后面,main有顺序//合法
            B.C c = b.new C();          //合法
        }
    }
```

2) 局部内部类和实例内部类一样，不能包含静态成员。

```
class A {
    public void method() {
        class B{
            static int v1;          //编译错误
            int v2;                 //合法

            static class C {        //编译错误
                int v3;
            }
        }
    }
}
```

3) 在局部内部类中定义的内部类也不能被public、protected和private这些访问控制修饰符修饰；

4) 局部内部类和实例内部类一样，可以访问外部类的所有成员，此外，局部内部类还可以访问所在方法中的final类型
的参数和变量。

4. 匿名内部类

匿名内部类也就是没有名字的内部类

正因为没有名字，所以匿名内部类只能使用一次，它通常用来简化代码编写

但使用匿名内部类还有个前提条件：必须继承一个父类或实现一个接口

```
abstract class Person {
        public abstract void eat();
}
class son extends person {
        public void eat(){}
}
```

```
public class Demo {
	public static void main(String[] args) {

	Person p = new Person() {
			public void eat() {
				System.out.println("eat something");
			}
		};
		p.eat();
	}
}
```

运行结果：eat something

可以看到，我们直接将抽象类Person中的方法在大括号中实现了

这样便可以省略一个类的书写

并且，匿名内部类还能用于接口上

```
interface Person {
	public void eat();
}

public class Demo {
	public static void main(String[] args) {
	Person p = new Person() {
			public void eat() {
			System.out.println("eat something");
			}
	};
		p.eat();
	}
}
```

运行结果：eat something

由上面的例子可以看出，只要一个类是抽象的或是一个接口，那么其子类中的方法都可以使用匿名内部类来实现

几种内部类的区别：

1. 创建
   a. 声明的位置:
      静态内部类：类的内部，方法的外部，用static关键字修饰；
      实例内部类：类的内部，方法的外部，不用static关键字修饰；
      局部内部类：方法的内部；

   b. 实例化方式:
      静态内部类：new Outer.Inner();　　//在外部类外创建；

new Inner(); //在外部类内内部类外创建
实例内部类：new Outer().new Inner(); //在外部类外创建；
this.new Inner(); //在外部类内内部类外创建
局部内部类：new Inner(); //只能在方法内部创建；

2. 访问
a. 外部类访问内部类：
静态内部类：通过完整的类名直接访问静态内部类的静态成员；
实例内部类：通过内部类的实例去访问内部类的成员；
局部内部类：不能访问；

b. 内部类访问外部类：
静态内部类：直接访问外部类的静态成员；
实例内部类：可以直接访问外部类的所有成员；
如果实例内部类B与外部类A包含同名的成员，那么在类B中， this.v表示类B的成员，
A.this.v表示类A的成员。
局部内部类：可以直接访问外部类的所有成员, 访问所在方法中的final类型的参数和变量；

七. 包装类

AutoBoxing	
primitive type	wrapper type
int	Integer
byte	Byte
long	Long
boolean	Boolean
short	Short
char	Character
double	Double
float	Float

作用： 1) 有些场合必须要引用类型；例如集合中只能存储引用类型；
2) 实现基本类型间以及与字符串间转换；
int -> String : valueOf(int i)
Integer -> String : toString()
String -> int :Integer.parseInt(String s)
Integer -> int : intValue();
int -> Integer: valueOf(int i)
String -> Integer: valueOf(String s)

增强的for循环

for循环的格式:

int[] iarray = new int[]{1,2,3,4}

for(遍历的每一项对应的类型(Object) 每一项的别名(随便起)obj:遍历的引用
iarray) {

Integer i = (Integer)obj;

System.out.println(i)

}

for(int i:iarray) {

System.out.println(i);

}

增强的for循环不需要构建变量,相对原始for简易.

但是需要注意2点

1.最好构建成Object类型 进行强制类型转换

2.因为没有构建变量,所以不能获取其中的某一个值

可变参数

可变长参数使得我们可以声明一个可接收可变数目参数的方法

```
public  int  sum(int a,int b) {
        return a+b;
}
public  int  sum(int a,int b,int c) {
        return a+b+c;
}
public int sum(int[] a) {
        //对数组进行遍历相加
        //必须先声明一个数组
}
int[] a = new int[]{1,2,3,4};
sum(a);
----------------------------------------------
public int sum(int... param) {
        //可变参数在处理的时候按照数组的方式进行处理
        //不需要声明数组
        //直接引用
}
sum(1,2);
sum(2,3,4);
sum();null
```

在使用可变参数后，在调用方法时可以依据类型传入一个或多个该类型的参数或者传入一个该类型的数组参数。

使用条件：

在一个方法中最多只能定义一个可变参数，并且必须位于参数列表的最后。

九. 集合

由数组的缺点引出集合: 数组长度不可变 只能一种类型

数组的长度是固定的，在许多应用场合，一组数据的数目是不固定的，比如一个单位的员工数目是变化的，有老的员工

跳槽，也有新的员工进来。比如一个单位的客户是变化的，有老的客户流失，也有新的客户签单。

为了使程序能方便地存储和操纵数目不固定的一组数据，JDK类库提供了Java集合，所有Java集合类都位于java.util包中。

与Java数组不同，Java集合中不能存放基本类型数据，而只能存放对象的引用。出于表达上的便利，下面把“集合中的对象

的引用”简称为“集合中的对象“。

Java中集合主要分为三种类型：

. Set : 无序，并且没有重复对象。
. List: 有序(放入的先后的次序), 可重复。
. Map : 集合中的每一个元素包含一对键对象和值对象，
集合中没有重复的键对象(相同
键对应的值，后一个覆盖前一个)，值对象可以重复。

1. Collection和Iterator接口

在Collection接口中声明了适用于Set和List的通用方法：

boolean add(Object o) ：向集合中加入一个对象的引用；
void clear() ：删除集合中的所有对象引用，即不再持有这些对象的引用；
boolean contains(Object o) ：判断在集合中是否持有特定对象的引用；
boolean isEmpty() ：判断集合是否为空；
Iterator iterator() ：返回一个Iterator对象，可用它来遍历集合中的元素；
boolean remove(Object o) ：从集合中删除一个对象的引用；整数根据索引删除,其他是内容
int size() ：返回集合中元素的数目；
Object[] toAttray() ：返回一个数组，该数组包含集合中的所有元素；即使使用范型,也只能转换是Object类型数组

Iterator接口隐藏底层集合的数据结构，向客户程序提供了遍历各种类型的集合的统一方法。Iterator接口中声明方法：

hasNext() ：判断集合中的元素是否遍历完毕，如没有，就返回true;
next() ：返回下一个元素；返回object类型

通过下面程序实践上面的方法：

```
import java.util.*;
public class Visitor {
    public static void print(Collection c) {
        Iterator it = c.iterator();
        while(it.hasNext()) {
            Object element = it.next();
            System.out.println(element);
        }
    }

    public static void main(String args[]) {
        Set set = new HashSet();
        set.add("Tom");
        set.add("Mary");
        set.add("Jack");
        print(set);

        List list = new ArrayList();
        list.add("Linda");
        list.add("Mary");
        list.add("Rose");
        print(list);

        Map map = new HashMap();
        map.put("M","男");
        map.put("F","女");
        print(map.entrySet());
    }
}
```

2. Set

最简单的一种集合，集合中的对象无序、不能重复。主要实现类包括：

. HashSet　　：按照哈希算法来存取集合中的对象，存取速度比较快；
. TreeSet　　：实现SortedSet接口，具有排序功能；

一般用法：

Set集合中存放的是对象的引用，并且没有重复对象。

```
Set set = new HashSet();
String s1 = new String("hello");
String s2 = s1;
String s3 = new String("world");
set.add(s1);
set.add(s2);
set.add(s3);
System.out.println(set.size());
```

当一个新的对象加入到Set集合中时，Set的add方法是如何判断这个对象是否已经存在于集合中的呢？它遍历既存对象，通过

equals方法比较新对象和既存对象是否有相等的。

```
boolean isExist = false;
Iterator it = set.iterator();
while(it.hasNext()) {
    String oldStr = it.next();
    if(newStr.equals(oldStr)) {
        isExists = true;
        break;
    }
}
```

举例：Set set = new HashSet();

```
String s1 = new String("hello");
String s2 = new String("hello");
set.add(s1);
set.add(s2);
System.out.println(set.size());                    //集合中对象数目为1;
```

1) HashSet

按照哈希算法来存取集合中的对象，存取速度比较快。当向集合中加入一个对象时，HashSet会调用对象的hashCode()

方法来获得哈希码，然后根据这个哈希码进一步计算出对象在集合中的存放位置。

哈希表与哈希方法

哈希方法在“键- 值对”的存储位置与它的键之间建立一个确定的对应函数关系 hash() ，使得每一个键与结构中的一个唯一的存储位置相对应：

存储位置=hash(键)

举一个例子，有一组“键值对”：<5, ” tom ” >、 <8, ” Jane ” >、 <12, ” Bit ” >、 <17, ” Lily ” >、 <20, ” sunny ” >，我们按照如下哈希函数对键进行计算 :

hash(x)=x%17+3 ，得出如下结果： hash(5)=8 、 hash(8)=11 、 hash(12)=15 、 hash(17)=3 、 hash(20)=6 。我们把 <5, ” tom ” >、 <8, ” Jane ” >、 <12, ” Bit ” >、 <17, ” Lily ” >、 <20, ” sunny ” >分别放到地址为 8 、 11 、 15 、 3 、 6 的位置上。当要检索 17 对应的值的时候，只要首先计算 17 的哈希值为 3 ，然后到地址为 3 的地方去取数据就可以找到 17 对应的数据是“ Lily ”了，可见检索速度是非常快的。

冲突与冲突的解决

通常键的取值范围比哈希表地址集合大很多，因此有可能经过同一哈希函数的计算，把不同的键映射到了同一个地址上面，这就叫冲突。比如，有一组“键- 值对”，其键分别为 12361 、 7251 、 3309 、 30976 ，采用的哈希函数是：

```
public static int hash(int key)
{
```

1. 王政泽
2020年8月4日 下午2:08:36
原来已经存放好的值 重新计算哈希值 再放吗

return key%73+13420;
}
则将会得到hash(12361)=hash(7251)=hash(3309)=hash(30976)=13444 ，即不同的键通过哈希函数对应到了同一个地址，我们称这种哈希计算结果相同的不同键为同义词。此时再换用另一种哈希函数计算.换言之,当存在多个键通过哈希计算结果相同时,会再选用另一个哈希计算直到不存在重复地址为止.

1

hashCode方法默认返回对象的地址,String,Integer等封装类型对它进行了重写返回一个整数
该整数的取值来自于当前字符串的每个字母的编码值.公示如下
public int hashCode(){
return “abcde”.hashCode();
}
在Object类中定义了hashCode()方法和equals()方法，Object类的equals()方法按照内存地址比较对象是否相等，因
此如果object.equals(object2)为true, 则表明object1变量和object2变量实际上引用同一个对象，那么object1和
object2的哈希码也肯定相同。

为了保证HashSet能正常工作， 要求当两对象用equals()方法比较的结果为true时，它们的哈希码也相等。如果用户
定义的Customer类覆盖了Object类的equals()方法，但是没有覆盖Object类的hashCode()方法，就会导致当
customer1.equals(customer2)为true时，而customer1和customer2的哈希码不一定一样，这会使HashSet无法正常工作。

```
public class Customer {
    private String name;
    private int age;

    public Customer(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public boolean equals(Object o) {
        if(this==o) return true;
        if(!(o instanceof Customer)) return false;
        Customer other = (Customer)o;
```

```
                if(this.name.equals(other.getName()) && this.age.equals(other.getAge())
                    return true;
                else
                    return false;

            }
        }
```

以下程序向HashSet中加入两个Customer对象。

```
Set set = new HashSet();
Customer customer1 = new Customer("Tom", 15);
Customer customer2 = new Customer("Tom", 15);
set.add(customer1);
set.add(customer2);
System.out.println(set.size());        //打印出 2
```

出现以上原因在于customer1和customer2的哈希码不一样，因此为两为customer对象计算出不同的位置，于是把它们
放到集中中的不同的地方。

应加入以下hashCode()方法:

```
public int hashCode() {
    int result;
    result = (name==null?0:name.hashCode());
    result = 29*result + age;
    return result;
}
```

2) TreeSet

TreeSet实现了SortedSet接口，能够对集合中的对象进行排序。当TreeSet向集合中加入一个对象时，会把它插入到有
序的对象序列中。那么TreeSet是如何对对象进行排序的呢？TreeSet支持两种排序方式：自然排序和客户化排序。默
认情况下TreeSet采用的是自然排序方式：

a. 自然排序 升序 默认返回负数
当前和传入的比较返回 升序
传入和当前的比较返回 降序
String类型比较大小,使用compareTo(Object obj)方法.

在JDK类库中， 有一部分类实现了Comparable接口，如Integer、Double和String等。Comparable接口有一个
compareTo(Object o)方法，它返回整数类型。对于x.comapreTo(y), 如
x表示第一个值 y表示第二个值

返回0,　　表明　x和y相等
返回值大于0, 表明　x>y　　1
返回值小于0, 表明　x<y　-1

e.g

```
当前类型 Student implements Comparable {
    int age;
    public int compareTo(Object obj) {
        Student stu = (Student)obj;
        return age-stu.age;当前age－传入对象的age　自然排序
                stu.age-age;传入对象的age－当前age 降序排序
    }
}
```

TreeSet调用对象的compareTo()方法比较集合中对象的大小，然后进行升序排序，这种排序方式称为自然排序。

JDK类库中实现了Comparable接口的一些类的排序方式：

Byte, Short, Integer, Long, Double, Float　　:　　按数字大小排序；
Character　　:　　按字符的Unicode值的数字大小排序；
String　　:　　按字符串中字符的Unicode值排序；

使用自然排序， TreeSet中只能加入相同类型对象，且这些对象必须实现了Comparable接口。否则会抛出
ClassCastException异常。

最适合TreeSet排序的是类的内容可以操作,可以修改限定类的内容.

b. 客户化排序

除了自然排序外， TreeSet还支持客户化排序。java.util.Comparator接口提供了具体的排序方法， 它有一个
compare(Object x, Object y)方法，用于比较两个对象的大小， 当compare(x,y)：

返回0,　　表明　x和y相等
返回值大于0, 表明　x>y
返回值小于0, 表明　x<y

如果希望TreeSet按照Customer对象的name属性进行降序排列，可以先创建一个实现Comparator接口的类

CustomerComparator, 参见：

使用compare(Object o1,Object o2)进行比较,

一定要o1.compare(o2).如果想保持当前顺序,返回－1

如果想更改当前的顺序.返回1

e.g

```
compare(Object o1,Object o2) {
    o1.compare(o2)>0 return -1 时 表示降序
    o1.compare(o2)<0  return 1 时 表示降序
    o1.compare(o2)>0 return 1时 表示升序
    o1.compare(o2)<0 return -1时 表示升序
}
import java.util.*;

 public class CustomerComparator implements Comparator {
 public int compare(Object o1, Object o2) {
      Customer c1 = (Customer)o1;
      Customer c2 = (Customer)o2;
                  1   3    5  2  4    5 4 3 2 1
      if(c1.getName().compareTo(c2.getName())>0) return -1;
      if(c1.getName().compareTo(c2.getName())<0) return 1;

                  return 0;
 }

 public static void main(String[] args) {
      Set set = new TreeSet(new CustomerComparator());

      Customer customer1 = new Customer("Tom",15);
      Customer customer3 = new Customer("Jack",16);
      Customer customer2 = new Customer("Mike",26);
      set.add(customer1);
      set.add(customer2);
      set.add(customer3);

      Iterator it = set.iterator();

      while(it.hasNext()) {
          Customer customer = it.next();
          System.out.println(customer.getName() + " " + customer.getAge());
      }
  }
}
```

打印输出：

```
Tom 15
Mike 26
Jack 16
```

3. List

Arrays.asList()将一个数组转化为一个List对象

Collection.toArray():将一个集合转换成一个数组对象 Object[]

主要特征是其元素以线性方式存储，集合中允许存放重复对象。主要实现类包括：

. ArrayList: 代表长度可变的数组。允许对元素进行快速的随机访问，但是向ArrayList中插入与删除元素的速度较慢；

遍历方式：

a. list.get(i);　//通过索引检索对象；

b. Iterator it = list.iterator();

it.next();

LinkedList和ArrayList的区别

LinkedList和ArrayList的差别主要来自于Array和LinkedList数据结构的不同。如果你很熟悉Array和LinkedList，你很容易得出下面的结论：

1) 因为Array是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。但是要删除数据却是开销很大的，因为这需要重排数组中的所有数据。

2) 相对于ArrayList，LinkedList插入是更快的。因为LinkedList不像ArrayList一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据重新装入一个新的数组，这是ArrayList最坏的一种情况，

3) 类似于插入数据，删除数据时，LinkedList也优于ArrayList。

4) LinkedList需要更多的内存.

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）

```
public class GenericTest {
    public static void main(String[] args) {
        /*
        List list = new ArrayList();
        list.add("qqyumidi");
        list.add("corn");
        list.add(100);
        */
        List<String> list = new ArrayList<String>();
        list.add("qqyumidi");
        list.add("corn");
        //list.add(100);  // 1  提示编译错误
        for (int i = 0; i < list.size(); i++) {
            String name = list.get(i); // 2
            System.out.println("name:" + name);
        }
    }
}
```

采用泛型写法后，在//1处想加入一个Integer类型的对象时会出现编译错误，通过List<String>，直接限定了list集合中只能含有String类型的元素，

从而在//2处无须进行强制类型转换，因为此时，集合能够记住元素的类型信息，编译器已经能够确认它是String类型了。

结合上面的泛型定义，我们知道在List<String>中，String是类型实参，也就是说，相应的List接口中肯定含有类型形参。且get()方法的返回结果也

直接是此形参类型（也就是对应的传入的类型实参）

ArrayList<E>

ArrayList称为该泛型类的原始类型

E为类型参数

ArrayList<E>称为ArrayList的泛型类型/参数类型

如果ArrayList什么都不写,默认是ArrayList<Object>

4. Map

Map是一种把键对象和值对象进行映射的集合，它的每一个元素都包含一对键对象和值对象。向Map集合中加入元素时，

必须提供一对键对象和值对象，从Map集合中检索元素时，只要给出键对象，就会返回对应的值对象。

```
map.put("2", "Tuesday");
map.put("3", "Wednsday");
map.put("4", "Thursday");

String day = map.get("2");   //day的值为"Tuesday"
```

Map集合中的键对象不允许重复，如以相同的键对象加入多个值对象，第一次加入的值对象将被覆盖。

对于值对象则没有唯一性的要求可以将任意多个键对象映射到同一个值对象上。

```
map.put("1", "Mon");
map.put("1", "Monday");     //"1"此时对应"Monday"
map.put("one", "Monday");   //"one"此时对应"Monday"
```

HashMap

按哈希算法来存取键对象，有很好的存取性能，为了保证HashMap能正常工作，和HashSet一样，要求当两个键对象

通过equals()方法比较为true时，这两个键对象的hashCode()方法返回的哈希码也一样。

反射:

主要是指程序可以访问，检测和修改它本身状态或行为的一种能力，并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义。

反射是java中一种强大的工具，能够使我们很方便的创建灵活的代码，这些代码可以再运行时装配，无需在组件之间进行源代码链接。但是反射使用不当会成本很高

反射机制的作用：

1,反编译：.class-->.java 配置文件xml @注解

2,通过反射机制访问java对象的属性，方法，构造方法等

在这里先看一下sun为我们提供了那些反射机制中的类：

java.lang.Class;
java.lang.reflect.Constructor; java.lang.reflect.Field;
java.lang.reflect.Method;
java.lang.reflect.Modifier;

很多反射中的方法，属性等操作我们可以从这四个类中查询

getModifiers():返回修饰符对应的整数

0:default 1:public 2:private 4:protected 其他数字由他们共同组成.