**Java学习笔记**

JDK版本：JDK1.4, JDK1.5(JDK5.0代号tiger), JDK1.6(JDK6.0代号Mustang), JDK1.7(JDK7.0代号Dolphin), JDK1.8(JDK8.0代号Mustang)

Java中的数据类型：

1. 整型：使用int表示，int a = 1; // 32 bit
2. 字节型：使用byte表示(表示范围为-128 ~ 127 )， byte b = 2; // 8 bit
3. 短整型：使用short表示，short c = 1; // 16 bit
4. 长整型：使用long表示，long d =1; // 64 bit
5. 单精度浮点型：使用float表示，float e = (float) 1.2; or float e = 1.2F;
6. 双精度浮点型：使用double表示，Java中所有的浮点型默认为double类型！！！
7. 字符型行：使用char表示，char a = ‘a’; char b = ‘张’;
8. 布尔类型，使用boolean表示，只有两种值，分别是true和false，boolean a = false;

变量在使用之前必须定义类型，而且必须赋初值。

Java中出现的短路现象

public class logictest

{

public static void main(string args[])

{

int a = 1;

int b = 2;

int c = 3;

int d = 4;

int f = 6;

boolean e = (a>b)&&((f=c) < d);

System.out.println(e);

System.out.println(f);

}

}

**由于在Java中，逻辑与运算，当第一个运算符已经决定结果时，第二个运算则没有必要继续进行，所以最终打印得到的f的结果为6.同样对于逻辑或也有这样的短路情况发生。**

面向对象：面向对象中有两个很重要的概念：类（class）和对象（object）

类：类是一种抽象概念，类中包含了数据，以及对数据的操纵。类包含的内容如下：

1. 数据，数据在类中称作属性（property或者attribute）或者成员变量（member variable）
2. 对数据的操纵，称作方法（method）

对象：对象（Object）又叫做实例（Instance）是一种具体的概念，是类的一种具体表示方式，比如人是一个类，而张三、李四等具体的人就是对象。

方法的定义：方法不可以嵌套。方法只能定义在类中。下述的例子中，return生效时，直接返回方法，不会执行println语句

public void test(int a)

{

if (a < 10)

{

return;

}

System.out.println(“Hello World”);

}

类名的命名约定：首字母大写，如果由多个单词构成，**每个单词的首字母都要大写**，例如，MemberTest

方法的命名约定：首字母小写，如果由多个单词构成，**第一个单词的所有字母小写，从第二个单词开始，每个单词的首字母大写**，例如：addThreeInt

属性命名约定：与方法的预定完全相同，例如ageOfPerson。

定义在类中的变量叫做**成员变量**，定义在方法中的变量叫做**局部变量**。

**引用类型（reference type）：引用类型是用在对象上的。一个对象可以被多个引用所指向，但同一时刻，每个引用只能指向唯一的一个对象，如果一个对象被多个引用所指向，那么无论哪个引用对对象的属性进行修改，都会反映到其它引用当中。具体例子如下所示：**

**public** **class** People

{

**int** age = 20;

**public** **void** change(People people1)

{

people1.age = 30;

}

**public** **static** **void** main(String[] args)

{

// **TODO** 自动生成的方法存根

People people = **new** People();

System.*out*.println(people.age);

people.change(people);

System.*out*.println(people.age);

}

}

**public class People**

**{**

**int age = 20;**

**public void change(People people1)**

**{**

**people1.age = 30;**

**}**

**public static void main(String[] args)**

**{**

**// TODO 自动生成的方法存根**

**People people1 = new People();**

**People people2 = new People();**

**people1.age = 30;**

**System.out.println(people1.age);**

**System.out.println(people2.age);**

**}**

**}**

注意上述两者之间的区别，第一个是将一个引用的地址传递给另外一个应用，而第二个是同时创建了2个应用，所以对其中一个引用更改了成员变量的数值并不影响第二个成员变量的值。（见视频第十四讲）。

注意区分引用传递和值传递对结果的不同影响（见视频第十一讲）。

**构造方法**：构造方法用于完成对象属性的初始化工作。特点：

1. 构造方法的名字必须与类名完全一致，所以如果按照正规的命名方式，构造方法应该是大写字母开通区别于普通的方法。
2. 构造方法没有返回值，连void也不能出现。
3. 如果在定义类时，没有声明构造方法，那么Java编译器会自动为类添加一个没有参数，且方法体为空的构造方法。
4. 不能显示调用类的构造方法，构造方法通常是用过new关键字隐式调用。

new关键字在声明对象时，完成了三件事情。

1. 为对象开辟内存空间
2. 调用类的构造方法
3. 返回对象的生成地址。

**方法的重载：参数个数不同 || 参数类型不同**

**构造方法也是可以重载的，不同的重载方法之间也是可以互相调用的，对于构造方法之间的互相调用，由于构造方法不可以显示的调用，所以需要通过this来实现互相调用，this表示对当前对象的引用。而且组要注意的是this语句在构造方法中必须为第一条语句。（视频第十七讲）**

public class People

{

public People()

{

this(3);

System.out.println("ahshsdhf sa");

}

public People(int i)

{

System.out.println("ahshsdhf sa");

}

public static void main(String[] args)

{

// TODO 自动生成的方法存根

int i;

People people = new People();

}

}

**继承：**Java是单继承的，一个类只能从另一个类继承，构造方法，当某个类是从父类有继承关系式，先会调用父类不带参数的构造方法再调用子类的构造方法（见视频第十七讲），如果父类中不存在不带参数的构造方法，调用时则会报错，可以通过super关键字来实现正常的调用，super表示对父类对象的引用（**和this一样，super必须出现在方法内的第一条语句**），而this表示对当前子类的引用，具体例子如下：

**public** **class** People **extends** Person

{

**public** People()

{

**super**(3);

System.*out*.println("People");

}

**public** **static** **void** main(String[] args)

{

// **TODO** 自动生成的方法存根

**int** i;

People people = **new** People();

}

}

**class** Person

{

**public** Person(**int** i)

{

System.*out*.println("Person");

}

}

关于继承的3点：

1. 父类有的，子类也有。
2. 父类没有的，子类可以增加（通过向下类型转换来实现调用）。
3. 父类有的，子类可以改变，而且子类的属性优先级更高，调用时优先调用子类的值。

继承注意事项：

1. 构造方法不可以被继承
2. 方法和属性可以被继承
3. 子类的构造方法隐式地调用父类的不带参数的构造方法。
4. **当父类没有不带参数的构造方法时，子类需要使用super来显示调用父类的构造方法，super指的是对父类的引用。**
5. Super关键字必须是构造方法中的第一条语句。

**重载与覆写之间的区别：重载发生在一个类的内部的两个或多个方法，覆写是子类覆写父类的方法。可以在子类中使用super.方法名来调用父类的方法，而且super的位置不局限于在第一条语句，区别于构造方法中super的用法，这是因为构造方法中使用super来初始化父类，当时父类还不存在，而在此处，父类已经存在，只是调用了父类的方法，所以此处super不局限于第一条语句。**

**多态：一句话表示，子类就是父类，父类的引用可以指向子类的应用，或者接口类型的引用可以指向实现该接口的类的实现，Flower rose = new Rose();如下的例子所示，为一个典型的多态的使用情况，判断输出结果主要看引用指向的对象是父类还是子类，而不是看引用的类型是父类还是子类。 多态视频需要再看一次（视频第二十讲）**

**即：Parent p = new Child();当使用多态方式调用方法时，首先检查父类中是否有sing()方法，如果没有则编译错误，（调用构造方法时也是一样，需要父类中有不带参数的构造方法）如果有，再去调用子类的sing（）方法。**

**public** **class** Ploy

{

**public** **static** **void** main(String[] args)

{

// **TODO** 自动生成的方法存根

Flower rose = **new** Rose();//多态，子类就是父类，父类不是子类

rose.sing();

}

}

**class** Flower

{

**public** **void** sing()

{

System.*out*.println("Flower is singing");

}

}

**class** Rose **extends** Flower

{

**public** **void** sing()

{

System.*out*.println("Rose is singing");

}

}

Rose rose2 = (Rose) rose;//向下类型转换，将之前的flower类型的引用转换为rose类型的引用。即向下类型转换只能转换为指向的类型

一共有两种强制类型转换：

1. 向上类型转换（upcast）：子类转换为父类，比如讲cat类转换成animal类型。对于向上类型转换，不需要显示指定。（类型转换的过程可以联想到原生数据类型的转换，从byte转换到int就像向上类型转换，不需要显示指定类型转换符，而从int转向byte，就犹如向下类型转换，需要显示指定型转换符）例如：

Animal animal = (Animal)cat; //可以省略强制类型转换(Animal)

实际上也就相当于Animal animal = new Cat();

1. 向下类型转换（downcast）：比如将Animal类型转换成Cat类型，即将父类型转换为子类型，对于向下的类型转换，必须要显示指定类型转换（**而且需要注意的是，只能转换成指向的类型，否则会报错**）。

Animal a = new Cat();

Cat cat = (Cat) a;

**cat.sing(); //结果和a.sing()相同！！！既然结果相同，为啥要转换，原因在于不转换的情况下，调用子类的方法必须也在父类中出现，如果子类中重新定义了一些自己属性的方法，则不转换的话，不可以调用子类中的方法。**

**此处还有一个问题不太明白，就是为什么需要向下类型转换，而不直接定义一个子类的引用，那子类的所有方法都可以直接调用？？？？？？？？？？？？？？**

**抽象类：**抽象类的特点，

1. 抽象类无法实例化，即无法new出来一个新的抽象类对象。

**抽象方法：抽象方法特点：有声明，没有实现，抽象方法必须声明在抽象类中，例如： public abstract void method();//不能省略分号，注意public abstract void method(){}不是抽象方法，有花括号。**

**无论何种情况，只要一个类是抽象类，那么这个类就无法实例化。抽象类中可以不包含抽象方法。**

**在子类继承父类（父类是个抽象类）的情况下，那么该子类必须要实现父类中所定义的所有抽象方法，否则，该子类需要声明成一个抽象类。**

**接口（interface）：接口的定位等同于class，接口中所有的方法都是抽象方法（所以接口以及接口内部方法的abstract关键字都可以省略）。接口与抽象类的区别在于，接口中所有的方法必须是抽象方法，且类可以实现多个接口，多个接口之间使用逗号分隔。**

**接口中的成员变量都是public, final static的。由于默认都是这种属性，因此关键字public, final and static都是可以省略的，不过通常在开发的过程中，接口中通常只是定义方法，并不定义成员变量。**

类可以**实现**某个接口，通过关键字**implements**实现**。**

同样，某个类如果实现了一个接口，那么这个类必须实现接口中的所有方法，除非这个类是抽象类（如果是抽象类可以不实现任何方法也可以实现部分方法）。

static关键字：static可以应用在成员变量、方法和类上。

**static修饰成员变量**：例子如下，无论一个类生成了多少个对象，所有这些对象共同使用唯一一份静态的成员变量，一个对象对该静态成员变量进行修改，其它对象的该静态成员变量的值也会随之发生变化。**由于使用静态关键字修饰之后的类名，不会对不同的对象之间加以区分，即该成员变量可以属于任何的对象，因此可以直接通过类名.成员变量的名字去访问它（而且Java推荐使用这种方式访问），例如 MyStatic.a = 10;**

**还需要注意一个问题在于静态方法是可以被继承的，但是子类是不可以覆写父类的静态方法的，即使覆写了父类的静态方法，也会存在一个父类静态方法隐藏的问题（见视频第二十二讲）。**

**一言以蔽之：静态方法可以被继承，但是不可以被覆写。**

class MyStatic

{

static int *a*;

}

public class StaticTest

{

public static void main(String[] args)

{

// TODO 自动生成的方法存根

MyStatic myStatic = new MyStatic();

MyStatic myStatic2 = new MyStatic();

mystatic.*a* = 10;

System.*out*.println(mystatic2.*a*);

}

}

**static修饰方法**：例子如下，对于静态方法而言，可以使用类名.方法名的方式来访问（Java推荐这么使用）(对于一个类，类中都是用static方法，这些方法提供一些基本的功能，例如查找、排序等等，类外可以直接通过函数名访问，这是static的常见使用场景)。

**class** MyStatic

{

**public** **static** **void** myStatic()

{

System.*out*.println("static mystatic");

}

}

**public** **class** StaticTest

{

**public** **static** **void** main(String[] args)

{

// **TODO** 自动生成的方法存根;

MyStatic.*myStatic*();

}

}

**静态方法不可以访问一个非静态的成员变量，而非静态方法可以访问一切类型的成员变量，静态方法中也不可以出现this关键字。**

**final关键字：可以修成员变量、方法、类。**

**final修饰类：**final类不可以被继承，因此一个类不可以既是abstract又是final类。

**final修饰方法：**final修饰方法时，表示表示方法是一个终态方法不可以被重写。

**final修饰属性：**final修饰属性时，表示该属性不能被更改，**final修饰的成员变量必须赋初值，赋初值的方式有两种，一种是直接赋初值，另外一种是在所有的构造函数中赋初值（见视频第二十四讲）。**

静态代码块先于构造方法执行。静态代码块如下：

static

{

}

static静态代码块：作用是为了完成一些初始化工作，首先执行静态代码块，然后执行构造方法，静态代码块被加载的时候执行，而构造方法时在生成对象时候执行，要想调用某个类来生成对象，首先需要将类加载到Java虚拟机中（JVM），然后由JVM加载这个类来生成。类的静态代码块只会执行一次，构造方法则每次生成对象时，都会执行一次构造方法。如果继承体系中既有构造方法又有静态代码块，那么首先执行顶层的静态代码块，一直执行到最底层的静态代码块，然后再去执行最顶层的构造方法，一直执行到最底层类的构造方法。

静态方法的只能访问静态的成员变量；非静态方法可以访问一切成员变量，静态方法中不可以出现this，因为this是最当前对象的引用，而静态方法不需要对象。

**Design Pattern(设计模式)：单例模式（Singleton）表示一个类只会生成唯一的一个对象。下述为一个Singleton的代码实现。**

**public** **class** Singleton

{

**public** **static** **void** main(String[] args)

{

SingletonTest a = SingletonTest.*getInstance*();

}

}

**class** SingletonTest

{

**private static SingletonTest singleton = new SingletonTest();**

**private** SingletonTest()

{

}

**public** **static** SingletonTest getInstance()

{

**return** *singleton*;

}

}

首先将构造函数设置为private的原因在于防止在类外使用new去生成一个新的类的对象，这样的话就不是单例了，然后需要在类内部声明一个对象，类的外部如果想使用这个类的对象又不能使用new来生成对象，只有通过static方法的方式，直接调用对象，因此类内部又生成了一个static的方法，而且该静态方法只能调用静态成员，因此内部声明的私有成员也必须是静态类型。

**Package：**类似于文件夹的使用，将王城不同功能的类进行分类，放在不同的目录下。

包的命名规则：每个字母都需小写，通常是使用.将多个单词拼接起来。

**Import：**将使用package分离的各个类导入回来，让编译器能够找到所需要的类，import .目录.\*(将目录下的所有文件全部导入，**不会导入子包下面的类，即当前文件夹下文件夹中的文件**，即import aa.bb.\*\*不会导入aa.bb.cc下面的包)。

**访问修饰符:**

1. **public：被public所修饰的属性和方法可以被所有的类访问。**
2. **private：被private所修饰的属性和方法只能在该类的内部使用。**
3. **protected：被protected所修饰的属性和方法可以在类内部、相同包以及该类的子类所访问。**
4. **默认（不加任何访问修饰符）：在类内部可以访问，以及相同包下面的类可以访问。**

**Instanceof关键字：判断每个对象是否是某个类的实例，语法形式：引用名 instanceof 类名（接口名）。返回值会boolean值。**

**Object包**

**Object类中，的equals()方法用来判断调用equals()的引用与传进来的引用是否一致，即这两个引用是否指向的是同一个对象。对于Object类的equals()方法来说，它等价于==**

**String类中的equals()方法来说，它是判断当前字符串与传进来的字符串内容是否一致，因为String中的equals()对Object中的equals()方法进行了重写 。**

**Java String Pool：Java中会维护一个字符串池的概念，（见视频第三十讲）**

String s = “aaa”;//1.查找String中String Pool中是否存在”aaa”,如果不存在则创建一个新的”aaa”,如果存在则不再创建任何对象，直接将String Pool中的这个”aaa”对象地址返回来，赋值给s引用。

String s = new String(“aaa”);//不管字符串池中有没有创建过”aaa”对象，堆（heap）中都会创建一个”aaa”对象，然后将堆中的字符串对象返回。

System.out.println(“100” + 200);// 当字符串出现+号时，不再是数学中的运算符，而是拼接操作。

**包装类（Wrapper Class）：**所有的包装类（8个）都位于java.lang包下。

int a = 10;

Integer integer = new Integer(a);

b = integer.intValue();

System.out.println(b);

**数组（Array）:相同类型**数据的集合。

int[] a = new int [10];// Java中推荐使用这种方法定义数组。

**二维数组：**定义不规则二维数组，实现代码如下：

int[][] a = new int [3][];

a[0] = new int [2];

a[1] = new int [3];

a[2] = new int [1];

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

int[] a = new int[4]{1, 2, 3, 4};//错误

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

对于数组而且，可以使用等号进行赋值，即：

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

int[] b = new int[4];

b = a;

但是需要注意的是上述只是一个浅拷贝，如果改变数组b中的数字，数组a中的数值也会跟着变化，这是因为数据b指向数据a，实现深拷贝可以通过for循环逐一复制，可以用clone或者System.arrayCopy来实现。

**二分查找：二分查找必须有序。**

**Java中常量的命名：**所有的单词字母都是大写，如果有多个单词，那么使用下划线连接即可。例如：

public static final int AGE\_OF\_PERSON = 20;// 常量经常加上static关键字，这是因为static关键字在系统中只保留一个备份，节省系统开销。

**不能将Object[]转换为Integer[]。**

**ArrayList底层采用数组实现，ArrayList中放置的都是Object类型，因此取出来的也是Object类型，无法放置原生数据类型，我们需要使用对生数据类型的包装类才能加入到集合当中， arrayList.add(new Integer(1)),同理取出来到额时候也需要使用强制类型转换才能获得值即:((Integer)arrayList.get(0)).intValue()。**

**ArrayList与LinkedList的比较：**当执行删除或者插入操作时，采用LinkedList比较好，当执行搜索操作时，使用ArrayList比较好，当向ArraryList添加一个对象时，实际上是将该对象放置到Arrary底层所维护的数组中。当向ArraryList中添加一个对象时，实际上LinkedList内部会生成一个Entry对象，该对象的结构为：

Entry

{

Entry Previous;

Object element;

Entry next;

}

其中，Object类型的元素element就是我们向LinkedList中所添加的元素，然后Entry又构造了向前与向后的应用，最后将生成的这个Entry对象加入到了链表中。换句话说，**LinkedList中所维护的是一个个的Entry对象。**

关于Object类的equal方法的特点：

1. **自反性：x.equal(x)应该返回ture。**
2. **对称性：x.equal(y)为真，则y.equal(x)也应该返回真。**
3. **传递性：x.equal(y)为真，y.equal(z)也为真，则x.equal(z) 也应该返回真。**
4. **一致性：x.equal(y)的第一次调用为真，则x.equal(y)在之后的调用过程也为真，前提条件是在比较之间没有修改x和y。**
5. **非空的x，x.equal(null)为假。**

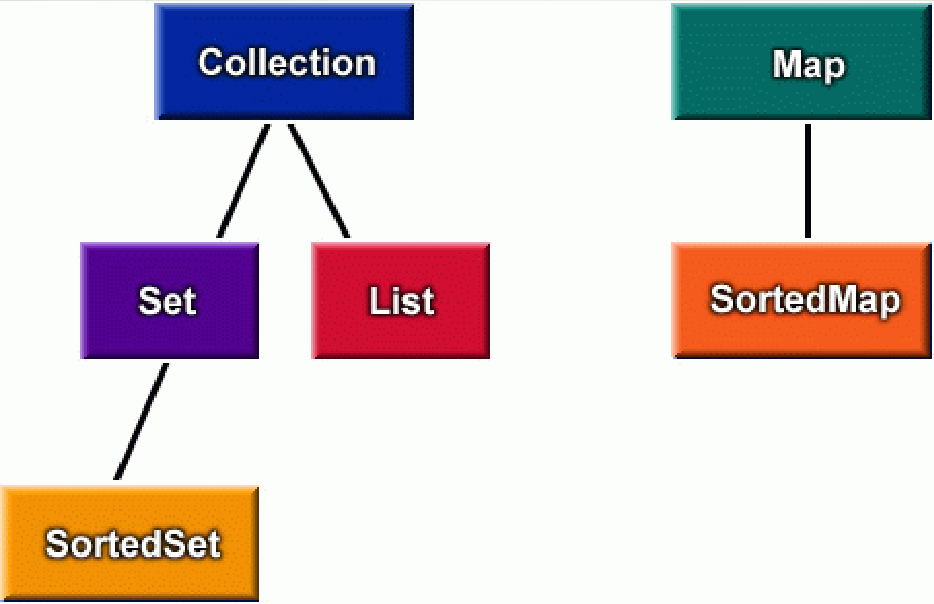
**关于Object类的hashCode()的方法的特点：**

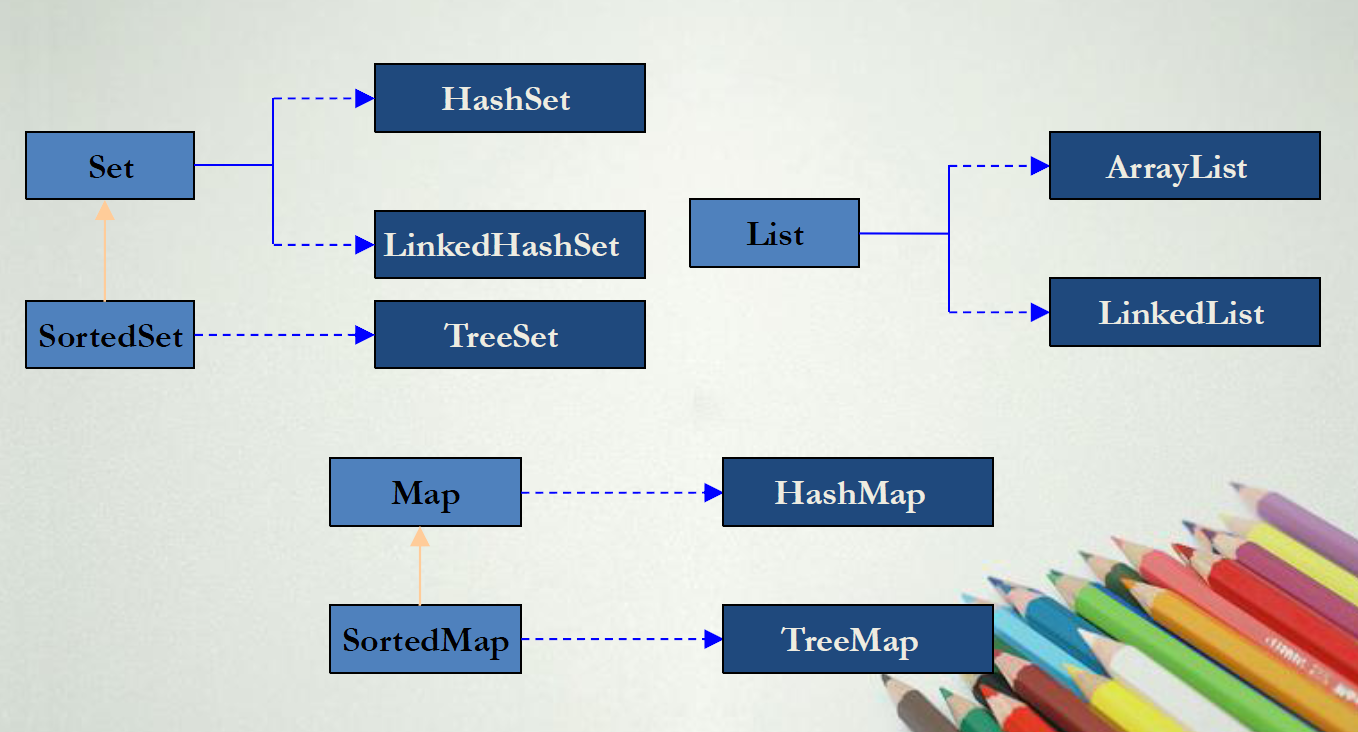
1. **在Java应用的一次执行过程中，对于同一个对象的hashCode方法的多次调用，他们应该返回同样的值(前提是该对象的信息没有发生变化)。**
2. **对于两个对象来说，如果使用equals方法比较返回ture，那么这两个对象的hashCode值一定是相同的。**
3. **对于两个对象来说，如果使用equals方法比较返回false，那么着两个对象的hashCode的值不要求一定不同（可以相同，也可以不同），但是如果不同则额可以提高应用的性能。**
4. **对于Object来说，不同的Object对象的hashCode值是不同的（Object类的hashCode值表示的是对象的地址）。**

**当使用HashSet时，hashCode()方法就会得到调用，判断已经存储在集合中的对象的hash code值是否与增加的对象的hash code一致，如果不一致，直接加进去，如果一致，再进行equals()方法的比较，equals()方法如果返回ture，表示对象已经加进去了，就不会再增加新的对象，否则再加进去。**

**如果重写equals()方法，那么也要重写hashCode()方法，反之亦然。**

Map映射：Map的keySet()方法会返回key的集合，因为Map的键是不能重复的，因此keySet()方法的返回值类型是set，而Map的值是可以重复的，因此values()方法的返回类型是collection，可以容纳重复的元素。





**策略模式：**

HashSet底层是使用HashMap实现的，当使用add方法将对象添加到set当中时，实际上是将该对象底层作为所维护的Map对象的key，而value则都是同一个Object对象。

HashMap底层维护一个数组，向HashMap数组中所放置的对象实际上是存储在该数组当中。

当向HashMap中put一对键值时，它会根据key的hashCode计算一个位置，该位置就是此对象准备往数据中存放的位置，如果该位置没有对象存在，则将此对象直接放在数组中，如果该位置已经有对象存在了，则顺着此存在的对象的链开始寻找（Entry类有一个Entry类型的next成员变量，指向了该对象的下一个对象），如果此链上有对象的话，再去使用equals方法进行比较，如果此链上的某个对象的equals方法比较为false，则将该对象放到数据当中，然后将数组中该位置以前存在的那个对象链接到此对象的后面。

**Java 1.5之后的新特性**

**1.所谓泛型：变量类型的参数化。**

public class ListGenerics<T extends List> //extends 限定了泛型使用类别，指定了这个类必须是继承某个类，或者实现某个接口。默认使用extends Object。

注意以下两者的区别

public class ListGenerics<T extends List>

ListGenerics<? extends List> ge = null;

//ListGenerics<? super List> ge = null; //只接收来自List父类的引用，此时，ArrayList则不可以继续使用，用的比较少

ge = new ListGenerics<ArrayList>();

ge = new ListGenerics<LinkedList>();

第一种情况指定了T可以接受哪些类，是在类的声明时注明，而第二个指定了ge可以指向哪些引用，在对象的声明时指定。

**2.增强型for循环。**

int [][] array = {{1, 3, 4, 5, 6, 7}, {1, 3, 4, 5}, {1, 3, 4, 5}};

for(int[] row : array)

{

for(int element : row)

{

System.out.println(element);

}

}

**3.自动装箱，自动拆箱:将int 3自动装箱成Integer对象，然后加入到ArrayList中，而不是直接将3加入到ArrayList中。**

int a = 3;

Collection<Integer> c = new ArrayList<Integer>();

c.add(new Integer(a));

c.add(a);

c.add(a + 3);

**4.可变参数：可变参数本质上就是一个数组，可以按照可变参数的方式传递参数，也可以传递一个数组。如果一个方法接收两个或者两个以上的参数的话，可变参数必须防止在最后一个位置。**

private static int sum(int... nums)

int result = sum(new int[]{1, 2});

result = sum(1, 2);

**5.枚举：定义的每一个枚举类型都继承自java.lang.Enum类，枚举中的每个成员默认都是public， static， final的。每个枚举的成员其实是枚举的一个实例。换句话说，当定义了一个枚举之后，在编译时刻就能确定该枚举类型有几个实例，分别是什么，在运行期间我们无法再使用该枚举类型创建新的实例了，这些在编译期间就已经完全确定了。（这也是枚举和类的区别）**

**6.静态导入：import static 包名.成员变量（或者方法）；过度使用静态导入会影响代码的可读性。**

**反射：Java语言的一个关键性质，通过反射API取得任何一个已知名称的class的内部信息。**

**程序运行时，允许改变程序结构或者变量类型，这种语言成为动态语言，从这个观点看，Perl，Python和Ruby都是动态语言，C++，Java和C#都不是动态语言。**

**在Java中，无论生成某个类的多少个对象，这些对象都会对应于同一个Class对象。**

如果想使用方法，首先需要获得待处理类的一个对象。

获取每个类或者每个对象所对应class对象的常用三种方式：

1. **使用class类的静态方法forName, Class.forName(“java.lang.String”)**
2. **使用类的class语法，String.class**
3. **使用对象的getClass方法 String s = ”aa”; Class<?> class = s.getClass();**

**若想通过类的不带参数的构造方法来生成对象，有两种方式：**

1. **先获得Class对象，然后通过该Class对象的newInstance()方法直接生成即可： Class<?> classType = Object.class; Object obj2 = classType.newInstance();**
2. **先获得Class对象，然后通过该对象获得对应的Constructor对象，再通过该Constructor对象的newInstance()方法生成：Class<?> classType = Object.class; System.out.println(classType.getName()); Constructor cons = classType.getConstructor(new Class[]{});**
3. **说明：以上两行代码是等价,只不过第一种方法只可以调用不带参数的构造方法去新生成一个对象，如果需要使用带参数的构造方法还需要调用上述两行代码**

**动态代理：**

**Annotation：**

**@Override：注解表示子类要重写父类的方法。**

**@Deprecated：表示方法不建议被使用。**

**@SuppressWarnings("unchecked")，@SuppressWarnings({"unchecked","deprecation"})：压制警告。**

**自定义注解：当注解中的属性名为value时，在对其赋值时可以不指定属性的名称，而直接写上属性值即可，处了value以外的其它值都需要使用name = value这种赋值方式，即明确指定给谁赋值。**

**public @interface AnnotationTest//自定义注解**

**当使用@interface关键字定义一个注解时，该注解隐含地继承了java.lang.annotataion.Annotataion，如果定义了一个接口，并且让该接口继承自Annotation，那么所定义的接口依然还是接口，不是注解。Annotation本身是接口不是注解。可以与Enum类比。**

**Junit(3.8 4.x)：Keep the bar green to keep the code clean.**

**JUnit4的执行的一般流程：首先获得待测试所对象的Class对象，然后通过该Class对象获得当前类中所有的public方法所对应的Method数组。遍历该Method数组，获得每一个Method对象，调用每个Method对象的isAnnotataionPresent(Test.class)方法，判断该方法是否被Test所修饰。如该方法返回true，那么调用metho.invoke()方法去执行该方法，否则不执行。**

**异常：**

**Checked exception(非 Runtime Exception)**

**Unchecked exception(Runtime Exception)**

**Java中所有的异常类都会直接或者间接的继承自Exception,如果异常继承自Exception下面的Runtime Exception类，则为Runtime Exception，否则为非Runtime Exception**

**try**

**{**

**}**

**catch(Exception e)**

**{**

**}**

**finally**

**{**

**}**

**try代码块中，一旦出现异常，Java会自动生成一个异常对象，然后异常后面的代码将不会再被执行，下面的catch则可以捕获到这些异常。try后面可以跟多个catch，但是要注意catch的位置，因为catch是顺序执行的，如果这些catch的异常是继承的关系，需要先将子类的catch放在前面先执行，否则直接执行父类的catch的话，子类的catch则无法到达，编译器会报错。**

**如果try块中存在return语句，那么首先将finally块中的代码执行完毕，然后方法再返回。如果try块中遇到System.exit(0);语句，则不会再去运行finally中的语句。因为System.exit(0)会终止当前运行的Java虚拟机，程序会在虚拟机终止前结束运行。**

**finally代码块中表示始终要运行的程序。**

**public void method() throws Exception**

**{**

**System.out.println("hello world");**

**throw new Exception();**

**}**

**对于Checked exception（非runtime exception）：必须要对其进行处理，处理方式有两种：第一种使用try catch finally进行捕获，第二种是在会产生异常的方法中声明throws exception（注意throw exception（在方法中抛出异常时使用throw exception）和throws exception的区别）。两者的区别在于使用throws exception时，当出现异常时程序将不再继续执行，而使用catch时，程序可以继续运行。**

**对于运行时异常（runtime exception），可以对其处理，也可以不对其进行处理，推荐不对其进行处理。**

**常见的异常：NullPointerException(空指针异常)：出现该异常的原因在于某个引用为null，但却调用了它的某个方法，则会出现该种异常。**

**自定义异常：通常定义了一个继承自Excetpion类的子类。那么这个类就是一个自定义异常。通常情况下，都会直接集成自Exception类，一般不会继承某个运行时的异常类。**

**GUI（Graphic User Interface）:分两种AWT(Abstract Window Toolkit):依赖于底层操作系统,Swing，独立于底层操作系统。**

**AWT：**

**事件：描述发生了什么的对象**

**事件源：事件的产生器**

**事件处理器：接收事件并处理用户交互的方法。**

**当单击一个按钮时就会产生一个事件（ActionEvent），然后会检查是否有该按钮相关的关联事件处理器（实际上就是一个方法），如果没有，那么什么都不执行，如果有，就会将该事件传递给与该按钮关联的事件处理器方法，作为该方法的参数，之后该事件处理器方法就会自动得到调用，并且该方法可以使用传递过来的ActionEvent对象，今儿获得事件发生时与该事件源相关的那些信息。**

**Java中的组件若想添加时间处理器，都会使用形如addXxxListener的方法来添加。**

**Adapters：为简化编程，JDK为大多数事件监听器接口定义了相应的实现类，称之为事件适配(Adapter)类，在适配器类中，实现了相应监听接口所有的方法（因为如果需要使用某个接口必须自己写一个实现某个接口的全部方法（接口实现的定义）），但不做任何事情，只要继承（继承和实现的区别在于，实现必须重写所有的的方法，而继承不需要）适配器，就等于实现了相应的监听器接口。只需要重写需要使用的监听接口，而不需要再重新实现所有的监听接口方法。**

**观察者模式：观察者模式定义了一种一对多的一列关系，让多个观察者对象同时监听某一个主题对象，这个主题对象在状态上发生变化时，会通知所有的观察者对象，让他们能够自动更新自己。**

**抽象主题角色：**

**抽象观察者角色：**

**具体主题角色：**

**具体观察者角色：**

**Swing**

**内部类（Inner Class）：共分4种。**

**静态内部类（Static Inner class）：只可以访问静态成员和静态方法，必须使用关键字static。只要是静态类，必须放在类的内部，外层的类不可以使用static修饰。main函数必须放在public class中。编译之后会单独生成OutClass$InnerClass.class.使用Outter.Inner in = new Outter.Inner();新生成对象。**

**成员内部类（member Inner Class）：和静态内部类的唯一区别就在于是否使用了static关键字修饰，Outter.Inner in = new Outter ().new Inner();如果是外部类引用内部类的对象，则可以使用Inner in = this.new Inner(); 新生成对象。内部类访问外部类的变量使用Outter.this.a.**

**局部内部类（Local Inner Class）：定义在类内方法中的类。局部内部类只可以访问方法中的final类型的变量。只可以在方法内部定义的地方使用该局部内部类**

**匿名内部类（）：局部内部类的特殊形式，没有名字，没有关键字，没有继承，没有实现，默认继承了一个父类，或者实现了一个接口。**

**String str = a.get(new Date()**

**{**

**public String toLocaleString()**

**{**

**return "hello world";**

**}**

**});**

**上述代码，实现了Date子类的一个对象，而不是Date()的对象，匿名内部类继承了Date类，并写重写了Date类的 toLocaleString的方法，匿名内部类也可以用来实现接口，通常匿名内部类在swing中使用的比较多，具体的实例，见视频第八十六讲。**

**递归（Recursion）：方法调用自身，对于递归而言，一定有一个出口，结束递归，这样才不会出现死循环。**

**字节流、字符流：**

**装饰模式（Decorator）：**

**序列化（将文件从内存加载到硬盘）：一个类若想被序列化，则需要使用java.io.Serializable接口，该接口中没有定义任何方法，是一个标识性接口（Maker Interface），当一个类实现了该接口，就表示这个类的对象是可以序列化的。**

**序列化时会出现serialVersionUID相关的警告，该警告作用是当对象序列化之后，存到硬盘中，后来却改变了类的属性，则反序列化时则会出现错误，造成不兼容的问题，但当serialVersionUID相同时，他就会将不一样的field以及type的缺省值反序列化，可以避开这个不兼容的问题（见视频一百零六讲）。**

在序列化时，static变量时无法序列化的，如果A包含了对B的引用，那么序列化A时，也会将B一并地序列化，如果此时A可以序列化，B无法序列化，那么当序列化A时就会发生异常 ，这时需要将对B的引用设置transient

Transient关键字标识不被序列化，static变量也不会被序列化。

Java中同时提供了如下两个方法，这两个方法可以自动被调用，可以控制哪些对象可以实现序列化。可以更加细粒度的方式实现序列化和反序列化。

**private void writeObject(java.io.ObjectOutputStream out)throws IOException**

**{**

**out.writeInt(age);**

**out.writeUTF(name);**

**System.out.println("write object");**

**}**

**private void readObject(java.io.ObjectInputStream in)throws IOException, ClassNotFoundException**

**{**

**age = in.readInt();**

**name = in.readUTF();**

**System.out.println("real object");**

**}**

多线程（Multi-Thread）：java中如果我们自己没有产生线程，那么系统会自动产生一个线程（主线程、main方法就是在主线程上运行），我们的程序都是由线程来完成的。

进程：执行中的程序（程序是静态的概念，进程是动态的概念）。

Java中有两种方法实现多线程：

1.**继承Thread类，并重写run方法，**将希望之星的代码放入run方法中，然后通过start方法来启动线程，start方法首先为线程的执行准备好系统资源，然后点去调用run方法。当每个类继承了Thread类之后，这个类就叫做一个线程类。

2.**通过定义实现Runnable接口的类进而实现run方法。**

当使用第一种方式来生成线程对象时，我们需要重写run方法，因为Thread类的run方法此时什么事情都不做。当使用第二种方式来生成线程对象时，我们需要实现Runnable接口的run方法，然后使用new Thread(new MyThread())来生成新的线程对象，这时的线程对象的run方法就会调用MyThread类的方法，这样我们自己编写的run方法就执行了。

一个进程，至少包含一个线程，对于单核CPU来说，某一时刻只能有一个线程在执行(微观线程)，从宏观角度来看，多个线程在同时执行（宏观并行），对于双核或者双核以上的CPU来说，可以做到真正微观并行。

Thread类也实现了Runnable接口，因此实现了Runnable接口中的run方法。

停止线程的方式：不能使用Thread类的stop方法来终止线程的执行，一般要设定一个变量，在run方法中循环，循环每次检查该变量，如果满足条件则继续执行，否则跳出循环，线程结束。

不能依靠线程的优先级决定线程的执行顺序。

关于成员变量与局部变量：如果一个变量是成员变量，那么多个线程对同一个对象的成员变量进行操作时，他们对该成员变量是彼此影响的（也就是说一个 线程对成员变量的改变会影响另外一个），如果一个变量是局部变量，那么每个线程都会有一个该局部变量一个拷贝，一个线程对该局部变量的改变不会影响到其它线程。（视频第一百讲） **synchronized关键字：**当synchronized关键字修饰一个方法时，这个方法叫做同步方法。

Java中的每个对象都有一个锁，这个锁叫做监视器（monitor），当访问某个对象的synchronized方法时，表示将该**对象**（并不是对方法上锁，尤其是一个对象含有多个方法时，线程只要调用了该对象中的synchronized方法，任何其它的线程都无法访问该对象，具体例子见视频第一百零一讲）上锁，此时其它任何线程都无法再次访问该synchronized方法，直至之前上锁的线程执行完毕，或者抛出了异常，那么将该锁释放掉，其它线程才有可能再去访问synchronized方法。

当被访问的方法同时被synchronized和static同时修饰时，由于static方法属于class对象，所以当线程分别访问同一个类的两个对象的两个static， synchronized方法时，**锁定的是static对应的class类**，因此他们的执行顺序是顺序的。

synchronized块，会对object上锁。synchronized块可以更加细粒度的锁定线程，推荐使用synchronized块

Object obj = new Object（）

synchronized（object）

{

}

Object obj = new Object（）

synchronized（this）

{

}

wait（）与notify（）方法都是定义在Object中，而且是final的，因此会被所有的Java类所继承，并且无法重写，这两个方法要求在调用时线程已经获得了对象的锁，因此，这两个方法的调用要放在synchronized方法或者块中，**当线程执行了wait方法时，它会释放掉对象的锁，**

另一个会导致线程暂停的方法就是Thread类的sleep方法，它会导致线程睡眠指定的毫秒数，**但线程在睡眠的过程中是不会释放掉对象的锁。**

深拷贝（deep clone）：深拷贝把所有引用的对象都复制了一遍。（序列化可以实现深克隆，剪视频第一百零六讲）

浅拷贝（shallow clone）：浅拷贝仅赋值所考虑的对象，不复制对象的引用，只复制原生数据类型（见视频第一百零五讲）。

简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

ClassLoader（类加载器）：JVM提供了三类加载器：

1. 根类加载器（使用C++编写，程序员无法再Java代码中获得该类）
2. 扩展加载器，使用Java代码编写。
3. 系统加载器（又叫做应用加载器），使用Java代码编写。

类加载器的父委托机制，父子加载器并非继承关系，也就是说子加载器不一定继承了父加载器。

如果某个类加载器能够加载一个类，那么该类加载器就称作定义加载器，定义加载器及其所有子加载器都成最初始类加载器。

当生成一个自定义的类加载器，如果没有指定它的父加载器，那么系统类加载器就将成为该类加载器的父类加载器。

XML（eXtensible Makeable Language）：

DTD（Document Type Definition）：文档类型定义

XML中的每个元素都是成对出现的（有开始，有结束）<student> </student>

XML语法要求非常严格，不能出现空格的地方绝对不能加任何的空白。

XML文档必须且只能有一个根元素。

如果元素之间没有内容，可以写为<score></score> 或者直接写作<score/>

XML中5种内置实体：

|  |  |
| --- | --- |
| &lt; | < |
| &gt; | > |
| &amp; | & |
| &quto; | “ |
| &apos; | ‘ |

当然也可以自定义实体。

属性中不能包含<,>和&符号。在一个元素上，相同的属性只能出现一次。

CDATA 用来存储不需要XML做任何处理的文本。

XML元素可以具有属性，属性的形式为：属性名=“属性值”，需要使用双引号或者单引号标识，多个属性之间使用空格隔开。

通过样式，可以将，现在有不止一种样式语言可以使用，一种CSS（Cascading Style Sheets），另外一种可能的选择是可扩展的样式语言XSL（Extensible Style Language）

XML的注释：<!—- comment --> //注释不允许嵌套。

格式正规（well formed）的XML文档，符合XML语法要求的XML文档就是格式正规的XML文档。

有效的（valid）XML文档，首先是个正规的XML文档，然后又需要满足DTD的要求，这样的XML文档称为有效的XML文档。

内部DTD文档<!DOCTYPE 根元素[定义内容]>

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE poem [

<!ELEMENT poem (author, title, content)>

<!ELEMENT author (#PCDATA)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT content (#PCDATA)>

]>

<poem>

<author>王维</author>

<title>鹿柴</title>

<content>空山不见人，但闻人语声</content>

</poem>

外部DTD文档<!DOCTYPE 根元素SYSTEM “DTD文件路径”>：外部DTD优势在于可以复用，可以限制多个xml文档。

DTD语法：

<!ELEMENT NAME CONTENT>

CONTENT 可以为：EMPTY，即该元素不能包含**子元素**和文本，但可以有属性（空元素），ANY，该元素可以包含任何在DTD中定义的元素，#PCDATD，可以包含任何字符数据，但是不能在其中定义任何**子元素**

修饰符号(修饰元素的)

|  |  |  |  |
| --- | --- | --- | --- |
| 符号 | 用途 | 示例 | 说明 |
| （） | 分组 | （张三，李四），（王朔，于杰），毛毛 | 分成三组 |
| | | 在列出的对象中选择一个 | （男人|女人） | 男人或者女人必须出现，二选一 |
| + | 该对象至少出现一次 | （成员+） | 成员必须出现，而且至少出现一次 |
| \* | 该对象可以出现零次到多次 | （爱好\*） | 爱好可以出现零次到多次 |
| ？ | 该对象可以出现，但只能出现一次 | （菜鸟？） | 可以出现，也可以不出现，如果出现，只能出现一次 |
| ， | 对象必须按指定顺序出现 | （西瓜，苹果，香蕉） | 西瓜，苹果，香蕉必须全部出现，而且必须按照这个顺序。 |

属性的定义语法：

<!ATTLIST 元素名称 属性名称 类型（CDATA ID IDREF NMTOKEN（属性值不能包含空格）/NMTOKENS Enumerated） 属性特点（#REQUIRED #IMPLIED #FIXED value）

实体语法：

<!ENTITY 实体名 “实体值”>

DTD示例：

<!ENTITY writer “Donald Duck”>

<!ENTITY copywriter “CopyWriter W3Schools”>

XML示例：

<author>&writer; &copywriter;</author>

外部实体语法：

<!ENTITY 实体名 SYSTEM “URL/URL”> //使用网址的内容来替换实习，如果不加SYSTEM,则表示使用网址字符串来代替实体。

实体分普通实体和参数实体，普通实体是在dtd中定义，在xml中使用，参数实体是在dtd中定义在dtd中使用。

#PCDATA（Parsed Character Data）,可解析的字符数据。

每一个XML文档都有且只有一个根元素。

XML描述的是文档的内容与语义，而不是文档应该如何显示。

DTD缺陷：DTD不遵守XML语法，DTD数据类型有限，DTD不可扩展，DTD不支持命名空间（命名冲突）

Schema（模式）：其作用与DTD一样，也是用于验证XML文档的有效性，此外Schema也是一个XML文件，而DTD却不是。

Schema新特性:Schema 基于XML语法，可以用来处理XML文档，Schema大大扩展了数据类型，可以自定义数据类型，而且支持元素的继承，支持组属性。

SimpleType类型的元素中不能包含元素或者属性。当需要定义的元素包含了子元素或者属性时，必须使用ComplexType。

SimpleContent用于ComplexType元素上，用于限定ComplexType的内容类型，表示该ComplexType没有子元素，同时ComplexType需要有属性，即一定会有attribute，否则就成为SimpleType了。

通过DOCTYPE可以明确指定文档的根元素，因为DOCTYPE后面跟的元素就是文档的根元素，通过Schema是没法明确指定目标XML文档的根元素，XMlspy是他欧诺个过推断哪个元素包含了其它元素来选择根元素。

HTML是肤浅的，只关心外观

XML解析：DOM（Document Object Mode） DOM就是一个对象化的XML数据接口，一个与语言无关，平台无关的标准接口。

DOM最大缺陷是需要将xml文档读入到内存中，如果xml文档很大的话，需要占用大量的内存空间，

DOM中要严格区分根节点和根元素节点，根节点（Document）代表XML文档本身，是解析XML文档的入口，根元素节点则表示XML根元素，它对应XML文档的root

JAXP(Java API for XML Parsing):用于XML解析的Java API

**对于XML解析来说，空格也算是节点孩子的组成部分。**

SAX(Simple APIs for XML):面向XML的简单API，使用DOM解析XML时，首先将XML文档加载到内存中，然后可以通过随机访问的方式访问内存中的DOM树，SAX是基于时间而且是顺序访问执行的，一旦经过了某个元素，没有办法再去访问它了，SAX不比事先将真个XML文档加载到内存中，因此站如内存要比DOM小，对于大型XML文档来说，通常使用SAX而不是DOM进行解析。

SAX也是使用观察者模式（类似于GUI）

JDOM：使用Java语言读、写、操作XML的新API方法。

Java中的方法链编程风格（method chain style），JDOM中即使用了方法链的编程风格。例如：

e.addContent(new Element(“aaa”).setAttribute(“a”, “b”).setAttribute(“x”, ”y”)); //即通过setAttribute返回element对象，然后不断的添加属性值，即方法链的编程风格。

Ant（）：将源代码转换为可执行文件的形式。

数组命名：

int a [];

a=new int[10];// 使用两条语句来完成，或者通过以下的一条语句来完成。

int a[]=new int[10];

如果在使用数组之前直接对数组进行赋值可以进行如下的操作：

Int a[]={1,2,3}; //即并不需要对数据的大小进行声明。

对于二维数据，java允许每个维度的数组长度不相等，如下的表达式是允许成立的：

int num[][]={{1,2,3},{1,2,3,4},{1,2}};

num.length;// 二维数组的行数

num[0].length;// 第一行含有的数组个数。

面向对象的概念：传统的过程式编程语言以过程为中心、以算法为驱动，面向对象则以对象为中心，以消息为驱动。以公式表示为，过程式编程语言为：程序=算法+数据，面向对象编程语言为：程序=对象+消息。

一个对象由一组属性和一组对属性进行操作的方法组成。

Java类属性和对象属性的初始化顺序如下：

1. 类属性（静态变量）定义时的初始化。
2. Static块中的初始化代码。
3. 对象属性（非静态变量的初始化）。
4. 构造函数中的初始化代码。

构造方法的初始化：

在Java程序中，构造方法所完成的主要工作是帮助新创建的对象赋初值，使用构造方法时需要注意以下两点：

1. 它具有与类名相同的名称。
2. 它没有返回值。
3. 调用机制不同，一般的方法时需要使用时才调用，而构造方法是在创建对象时自动执行。
4. 构造方法是可以重载的，可以通过构造方法的个数不同或者类型不同进行重载。

在java中，类的**继承**使用关键字extends来实现。在Java中多重继承是不允许的，即如下的代码是不正确的：

class A

{}

class B

{}

class C extends A,B

{}

但是上述的代码可以通过多层继承来实现，即：

class A

{}

class B extends A

{}

class C extends B

{}

**重载**，在一个类定义中，可以编写几个同名的方法，但是只要它们的签名参数列表不同，Java就会将它们看做唯一的方法。简单的说，一个类中的方法与另一个方法同名，但是参数表不同，这种方法称之为重载方法。

**覆写**的概念与重载相似，均是java的多态技术之一，当一个子类继承一个父类，子类中的方法与父类的方法的名称、参数个数、类型完全一致时，就称子类中的这个方法覆写了父类中的方法。

在Java中类的**多态性**分两种：向上转型和向下转型。

向上转型：父类对象通过子类对象去实例化，会出现丢失精度的问题。（自动完成）

向下转型：父类对象可以转化为子类对象（需要进行强制类型转换）。

**抽象类的定义如下：**

1. **抽象类和抽象方法都必须使用abstract关键字来修饰。**
2. **抽象类不能被直接实例化，也就是不能直接使用new关键字去产生对象。**
3. **抽象方法只需声明，不需实现。**
4. **含有抽象方法的类必须被声明为抽象类，抽象类的子类必须覆写所有的抽象方法之后才能被实例化，否则这个子例还是抽象类。**

Interface 是Java提供的另一种重要技术，它的结构和抽象类非常相似，也具有数据成员与抽象方法，但它与抽象类又有一下区别：

1. 接口里的数据成员必须初始化，且数据成员均为常量，不可以再进行更改（使用final进行修饰，也正因如此，final可以省略）。
2. 接口里的方法必须全部声明为abstract（正因如此，abstract都可以省略），也就是说接口不可以像抽象类一样保有一般的方法，必须全部为抽象方法。

子类调用抽象类使用关键字extends，而调用接口使用关键字implements来实现。接口也可以实现继承，通过关键字extends来实现，而且接口也是可以继承多个接口。

**接口与抽象类的最大区别为：接口是Java实现多继承的一种机制，一个类只可以继承一个父类，但是可以实现多个接口。**

Java中有一个比较特殊的类，即**Object**类，它是所有类的父类，如果一个类没有使用extends关键字明确标识继承另外一个类，那么这个类就是默认继承Object类。

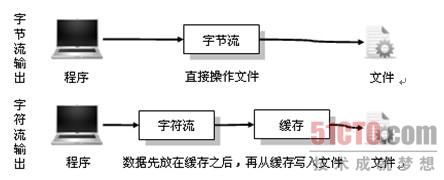
内部类：在类内部或者类内部的方法内部可以再次定义类，但是外部类无法访问内部类中所声明的属性，而内部类则可以访问外部类的属性。而且内部类也可以实现继承或者接口。

Java中没有指针，但是有引用。

程序员面试常考问题：

1. 逻辑短路问题
2. Break语句，如果某个case语句中缺失break，则会跑到下面一个case，直至遇到下面一个break，break只跳出一层循环，联想到return的使用，由于方法不可以嵌套，return直接返回出方法。
3. 注意区分引用传递和值传递对结果的不同影响（见视频第十一讲）。

**字节流与字符流的区别**

字节流与和字符流的使用非常相似，两者除了操作代码上的不同之外，是否还有其他的不同呢？  
  
实际上字节流在操作时本身不会用到缓冲区（内存），是文件本身直接操作的，而字符流在操作时使用了缓冲区，通过缓冲区再操作文件，如图12-6所示。  
  
下面以两个写文件的操作为主进行比较，但是在操作时字节流和字符流的操作完成之后都不关闭输出流。  
  
范例：使用字节流不关闭执行

Java代码  收藏代码

1. **package** org.lxh.demo12.byteiodemo;
2. **import** java.io.File;
3. **import** java.io.FileOutputStream;
4. **import** java.io.OutputStream;
5. **public** **class** OutputStreamDemo05 {
6. **public** **static** **void** main(String[] args) **throws** Exception {   // 异常抛出，  不处理
7. // 第1步：使用File类找到一个文件
8. File f = **new** File("d:" + File.separator + "test.txt"); // 声明File  对象
9. // 第2步：通过子类实例化父类对象
10. OutputStream out = **null**;
11. // 准备好一个输出的对象
12. out = **new** FileOutputStream(f);
13. // 通过对象多态性进行实例化
14. // 第3步：进行写操作
15. String str = "Hello World!!!";
16. // 准备一个字符串
17. **byte** b[] = str.getBytes();
18. // 字符串转byte数组
19. out.write(b);
20. // 将内容输出
21. // 第4步：关闭输出流
22. // out.close();
23. // 此时没有关闭
24. }
25. }

程序运行结果：  
  
此时没有关闭字节流操作，但是文件中也依然存在了输出的内容，证明字节流是直接操作文件本身的。而下面继续使用字符流完成，再观察效果。  
  
范例：使用字符流不关闭执行

Java代码  收藏代码

1. **package** org.lxh.demo12.chariodemo;
2. **import** java.io.File;
3. **import** java.io.FileWriter;
4. **import** java.io.Writer;
5. **public** **class** WriterDemo03 {
6. **public** **static** **void** main(String[] args) **throws** Exception { // 异常抛出，  不处理
7. // 第1步：使用File类找到一个文件
8. File f = **new** File("d:" + File.separator + "test.txt");// 声明File 对象
9. // 第2步：通过子类实例化父类对象
10. Writer out = **null**;
11. // 准备好一个输出的对象
12. out = **new** FileWriter(f);
13. // 通过对象多态性进行实例化
14. // 第3步：进行写操作
15. String str = "Hello World!!!";
16. // 准备一个字符串
17. out.write(str);
18. // 将内容输出
19. // 第4步：关闭输出流
20. // out.close();
21. // 此时没有关闭
22. }
23. }

程序运行结果：  
  
  
  
  
程序运行后会发现文件中没有任何内容，这是因为字符流操作时使用了缓冲区，而   在关闭字符流时会强制性地将缓冲区中的内容进行输出，但是如果程序没有关闭，则缓冲区中的内容是无法输出的，所以得出结论：字符流使用了缓冲区，而字节流没有使用缓冲区。  
  
提问：什么叫缓冲区？  
  
在很多地方都碰到缓冲区这个名词，那么到底什么是缓冲区？又有什么作用呢？  
  
回答：缓冲区可以简单地理解为一段内存区域。  
  
可以简单地把缓冲区理解为一段特殊的内存。  
  
某些情况下，如果一个程序频繁地操作一个资源（如文件或数据库），则性能会很低，此时为了提升性能，就可以将一部分数据暂时读入到内存的一块区域之中，以后直接从此区域中读取数据即可，因为读取内存速度会比较快，这样可以提升程序的性能。  
  
在字符流的操作中，所有的字符都是在内存中形成的，在输出前会将所有的内容暂时保存在内存之中，所以使用了缓冲区暂存数据。  
  
如果想在不关闭时也可以将字符流的内容全部输出，则可以使用Writer类中的flush()方法完成。  
  
范例：强制性清空缓冲区

Java代码  收藏代码

1. **package** org.lxh.demo12.chariodemo;
2. **import** java.io.File;
3. **import** java.io.FileWriter;
4. **import** java.io.Writer;
5. **public** **class** WriterDemo04 {
6. **public** **static** **void** main(String[] args) **throws** Exception { // 异常抛出不处理
7. // 第1步：使用File类找到一个文件
8. File f = **new** File("d:" + File.separator + "test.txt");// 声明File
9. 对象
10. // 第2步：通过子类实例化父类对象
11. Writer out = **null**;
12. // 准备好一个输出的对象
13. out = **new** FileWriter(f);
14. // 通过对象多态性进行实例化
15. // 第3步：进行写操作
16. String str = "Hello World!!!";
17. // 准备一个字符串
18. out.write(str);
19. // 将内容输出
20. out.flush();
21. // 强制性清空缓冲区中的内容
22. // 第4步：关闭输出流
23. // out.close();
24. // 此时没有关闭
25. }
26. }

程序运行结果：  
  
此时，文件中已经存在了内容，更进一步证明内容是保存在缓冲区的。这一点在读者日后的开发中要特别引起注意。  
  
提问：使用字节流好还是字符流好？  
  
学习完字节流和字符流的基本操作后，已经大概地明白了操作流程的各个区别，那么在开发中是使用字节流好还是字符流好呢？  
  
回答：使用字节流更好。  
  
在回答之前，先为读者讲解这样的一个概念，所有的文件在硬盘或在传输时都是以字节的方式进行的，包括图片等都是按字节的方式存储的，而字符是只有在内存中才会形成，所以在开发中，字节流使用较为广泛。  
  
  
字节流与字符流主要的区别是他们的的处理方式  
  
流分类：  
1.Java的字节流  
   InputStream是所有字节输入流的祖先，而OutputStream是所有字节输出流的祖先。  
2.Java的字符流  
  Reader是所有读取字符串输入流的祖先，而writer是所有输出字符串的祖先。  
InputStream，OutputStream,Reader,writer都是抽象类。所以不能直接new   
  
  
  
字节流是最基本的，所有的InputStream和OutputStream的子类都是,主要用在处理二进制数据，它是按字节来处理的  
但实际中很多的数据是文本，又提出了字符流的概念，它是按虚拟机的encode来处理，也就是要进行字符集的转化  
这两个之间通过 InputStreamReader,OutputStreamWriter来关联，实际上是通过byte[]和String来关联  
在实际开发中出现的汉字问题实际上都是在字符流和字节流之间转化不统一而造成的  
  
在从字节流转化为字符流时，实际上就是byte[]转化为String时，  
public String(byte bytes[], String charsetName)  
有一个关键的参数字符集编码，通常我们都省略了，那系统就用操作系统的lang  
而在字符流转化为字节流时，实际上是String转化为byte[]时，  
byte[]    String.getBytes(String charsetName)  
也是一样的道理  
  
至于java.io中还出现了许多其他的流，按主要是为了提高性能和使用方便，  
如BufferedInputStream,PipedInputStream等

**常用设计模式**

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其实还有两类：并发型模式和线程池模式。

**1.工厂模式：**代码不再叙述，对于工厂的共同接口，可以使用接口实现也可以使用抽象类实现，创建工厂时，工厂方法最好使用static修饰，这样就直接可以访问工厂方法，而不需要使用new去访问，访问起来比较方便。

**2.单例模式：**单例模式（Singleton）是一种常用的设计模式。在Java应用中，单例模式能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

**3.适配器模式：**适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

总结一下三种适配器模式的应用场景：类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

**4.装饰模式：**顾名思义，装饰模式就是给一个对象增加一些新的功能，装饰器模式的应用场景：

1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

缺点：产生过多相似的对象，不易排错！

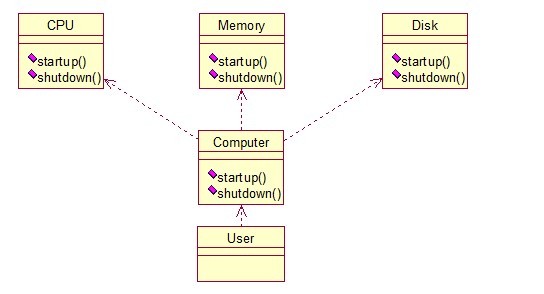
**5.代理模式：**代理模式的应用场景：如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

**6.外观模式：**外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口，看下类图：（我们以一个计算机的启动过程为例）



如果我们没有Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会带来其他类的修改，这不是我们想要看到的，有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这就是外观模式！

先来张图，看看这11行为型模式的关系：

第一类：通过父类与子类的关系进行实现。第二类：两个类之间。第三类：类的状态。第四类：通过中间类



**7.策略模式（strategy）：**策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数，关系图如下：

**8：观察者模式：**包括这个模式在内的接下来的四个模式，都是类和类之间的关系，不涉及到继承，学的时候应该 记得归纳，记得本文最开始的那个图。观察者模式很好理解，类似于邮件订阅和RSS订阅，当我们浏览一些博客或wiki时，经常会看到RSS图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。

Eclipse使用快捷键：

Alt + / ：会开启辅助提示，例如输入main的时候按快捷键可以自动补上后面的内容。或者直接打入sysout，然后按住辅助键就可以跳出System.out.println();

Ctrl + /：直接注释本行，或者取消本行的注释。

在输入{或者（时直接按回车，会自动跳出下面的内容，或者跳至行尾。

Ctrl +Shift +o: 可以先使用该包，然后通过该快捷键自动导入包，即类中会自动添加import java.util.ArrayList;

Ctrl + d：删除本行

Alt+ <-(左箭头)：后退

Ctrl + t：任意点一个方法，然后按住Ctrl + t即可以看到所有的策略模式。

Java中\表示转义字符，而windows目录使用\表示目录，所以Java中如果需要访问windows中的目录，需要使用\\来访问（也可以使用/直接访问，这样便于代码的移植），第一个\将第二个\转义为反斜杠。其它（linux）系统中，目录都是使用/表示，所以不需要是用\进行转义。

Ctrl + 1：放在语句上，按住次快捷键会提出出现的错误。

Alt +Shift+r：同时更改工程中的所有变量。

Ctrl +Shift+f：格式化eclipse中的代码。

Ctrl +Shift+w：关闭eclipse中所有的窗口。