1 java中的equals是用于比较字符串本身，==是用于比较两个对象的地址

String s1 = "hello";

String s2 = **new** String("hello");

String s3 = "hel" + "lo";

System.*out*.println(s1 == s2);//false

System.*out*.println(s1 == s3);//true

System.*out*.println(s1.equals(s2));//true

2 比较下列两个例子

在hashset中不允许出现重复对象，元素的位置也是不确定的，在hashset中友是怎样判断重复的呢？

1. 判断两个对象的hashCode是否相等；

如果不相等，认为两个对象也不相等，完毕

如果相等，转入2）

1. 判断两个对象用equals晕眩是否相等

如果不相等，认为两个对象不相等

如果相等，认识两个对象相等（equals是判断两个对象是否相等的关键）

HashSet hashset = **new** HashSet();

hashset.add(**new** Student("wuqianhu", 20));

hashset.add(**new** Student("wuqianhu", 20));

Iterator iter = hashset.iterator();

**while**(iter.hasNext())

{

Student obj = (Student)iter.next();

System.*out*.println(obj.name + ": " + obj.age);

}//输出为wuqianhu 20 wuqianhu 20

HashSet strHashSet = **new** HashSet();

strHashSet.add(**new** String("zhongguo"));

strHashSet.add(**new** String("zhongguo"));

iter = strHashSet.iterator();

**while**(iter.hasNext())

{

String obj = (String)iter.next();

System.*out*.println(obj);

}//输出为zhongguo

在第一种情况下，用equals比较的是地址，因为Student类没有重写equals方法

在第二种情况下，用equals比较的是值，因为String类重写了equals方法，而且先比较的hashCode也是比较的地址，因此可以重写hashCode方法与equals方法

**public** **int** hashCode()

{

System.*out*.println("hashCode: " + age + name.hashCode());

**return** age \* name.hashCode();

}

**public** **boolean** equals(Object obj)

{

Student s = (Student)obj;

System.*out*.print("equals: ");

System.*out*.println(age == s.age && name.equals(s.name));

**return** age == s.age && name.equals(s.name);

}//输出为wuqianhu 20

3 关于String中的nextLine函数与next函数

nextLine是读取一行字符串

next是读取下一个单词，以空格符或换行符作为分隔单词的标记

4 java可以支持非规则的二维数组

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

a[0] = **new** **int**[2];

a[1] = **new** **int**[3];

a[2] = **new** **int**[3];

一维数组的拷贝：

**int**[] srcArray = {5, 5, 5, 5, 5};

**int**[] destArray = {1, 2, 3, 4, 5, 6, 7, 8 ,9};

System.*arraycopy*(srcArray, 0, destArray, 0, srcArray.length);

**for**(**int** i = 0; i < destArray.length; i++)

System.*out*.print(destArray[i] + " ");//输出微5 5 5 5 5 6 7 8 9

二维数组的拷贝：

**int**[][] a = {{1, 2}, {3, 4, 5}, {6, 7, 8, 9}};

**int**[][] b = {{5, 5}, {5, 5}, {5, 5}};

System.*arraycopy*(a, 0, b, 0, a.length);

**for**(**int** i = 0; i < b.length; i++)

{

**for**(**int** j = 0; j < b[i].length; j++)

System.*out*.print(b[i][j] + " ");

System.*out*.println();

}//输出为 1 2

//3 4 5

//6 7 8 9

可以考虑为将一维数组的引用拷贝复制了

数组的排序

**int**[] a = {6, 23, 45, 12, -83, 94, 41};

Arrays.*sort*(a);

**for**(**int** i = 0; i < a.length; i++)

System.*out*.print(a[i] + " ");

5 JDK API中常用包

Java.lang 包含了Java语言程序设计的基础类

Java.awt 包含用于创建图形用户界面和绘制图形图像的相关类

Java.util 包含集合、日期、国际化和各种实用工具类

Java.io 包含可提供数据输入/输出相关功能的类

Java.net 提供用于实现java网络编程的相关功能类

Java.sql 提供数据库操作相关功能类

6 Java中只支持单重继承，不支持多重继承

7 Java类时间的关系：

1. 依赖关系

依赖关系是指一个类的方法使用到了另外一个类的对象

也可以被用来称为“使用关系”，即use-a关系

Class Goods{}

Public class Car{

Public void load(Goods g){}

}

1. 聚合关系

聚合关系是指一个类的对象包含（通过属性引用）了另一个类的对象

也可以被用来称为“拥有关系”，即“has-a”关系

Public class Book{

Private String name;

Private java.util.Date.publishDate;

}

1. 泛化关系

泛化关系也被称为一般化关系或者“is-a”关系，表示的是类之间的继承关系（还包括类和接口之间的实现关系以及接口之间的继承关系）

在声明一个类时可以在class关键字前使用public来修饰，也可以不使用该修饰符。如果一个类没有声明微public的，则该类只能在其所在包中使用，在其他包中无法引用或使用该类

Java中的成员访问控制有四种形式：public protected 无修饰符 private

8 关于java中派生类属性与基类属性同名的问题

Person类：（基类）

**protected** **int** no = 1;

**public** **int** getNo(){

**return** no;

}

**public** **int** getBaseNo(){

**return** no;

}

Solider类（派生类）

**protected** **int** no = 10;

**public** **int** getNo(){

**return** no;

}

System.*out*.println(person.getNo());//输出为1

System.*out*.println(person.getBaseNo());//输出为1

System.*out*.println(solider.getBaseNo());//输出为1

System.*out*.println(solider.getNo());//输出为10

这个可以理解为在哪个层次的累得代码中，就优先使用该层次类中定义的属性，如果没有才会追溯调用其父类中声明的属性

9 对象造型

在多态情况下，由于对象以其父类的身份出现，对子类中新添加的成员的访问收到限制，有时我们可能又需要回复一个对象的本来面目---造型，所谓对象造型就是引用类型数据值之间的强制类型转换

Person p = **new** Solider("wuqianhu", 10, "hello");

Solider q = (Solider)p; //需要强制类型转换

System.*out*.println(q.getType());

10 instanceof用于判断当前调用对象的真实类型

sample\_1\_1 p = **new** sample\_1\_1();

Person q = **new** Solider("wuqianhu", 20, "hel");

p.show(q);//输出 这是Solider类型及其子类型对象

**public** **void** show(Person obj)

{

**if**(obj **instanceof** Solider)

System.*out*.println("这是Solider类型及其子类型对象 ");

**else**

System.*out*.println("这是Person类型及其子类型");

}

11 协变返回类型

协变返回类型是从Java SE5.0开始引用的一种新机制-允许重写方法时修改器返回值的类型，但必须是重写钱方法返回值类型的子类或实现类类型

12 使用this调用其他重载构造函数

**public** Shot(**int** begin, **int** end)

{

**this**.begin = begin;

**this**.end = end;

}

**public** Shot(**int** begin)

{

**this**(begin, 100);

}

Shot shot = **new** Shot(60);

shot.show();//输出 begin: 60 end: 100

这种方式的构造方法调用与用关键字new调用的方式有所不同，此时只是将被调用构造方法体中的语句执行一遍，而不会再创建一个新的Shot对象，因此其作用主要是代码重用

**对于调用重载法官法的多重匹配问题**

在遇到这种情况时，编译器是以最低的类型转换开销为准来进行方法调用的，所谓的类型转换开销可以理解微不同类型数据值之间转换时所需的存储空间和处理器运算时间上的消耗

13 初始化快

**静态初始化块：**

在定义Java类时候，还可以在类体重、方法的外部包含一个或多个static标记的语句块，这些语句仅在其所属的类被载入JVM时被执行一次，通常用于一次性的初始化工作

**public** **class** test {

**public** **static** **int** *total*;

**static**

{

*total* = 100;

System.*out*.println("in static block");

}

**public** test()

{

System.*out*.println("test Constructor");

}

}

test p1 = **new** test();

test p2 = **new** test();

输出为：

in static block

test Constructor

test Constructor

static初始化快无法人为调用，而只是在其所在类被载入JVM内存时自动执行一次，其具体实际是在改类的所有static属性已被分配空间、默认初始化和显式初始化之后，在执行任何其他代码之前。如果一个java类中同时出现多个static块，泽按照其出现的先后顺序来执行

**非静态初始化块：**

**public** **class** test {

**public** **static** **int** *total*;

{

*total* = 100;

System.*out*.println("in no static block");

}

**public** test()

{

System.*out*.println("test Constructor");

}

}

test p1 = **new** test();

test p2 = **new** test();

输出为：

in no static block

test Constructor

in no static block

test Constructor

非static的初始化语句块会在改类的对象创建时（其所在类的每个构造方法执行过程中）被调用一次，其具体运行时机是在对象分配空间、实例变量默认和显式初始化之后、执行构造方法体重语句之前

14 单例模式

**public** **class** Single {

**private** String name;

**private** **static** Single *onlyone* = **new** Single();

**private** Single(){

}

**public** **void** setName(String name){

**this**.name = name;

}

**public** String getName(){

**return** name;

}

**public** **static** Single getSingle(){

**return** *onlyone*;

}

}

15 final类和final方法

声明微final的java类不能被继承，否则编译将出错。因此当我们认为一个类是完备的或通用性的，且不需要、也不允许将来定义子类继承它时，可以将该类声明微final的

Public final class A{}

忧郁final不会有子类，因此其中的方法将永远不会被重写

声明为final的方法不能被子类重写，否则将编译出错。有时我们允许自己定义的类被继承，但又要保护其中的部分方法不被重写，泽可以在每一个要保护的方法前加上final修饰，而不降整个类声明微final

Public class A{

Public final void m1（）{}

Public int m2(){}

}

Final修饰符出了能降方法锁定、防止任何子类篡改它的功能和具体实现之外，还有一个可能的好处---java语言规范允许编译器对final方法的调用进行优化，即将函数调用进行inline内联化，这样就会省去了多次函数调用时的栈内存操作，会提高效率

Final标记的变量为常量—只能赋值一次，可以在声明的同时赋初值，也可以单独赋值

16 抽象类

**public** **abstract** **class** Animal {

**private** **int** age;

**public** **void** setAge(**int** age){

**this**.age = age;

}

**public** **int** getAge(){

**return** age;

}

**public** **abstract** **void** eat();

}

可以看出在抽象类中，可以有自己的成员变量，也可以有自己的成员函数，还可以定义抽象方法，但是定义的抽象方法不能有函数体（已验证）

包含抽象方法的类必须定义为抽象类（已验证）

注意：不包含任何抽象方法的java类也可以声明为抽象类

**子类必须实现父类中的所有抽象方法，否则子类也只能声明微抽象类，因为它们是不完备的（已验证）**

归纳一下，抽象类主要是通过继承，再由其子类发挥作用的，其作用包括两方面：

1. 代码重用---子类可以重用抽象父类中的属性和非抽象方法
2. 规划---子类中通过抽象方法的重写来实现父类规划的功能

如果方法形参声明微抽象类类型，则调用时实参引用的一定是其子类的对象，再结合多态性的虚方法调用机制，最终执行的也是其子类中实现了的方法。换句话说，这样的方法可以处理该抽象类的任何子类的对象。

17 接口

接口中定义的属性必须是也一定是**public** static final的，而接口中定义的方法则必须是public abstract，因此这些修饰符可以部分或全部省略

可以通过接口实现不相关类的相同行为，而不需要考虑这些类之间的层次关系

接口和java类最大的区别在于，接口可以多重实现，即一个java类可同时实现多个接口

**public** **interface** Runner {

**public** **void** run();

}

**public** **interface** Swimmer {

**public** **void** swim();

}

**public** **abstract** **class** Animal {

**public** **abstract** **void** eat();

}

**public** **class** Person **extends** Animal **implements** Runner, Swimmer {

**public** **void** run(){

System.*out*.println("I'm running");

}

**public** **void** swim(){

System.*out*.println("I'm swimming");

}

**public** **void** eat(){

System.*out*.println("I'm eating");

}

}

这种接口的多重实现机制看似多重继承，但饼不会导致将来调用方法时追溯上的不确定性，因为无论一个类同时实现多少个接口，由于接口中定义的一定是抽象方法，而且接口不能被实例化，所以真正执行的就只能是其实现类中重写后的方法

**public** **void** display(){

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

}

在接口的多重实现过程中，如果一个类同时实现的多个接口中定义了同名的属性，泽在调用时可以通过接口名对属性进行访问，如上述代码

**接口的多重实现与接口的多重继承**

接口之间可以进行继承，也就是说可以定义新的接口继承现有接口，添加新的常量属性和抽象方法定义，在其父接口的基础上进一步深化或分化其“规划”作用，当然最终还是要靠其实现类实现所有规划的功能

**public** **interface** extendInterface **extends** Runner, Swimmer{

**public** **void** show();

}

18 嵌套类

Java语言支持类的嵌套定义，及允许将一个类定义在其他类的内部，其中内层的类被称为嵌套类

Public class A{

Private class B{}

Private static class C{}

}

和其他属性及方法一样，嵌套类也属于其所在的外层类的成员，因此可以将嵌套类声明为static或非static，当我们只是一次性地使用一个java类时，可以考虑将之声明微嵌套类

和普通的外层类不同，内部类与其所在的外层类之间存在着逻辑上的隶属关系，或者说依赖关系--内部类的对象不能单独存在，它必须依赖一个其外层类的对象；作为这种没有“独立性”的“回报”，在内部内中可以直接访问其外层类中的成员，包括属性和方法，即使这些属性和方法是private的

匿名内部类：

Animal p = **new** Animal(){

**public** **void** eat()

{

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

}

};

p.eat();//输出IIIII

其中Animal是抽象类，有一个抽象方法为eat

只有顶层类或静态嵌套类中才可以定义static成员，而飞静态的内部类中则不允许

19 枚举类型

Week k = Week.*FRI*;

**if**(k.equals(Week.*MON*))

System.*out*.println("今天是星期一");

**else**

System.*out*.println("今天不是星期一");

20 异常处理

（1）链接错误

指的是程序链接错误，最常见的就是后一个类的.class文件被误删除而导致前一个类无法正常工作

Exception in thread "main" java.lang.NoClassDefFoundError: A

其中NoClassDefFoundError类是LinkageError的子类，当JVM动态链接（即加载）所需Java类却找不到该类的定义（.class）时，则出现此异常

（2）虚拟机错误

当java虚拟机崩溃或用尽了它继续操作所需的资源时，会跑出改错误，其中较有代表性的一种是StackOverflowErroe，当应用程序递归太深而导致栈内存溢出会出现此错误

Exception in thread "main" java.lang.StackOverflowError

（3）空指针异常

对于空引用，试图访问其成员的话将出现空指针异常

Exception in thread "main" java.lang.NullPointerException

（4）输入/输出异常

和先前的异常不同，只要是又可能出现IOException的Java代码，在编译时就会出错，而不会等到运行时才发生

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

Unhandled exception type FileNotFoundException

Java异常处理机制

String[] friends = {"Lisa", "Billy", "Kessy"};

**try**{

**for**(**int** i = 0; i < 5; i++)

System.*out*.println(friends[i]);

}

**catch**(ArrayIndexOutOfBoundsException e){

System.*out*.println("index error");

}

**finally**{

System.*out*.println("in finally block");

}

Lisa

Billy

Kessy

index error

in finally block

其中的finally语句块中的代码是无条件执行的，咋一看上去，好像有没有finally都一样，但是当catch中有了return就不一样了，即使有了return，finally中的语句仍然会执行

对于异常信息可以通过父类的Throwable定义的两个成员方法来获得错误信息

getMessage方法与printStackTrace方法

**public** **class** A {

**public** **void** work(**int**[] a){

String s = **this**.contain(a, 3);

System.*out*.println("Results: " + s);

}

//用于寻找source数组中是否存在dest

**public** String contain(**int**[] source, **int** dest){

String results = "no";

**try**{

**for**(**int** i = 0; i < source.length; i++)

{

**if**(source[i] == dest)

{

results = "yes";

**break**;

}

}

}**catch**(Exception e){

System.*out*.println("Exception Message: " + e.getMessage());

System.*out*.println("Stack Trace: ");

e.printStackTrace();

results = "error";

}

**finally**{

**return** results;

}

}

}

输出信息：

Exception Message: null

Stack Trace:

java.lang.NullPointerException

Results: error

at A.contain(A.java:15)

at A.work(A.java:5)

at TestVirtualMachineError.main(TestVirtualMachineError.java:35)

一个IO异常的例子

**try**

{

FileInputStream in = **new** FileInputStream("myfile1.txt");

**int** b = in.read();

**while**(b != -1)

{

System.*out*.println((**char**)b);

b = in.read();

}

}**catch**(FileNotFoundException e){

System.*out*.println("File not found");

}**catch**(IOException e){

e.printStackTrace();

}

System.*out*.println("It's OK");

输出：

File not found

It's OK

关于FileInputStream的功能：

从本地文件系统的某个文件中字节为单位读取数据到程序中，其read（）方法的作用是从当前对象（调用改方法的对象）所关联到的数据源，比如文件或键盘输入中，读取下一个字节数据，然后将其左侧（高位）补24个0，转换成取值范围0~255的int型整数返回，如果已到达数据源的结尾再没有可用的字节，则返回数值-1，

**Java中的第二种处理异常的方法，声明异常类型**

如果一个方法中的代码在运行时可能生成某种异常，但是在本方法中不必要，或者不能确定如何处理此类异常时，则可以声明抛弃该异常，表明改方法中将不对此类型的异常进行处理，而由改方法的调用者负责处理，泽系统将在调用改方法的上层方法体内处寻找合适的异常处理代码，而不再继续执行改方法的正常处理流程。

**public** **void** readFile() **throws** IOException

{

FileInputStream in = **new** FileInputStream("my.txt");

**int** b = in.read();

**while**(b != -1)

{

System.*out*.println((**char**)b);

b = in.read();

}

}

在主函数中一定要这样定义：

TestVirtualMachineError test = **new** TestVirtualMachineError();

**try**{

test.readFile();

}**catch**(IOException e){

System.*out*.println(e);

}

Java在重写方法在声明抛出异常方面受到限制，及重写方法不允许抛出比被重写方法范围更大的异常类型，及重写方法只能抛出本身及其异常派生类

**Java中的第三种处理异常的方法，人工抛出异常**

Java异常类对象除了在程序运行出错时由系统自动生成并抛出之外，也可以根据需要人工创建并抛出，此时首先要人工创建一个异常类的对象，然后通过throw语句实现抛出操作，即将该异常对象提交给java运行环境

Exception me = new Exception(“年龄超出合适范围|”)

Throw me；

21 断言

判断eclipse断言是否开启

**boolean** isOpen = **false**;

**assert** isOpen = **true**;

System.*out*.println(isOpen);

未开启的状态下输出：false（即直接忽略）

开启的情况下输出：true

如何在eclipse中设置断言：

Run--run configurations—-Arguments--VM arguments

在对话框冲输入-ea（开启断言） -da（禁止断言）

**Assert -1 > 0;**

Exception in thread "main" java.lang.AssertionError

at TestVirtualMachineError.main(TestVirtualMachineError.java:5)

22 常用的java类型

（1）hashCode方法

哈希码可以理解微系统微每个java对象自动创建的整型编号，任何两个不同的java对象的哈希码一定不同

（2）toString方法

Object类中toString方法的原始定义如下：

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

例子：

Person person = **new** Person(20);

System.*out*.println(person.toString());

如果不重写toString方法则输出：Person@1fb8ee3

重写了toString方法则输出：This is an instance of Person, age = 20

（3）equals方法

Public boolean equals(Object obj){

Return (this == obj)

}

其功能相当于比较当前对象和方法参数obj所引用的对象两者的等价性，如果等价泽伟true，否则返回false。首先要明白java语言中的等价性标准：基本类型数据比较的是数据的值，而引用类型数据比较的则是对象的句柄，即对象的hashCode编码或者说引用类型变量的值，而非对象本身。简单地说，比较的永远是变量的值是否相等

对于一般的类而言，在没有重写equals方法时和==的结果相同，但是对于一些特殊的类比如String、File、Date以及封装类，它只是比较对象的内容而已，因为重写了

注意：String常量相同的话，在内容中将只保存一份

String s1 = "abc";

String s2 = "abc";//此时s1和s2指向同一个对象

System.*out*.println(s1 == s2);//输出为true

（4）finalize方法

Java运行时环境的垃圾收集器在销毁一个对象之前，会自动调用改对象的finalize方法，然后才释放对象的内存空间

（5）clone方法

23 字符串相关类型

（1）String类

Public String(byte[] bytes)

使用当前平台的默认字符集（所在操作系统中文件系统的默认字符集）解码制定的字节数组，并将解析出得字符依原来顺序构造一个新的String对象

**byte**[] b = {97, 98, 99, 100, 101, 102, 103};

System.*out*.println(**new** String(b));//输出阿abcdefg

Public String(byte[] bytes, int offset, int length)

使用当前平台默认字符集解码字节数组bytes中从下表offset（包含该下标）开始的后续所有数组元素，并将解析出得字符依原来顺序构造一个新的String对象

**byte**[] b = {97, 98, 99, 100, 101, 102, 103};

System.*out*.println(**new** String(b,3, 2));//输出阿de

如果length超过了bytes的索引则会报错

String s = **new** String("abcdefg");

//s[2] = 'l';//出错

Replace函数

String s = **new** String("abcdefg");

String s1 = s.replace("ab", "aaa");

System.*out*.println(s);//输出abcdefg

System.*out*.println(s1);//输出aaacdefg

Substring函数

String s1 = s.substring(3);

System.*out*.println(s1);//输出defg

String s2 = s.substring(2, 5);

System.*out*.println(s2);//输出cde

String s1 = s.toLowerCase();

String s2 = s.toUpperCase();

String s3 = s.trim();

charAt函数，返回指定下标出的字符

endsWith函数，判断当前字符串是否以指定的子串结尾

startsWith函数，平【按段当前字符串是否以指定的子串开头

indexOf函数，查找指定的字符或子字符串在当前字符串中出现的位置

lastindexOf函数，查找指定的字符串或子字符串在当前字符串中最后一次出现的位置

String s1 = **new** String("abc");

String s2 = **new** String("ABc");

System.*out*.println(s1.equalsIgnoreCase(s2));//输出为true

compareTo函数，按照字典顺序比较两个字符串的大小

（2）StringBuffer类

在实际开发过程中，我们经常需要对字符串的内容进行修改，使用String类型不是不可以，但是效率不高（因其对象一经创建不可改变，就只能不断地创建新对象并销毁旧的对象），这种情况下使用java.lang.StringBuffer就比较合适了，该类表示内容可以修改的Unicode编码字符序列，其对象创建之后，所保存的字符串内容和长度均可以修改，实际上每个StringBuffer对象都拥有一个可变容量的字符串缓冲区，该缓冲区的容量（缓冲区占用的内存空间大小，或者说可以容纳的字符数）可以随着内容的增加自动扩充，也可以直接设定

StringBuffer s = **new** StringBuffer("abcdefg");

s.append("hijkl");

System.*out*.println(s);//输出abcdefghijkl

s.insert(3, "xy");

System.*out*.println(s);//输出abcxydefghijkl

s.reverse();

System.*out*.println(s);//输出lkjihgfedyxcba

s.setCharAt(3, '5');

System.*out*.println(s);//输出lkj5hgfedyxcba

（3）StringBuilder类

StringBuilder比StringBuffer性能要好一点，但是它在多线程中是不安全的，即不保证多线程的安全性

String s = **new** String("ab cd ef");

String[] s1 = s.split(" ");

**for**(**int** i = 0; i < s1.length; i++)

System.*out*.println(s1[i]);

**for**(String each: s1)

System.*out*.println(each);

（4）StringTokenizer类

改类的功能是将当前字符串按照默认或指定的分隔符（即分隔标记）分解为多个片段，和上面的String的split功能一样

24 封装类

为了补救基本数据类型在面向对象方面的欠缺，java语言中引入了封装--针对各种基本数据类型均提供了相应的引用数据类型

Boolean Boolean

byte Byte

short Short

int Integer

Integer in = 90;

String s = in.toString();

System.*out*.println(s);//输出为90

比如说Integer类型中有一些静态方法

Public static String toBinaryString（int i）；

Public static intparseInt（String s）throw NumberFormatException

String s = **new** String("456s");

**try**

{

**int** no = Integer.*parseInt*(s);

System.*out*.println(no);

}

**catch**(NumberFormatException e)

{

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

}

注意parseInt函数在实际应用开发中尤为常用

而且String类中的静态方法可以实现和Integer类的静态方法toString同样的功能

**int** a = 100;

String b = String.*valueOf*(a);

System.*out*.println(b);

封装类均被定义为final的，因此不能被继承。封装类的对象一经创建，其内容不能改变，及其所封装的基本类型数据为只读得

**一个小总结：**

int -- String String.valueOf(int a);

String -- int Integer.parseInt(String a);

Integer -- String Integer.toString(int a);

25 日期相关类型

（1）Date类

（2）Calender类

Calender类是一个抽象类，可以通过调用其静态方法getInstance来获得该类的实例（实际上是其子类的实例），而获取和设置特定时间的值泽通过get和set方法实现

**public** **void** display(Calendar date)

{

String s = date.get(Calendar.*YEAR*) + "-" + (date.get(Calendar.*MONDAY*) + 1) + "-" + date.get(Calendar.*DATE*)+ " " + date.get(Calendar.*HOUR\_OF\_DAY*) + ":"+ date.get(Calendar.*MINUTE*) + ":" + date.get(Calendar.*SECOND*)+ " " + (date.get(Calendar.*AM\_PM*) == 0 ? "上午" : "下午");

System.*out*.println(s);

}

主函数调用：

Calendar c = Calendar.*getInstance*();

test.display(c);

输出：2011-11-26 16:26:22 下午

注意：月份要+1，小时如果要是24小时制则必须采用Calendar.HOUR\_OF\_DAY

要得到星期要-1

System.*out*.println(date.get(Calendar.*DAY\_OF\_WEEK*) - 1);

（3）Locale类

Locale对象主要包含两方面的信息--地区名称和语言种类，当程序要用到一些“语言环境敏感”的操作时，就会用到Locale为用户量身订制信息

（4）SimpledateFormat

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy/MM/dd hh:mm:ss");

Date date = **new** Date();

String str = sdf.format(date);

System.*out*.println(str);//输出2011/11/26 07:18:03

生成随机数

System.*out*.println(Math.*random*() \* 100 + 1);

大整数类型BigInteger/BigDecimal

public BigInteger(String val)

将大整数的饿十进制字符串表示形式转换为BigInteger实例。该字符串表示形式包括一个可选的减号，后跟一个或多个十进制数字序列

Public static BigInteger valueOf(long val)

返回其值等于指定long的值的大整数/BigInteger实例

Public BigInteger remainder(BigInteger val)

实现两个大整数的取模/求余操作

Public BigInteger pow(int exponent)

进行当前大整数值的指定次幂运算

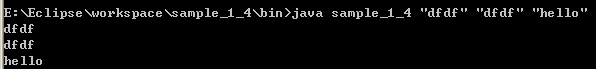
BigInteger a = BigInteger.*valueOf*(43);

BigInteger b = **new** BigInteger("12345678901234560000");

BigInteger c = a.add(b);

System.*out*.println(c);//输出12345678901234560043

26 命令行参数



27 标准输入输出

Java.lang.System类的三个静态类成员提供了有关I/O操作功能

Public static final PrintStream out；

Public static final InputStream in；

Public static final PrintStream err；

可以采用Scanner类输入的便捷方式

采用标准的输入传统方式

String s;

InputStreamReader isr = **new** InputStreamReader(System.*in*);

BufferedReader br = **new** BufferedReader(isr);

**try**{

s = br.readLine();

**while**(!s.equals(""))

{

System.*out*.println("Read: " + s);

s = br.readLine();

}

}**catch**(IOException e){

e.printStackTrace();

}

System.in提供了以字节读入的方式，将其封装成为InputStreamReader以字符读入方式，然后再封装成为BufferedReader以字符串读入方式，以换行符为分隔

这种做法称为IO流的封装和连接

注意：当在键盘上空回车时，BufferedReader的readLine方法接受到的不是null，而是一个长度为零的字符串“”

28 文件操作

（1）File类

File f1 = **new** File("a.jpg");

System.*out*.println("Name = " + f1.getName());//Name = a.jpg

System.*out*.println("Path = " + f1.getPath());//Path = a.jpg

System.*out*.println("AbsolutePath = " + f1.getAbsolutePath());//AbsolutePath=E:\Eclipse\workspace\sample\_1\_4\

a.jpg

System.*out*.println("Parent = " + f1.getParent());//Parent = null

System.*out*.println("Length = " + f1.length());//Length = 0

System.*out*.println("exists = " + f1.exists());//exists = false System.*out*.println("canRead = " + f1.canRead());//canRead = false

System.*out*.println("canWrite = " + f1.canWrite());//canWrite = false

System.*out*.println("isFile = " + f1.isFile());//isFile = false

System.*out*.println("isHidden = " + f1.isHidden());//isHidden = false

File newFile = **new** File("a.txt");

File moveFile = **new** File("D:\\hello.txt");

**try**{

**boolean** bool = newFile.createNewFile();

**if**(bool){

System.*out*.println("创建成功!");

}

newFile.renameTo(moveFile);//移动并重命名为moveFile文件

}**catch**(IOException e){

e.printStackTrace();

}

System.in FileInputStream 字节

InputStreamReader FileEeader FileWrite 字符

BufferedReader BufferedReader PrintWriter 一行

关于文件的读入

Scanner scan = **new** Scanner(System.*in*);

String fileName = scan.next();

File f = **new** File(fileName);

**try**{

FileReader fr = **new** FileReader(f);

BufferedReader br = **new** BufferedReader(fr);

String s = br.readLine();

**while**(s != **null**)

{

System.*out*.println("Read: " + s);

s = br.readLine();

}

}**catch**(FileNotFoundException e){

System.*out*.println("Can't found the file!");

} **catch**(IOException e){

e.printStackTrace();

}输出：

Read: 无语

Read: hello

Read: 你是谁

注意：Filereader类的构造方法被重载过，也接受直接以字符串形式(String类型的参数)给出的文件名

FileReader fr = **new** FileReader("hello.txt");

注意：使用BufferedReader的readLine函数读文件时，遇到文件的结尾则返回null而不是””，这与先前读取键盘输入遇到空回车时返回空字符串””的情况有所不同

关于文件的写入

String[] str = {"hello", "wuqianhu"};

**try**

{

PrintWriter out = **new** PrintWriter(**new** FileWriter("a.txt"));

**for**(**int** i = 0; i < str.length; i++)

out.println(str[i]);

out.close();//切记一定要关闭，否则就不能写入文件

}**catch**(IOException e){

e.printStackTrace();

}

文件过滤

在文件操作中可能会涉及到文件过滤，即只检索和处理符合特定条件的文件，使用java中提供的java.io.FileFilter接口可以实现，其接口中只定义了一个抽象方法：

Boolean accept(File pathname)

测试参数指定的File对象对应的文件（子目录）是否应保留在文件列表中，即不被过滤掉

在实际应用中可以定义一个该接口的实现类，重写其中的accept方法，在方法体重加入所需的文件过滤逻辑，然后创建一个该实现类的对象作为参数传递给File对象的文件列表方法list就可以了，在list方法执行过程中会自动调用前者的accept方法来过滤文件

**public** **class** MyFilter **implements** FileFilter {

**private** String extension;

**public** MyFilter(String extension){

**this**.extension = extension;

}

**public** **boolean** accept(File file){

**if**(file.isDirectory())**//是否是目录，目录也当做文件处理**

**return** **false**;

String fileName = file.getName();

**int** index = fileName.lastIndexOf(".");

**if**(index == -1 || index == (fileName.length()) - 1)

**return** **false**;

**return** fileName.substring(index).equals(extension);

}

}

主函数调用：

**public** **void** dir(String path, String extension)

{

File directory = **new** File(path);

MyFilter filter = **new** MyFilter(extension);

File[] files = directory.listFiles(filter);

System.*out*.println("路径：" + path);

**for**(File file: files)

System.*out*.println(file.getName());

}

29 可变参数方法

可变参数的语法格式是在参数类型后加“...“，需要注意的是，可变长度参数必须放在参数列表的最后，而且一个方法最多只恩那个包含一个可变长度参数

**public** **void** myPrint(String s, **int** i, Object... objs)

{

System.*out*.println(s.toUpperCase());

System.*out*.println(i \* 100);

**for**(Object o: objs)

System.*out*.println(o);

}

30 注解（内置注解类型）

@override其作用是指明被注解的方法重写/覆盖了父类的方法，因此该类型注解只能用于方法声明

**public** **class** Person {

**private** String name;

**public** **void** setName(String name){

**this**.name = name;

}

**public** String getName(){

**return** name;

}

@Override

**public** String toString(){

**return** "Name; " + name;

}

}

这里使用Override注解能起到辅助检查的功能

@Deprecated注解用来标记过时的API

@SuppressWarnings注解的作用是关闭/阻止编译时的提示和警告信息

31 归档工具jar

32 集合与映射

Java集合中只能保存引用类型的数据，实际上存放的是对象的引用，而飞对象本身，这和引用类型元素组成的一维数组的情形一样，集合中元素也相当于引用类型变量

（1）集合Set

Set集合中不区分元素的顺序（当然也就不记录元素的加入顺序）

Set集合中不允许出现重复元素，这里所谓的“重复元素“是指按照equals方法判断为等价的对象

HashSet类

它不允许出现重复元素，不保证集合中元素的顺序

TreeSet类

是一个可以实现排序功能的集合

使用通过实现Comparable接口来实现排序的规则

@Override

**public** **int** compareTo(Object o){

Person p = (Person)o;

**return** **this**.id - p.id;

}

@Override

**public** **boolean** equals(Object o){

**if**(o **instanceof** Person){

**if**(**this**.id == ((Person)o).id)

**return** **true**;

}

**return** **false**;

}

需要注意的是，用户在自己定义的类型中重写自然排序方法compareTo()给出比较逻辑时，一定要确保其与等价性判断方法equals()保持一致，即确保(x.compareTo(y)==0)==(x.equals(y))永远成立，也就是说如果compareTo方法比较连个对象x和y是“平等的”，则使用equals方法比较它们也是相等的。

（2）列表List

ArrayList类

实现了List接口，用于表述长度（元素数目）可变的数据列表，ArrayList实例是通过内部封装一个Object数组来存储列表元素的，该数组的长度称为ArrayList的容量，可以在创建ArrayList列表对象时荣国构造方法参数显式指定其初始容量，如不指定则默认设置容量为10，对着想ArrayList中不断添加元素，元素数目超过其列表容量时，其容量也自动增长，实际上市闯将一个新的长度增1的Object数组，然后再进行批量拷贝

ArrayList h = **new** ArrayList();

System.*out*.println(h.size());

h.add("1st");

h.add("2nd");

h.add(**new** Integer(3));

h.add(**new** Double(4.0));

h.add("2nd");

System.*out*.println(h);//输出[1st, 2nd, 3, 4.0, 2nd]

h.set(3, "replace");

**for**(**int** i = 0; i < h.size(); i++)

System.*out*.print(h.get(i) + " ");//输出1st 2nd 3 replace 2nd

Integer it = (Integer)h.get(2);

System.*out*.println(it.intValue());//输出为3

Object[] os = h.toArray();//转换为数组

**for**(Object o: os)

System.*out*.print(o + " ");//输出为1st 2nd 3 replace 2nd

Vector类

Vector和ArrayList的用法几乎完全相同，而这最主要的差别在于，Vector类是同步（线程安全）的，运行效率要低一些，主要用在多线程环境中，而ArrayList则是不同步的，更适合在单线程编程中使用

Stack类

后进先出

Push pop

Peek方法：查看/返回栈顶对象，但不从栈中移除它

（3）映射（Map）

Java中提供了多种map结构

1 HashMap结构

HashMap不保证其中元素的先后顺序，并且允许使用null和null键值对

HashMap hm = **new** HashMap();

hm.put(1001, **new** Person("lisi", 20));

hm.put(1001, **new** Person("zhangsan", 20));

hm.put(1003, **new** Person("wangwu", 20));

hm.put(1004, **new** Person("zhaoliu", 20));

Person p = (Person)hm.get(1001);

System.*out*.println(p);//这里会有覆盖现象

//获得所有键值集合

Set names = hm.keySet();

**for**(Object o: names)

System.*out*.println(o);

//获得所有值的集合

Collection values = hm.values();

**for**(Object o: values)

System.*out*.println(o);

2 TreeMap

TreeMap实现了将Map映射中的元素按照“键”进行升序排列的功能

（4）迭代器

提供统一方式供各种集合元素进行遍历。迭代器只提供了几个函数

Boolean hasNext（）

Object next（）

Void remove（）

ArrayList a = **new** ArrayList();

a.add("China");

a.add("USA");

a.add("Korea");

Iterator iter = a.iterator();

**while**(iter.hasNext()){

String country = (String)iter.next();

System.*out*.println(country);

}

Collections类

Collections类中定义了多种集合操作方法，实现了对元素的排序、取极值、批量拷贝、集合结构转换、循环移位以及匹配性检查等功能，所有这些方法都是static的

ArrayList alist = **new** ArrayList();

alist.add(75);

alist.add(38);

alist.add(21);

alist.add(4);

alist.add(12);

System.*out*.println("原列表：" + alist);

Collections.*sort*(alist);

System.*out*.println("排序后：" + alist);

Collections.*reverse*(alist);

System.*out*.println("逆序后：" + alist);

Collections.*shuffle*(alist);

System.*out*.println("洗牌后：" + alist);

Collections.*rotate*(alist, 2);

System.*out*.println("移位后：" + alist);

ArrayList blist = **new** ArrayList();

blist.add(55);

blist.add(66);

System.*out*.println("新列表：" + blist);

Collections.*copy*(alist, blist);

System.*out*.println("拷贝后：" + alist);//只是出现了覆盖

System.*out*.println("列表中出现元素66的次数为：" + Collections.*frequency*(alist, 66));

System.*out*.println("列表中最大值为：" + Collections.*max*(alist));

System.*out*.println("列表中最小值为：" + Collections.*min*(alist));

Array类

和Collections类中的方法类似，灵活运用

Java中的抽象类与接口

在抽象类的方法中，可以有自己的数据成员，也可以有非abstract的成员方法，而在interface方法的实现中，只能够有静态的不能被修改的数据成员，也就是必须是static final，不过在interface中一般不定义数据成员，所有的成员方法都是abstract的，从某种意义上来说，interface是一种特殊形式的abstract class

从编程角度上来说，abstract class和interface的用法有一定的区别：

首先，abstract class在java语言中表示的是一种继承关系，一个类只能使用一次继承关系，但是，一个类却可以实现多个interface

其次，在abstract class的定义中，我们可以赋予方法默认行为，但是在interface的定义中，方法却不能拥有默认行为，为了绕过这个限制，必须使用委托，但是这会增加一些复杂性，有时会造成很大的麻烦。

另外注意到，在抽象类中是可以实例化接口的，而接口中不能实例化抽象类，在其他类中也不能实例化接口。

GUI编程