Advanced Language Features

知识点：一. static修饰符

static修饰符可以用来修饰类的成员变量、成员方法和代码块。

. 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。

1. static 变量

成员变量：定义在类里面、方法外面的变量, 分两种：

a. 实例变量;

b. 静态变量；形式和实例变量类似，在实例变量前面加static关键字；

static变量和实例变量的区别：

. static变量对于每个类而言在内存中只有一个，能被类的所有实例所共享；实例变量对于每个类的每个实例都有一份，它们之间互不影响；

. 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变量可以直接通过类名访问，实例变量通过引用类型变量访问；

举例： public class Counter {

public int count1 = 0;

public static int count2 = 0;

public static void main(String[] args) {

Counter counterA = new Counter();

Counter counterB = new Counter();

counterA.count1++;

counterA.count2++;

counterB.count1++;

counterB.count2++;

}

}

练习：统计一个类创建实例的个数；

2. static 方法

成员方法分为静态方法和实例方法。用static修饰的方法叫静态方法，或类方法。静态方法也和静态变量一样，不需要创建类的实例，可以直接通过类名来访问。

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方法可以直接访问所属类的实例变量和实例方法，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；

2) 不能使用super关键字,super关键字用来访问当前实例从父类中继承的方法和属性。super关键字与类的特定实例相关；

3) 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

}

c. 父类的非静态方法不能被子类覆盖为静态方法；

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。(思考:什么时候JVM对一个类进行类加载)

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);

}

}

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。

//Student s = new Student();

new一个对象的时候JVM都做了那些事情:

1.之前没有进行类加载

1.类加载,同时初始化类中静态的属性(赋默认值)

2.执行静态代码块

3.分配内存空间,同时初始化非静态的属性(赋默认值)

4.调用父类构造器

5.父类构造器执行完后,如果自己声明属性的同时有显示的赋值,那么进行显示赋值把默认值覆盖

6.执行匿名代码块

7.执行构造器

8.返回内存地址

例子:

package com.briup.ch06;

public class Test {

public static void main(String[] args) {

A a = new B();

}

}

class A{

protected String name = "lisi";

public A() {

System.out.println("父类构造器A");

System.out.println("父类构造器A中调用test方法开始,由于子类重写过test方法所以这里执行子类的test方法");

test();

System.out.println("父类构造器A中调用test方法结束");

}

public void test(){

}

}

class B extends A{

private String name = "tom";

{

System.out.println("子类匿名代码块中:"+name);

}

public B() {

System.out.println("子类构造器B");

}

public void test(){

System.out.println("test方法中:this.name="+this.name);

System.out.println("test方法中:super.name="+super.name);

}

}

打印结果::

父类构造器A //子类构造器调用父类构造器

父类构造器A中调用test方法开始,由于子类重写过test方法所以这里执行子类的test方法

test方法中:this.name=null //这个时候父类构造器还没有执行完 所以子类中的属性不会显示赋值 所以只有初始的默认值null

test方法中:super.name=lisi //这个时候父类构造器开始调用 了 所以父类中的属性已经有了显示赋的值了

父类构造器A中调用test方法结束

子类匿名代码块中:tom //这个时候父类构造器已经调用结束 所以子类中的属性已经有了显示赋的值了

子类构造器B

结论: 子类中的属性的显示赋值的时机 是在 父类构造器执行完之后和子类的匿名代码块执行之前的某个时候

2.之前已经进行了类加载

1.分配内存空间,同时初始化非静态的属性(赋默认值)

2.调用父类构造器

3.父类构造器执行完后,如果自己声明属性的同时有显示的赋值,那么进行显示赋值把默认值覆盖

4.执行匿名代码块

5.执行构造器

6.返回内存地址

练习例子:StaticTest.java StaticTest2.java

4. 静态导入

静态导入也是JDK5.0引入的新特性。

要使用静态成员（方法和变量）我们必须给出提供这个静态成员的类。使用静态导入可以使被导入类的静态变量和静态方法在当前类中可以直接使用，使用这些静态成员无需再在前面写上他们所属的类名。

//例如:

import static java.lang.Math.random;

import static java.lang.Math.PI;;

public class Test {

public static void main(String[] args) {

//之前是需要Math.random()调用的

System.out.println(random());

System.out.println(PI);

}

}

二. final修改符

final具有"不可改变的"含义，它可以修饰非抽象类、非抽象成员方法和变量。

. 用final修饰的类不能被继承，没有子类；

. 用final修饰的方法不能被子类的方法覆盖；

. 用final修饰的变量表示常量，只能被赋一次值；

final不能用来修饰构造方法，因为"方法覆盖"这一概念仅适用于类的成员方法，而不适用于类的构造方法，父类的构造方法和子类的构造方法之间不存在覆盖关系. 因此用final修饰构造方法是无意义的。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方法，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：

final修饰的属性(成员变量)赋值的位置:

非静态的成员变量

1.声明的同时

2.匿名代码块

3.构造器(类中出现的所有构造器)

静态的成员变量

1.声明的同时

2.static代码块

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，定义变量时进行初始化或者static代码块中赋值；

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；

c. final变量只能赋一次值。

public class Sample {

private final int var1 = 1;

public Sample() {

var1 = 2; //编译出错，不允许改变var1实例变量的值；

}

public void method(final int param) {

final int var2 = 1;

var2++; //编译出错，不允许改变var2局部常量的值

param++; //编译出错，不允许改变final类型参数的值；

}

}

public class Sample {

final int var1; //定义var1实例常量

final int var2 = 0; //定义并初始化var2实例常量

Sample() {

var1 = 1; //初始化var1实例常量

}

Sample(int x) {

var1 = x; //初始化var1实例常量

}

}

练习 FinalTest.java

三. abstract修饰符

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，即不允许创建抽象类本身的实例。没有用abstract修饰的类称为具体类，具体类可以被实例化。

. 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。

没有abstract修饰的方法称为具体方法，具体方法具有方法体。

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；

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。

抽象类不允许实例化：思考原因?

练习 AbstractTest.java

四. 接口

接口使用的目的：解决多重继承问题；例如Fish类继承Animal类，表明Fish是一种动物，但鱼同样也是一种食物，如何表示这种关系呢？ 由于Java语言不支持一个类有多个直接的父类，因此无法用继承关系来描述鱼既是一种食物，又是一种动物，为了解决这一问题，Java语言引入接口类型，简称接口。一个类只能有一个直接的父类，但是可以实现多个接口。 采用这种方式，Java语言对多继承提供了有力的支持。

1. 接口是抽象类的另外一种形式

抽象类抽象到极致就是接口，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；

public interface A {

void method1(); //合法，默认为public、abstract类型

public abstract void method2();//合法，显示声明为public、abstract类型

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；

public interface A {

int CONST = 1; //合法，CONST默认为public, static, final类型

public static final int OPAQUE = 1; //合法，显示声明为public static final 类型

}

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。

6. 一个类只能继承一个直接的父类，但能实现多个接口。

抽象类和接口比较：

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。

练习:InterfaceTest.java InterfaceTest2.java

五. 访问控制

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 不同的包

公开级别： public y y y y

受保护 protected y y y

默认 没有访问控制符 y y

私有 private y

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；

顶层类只可以处于公开或默认访问级别；

注意:protected和default都有包访问权限(同包下可以访问)

六. 内部类

在一个类的内部定义的类称为内部类。

内部类分为：

成员内部类 静态内部类 局部内部类 匿名内部类

顶层类(正常类)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；

例子:StaticOutterClass.java

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：(相对应类中的一个静态变量)

静态内部类中访问不到外部类的非静态属性或者方法

静态内部类的对象不需要依赖于外部类的对象

内部类 变量名字 = new 内部类();

public class A {

public static class B{

private int v;

public void say(){

System.out.println("hello");

}

}

public static void main(String[] args){

B b = new B();

}

}

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 = A.this.a1; //不合法 静态内部类中不能访问外部对象的this

}

}

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;

v.v2 = 1;

A.B.v1 = 1; //编译错误

A.B.v2 = 1; //合法

A.B.C.v3 = 1; //合法

}

}

2. 成员内部类:(相当于类中的一个成员变量)

成员内部类中不能有static的声明属性或者方法

成员内部类可以由public protected default private修饰

成员内部类是依赖于外部类的对象而存在的

外部类.内部类 var = new 外部类().内部类();

例子:InstanceOutterClass.java

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 a1;

static int a1;

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 = A.this.a1; //访问类A的当前实例中的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(); //valid

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：(相当于一个方法中的局部变量)

局部内部类不能用public private等修饰符修饰

写在方法当中,而且只能在方法当中使用

可以访问外层类的普通成员变量和静态成员变量以及普通方法和静态方法,也可以访问该内部类所在方法当中的局部变量,但是这个局部变量必须是final修饰；

例子:LocalOutterClass.java

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(); //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:(和局部内部类很相似)

匿名内部类也是用的最多的内部类

可以写成成员变量的形式,也可以写在方法当中,一般写在方法当中较多

匿名内部类里可以访问外部类的普通属性和方法,已经静态属性和方法,如果要访问这个内部类所在方法中的局部变量,那么要求这个局部变量必须是final修饰的

匿名内部类里面没有构造函数,因为这个类没有名字,所以在其他地方不能用

例子:AnonymousOutterClass.java

public class Hello{

public void test(){

//假如A是同包下的一个接口，有一个抽象方法go

A a = new A(){

public void go(){

System.out.println("gogogo");

}

};

}

}

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；

匿名内部类：既可以在类的内部，方法的外部，也可以在方法的内部；

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；

匿名内部类：new 类名() {};

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；

匿名内部类：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；

匿名内部类：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. == 和 equals() 的区别

== :比较的是,值是不是相等

基本数据类型比较的是值,引用类型比较的是地址值

equals(Object o): Object类中的方法,所以,在每一个java类中,都会有这个方法,因为每一个java类都是直接或者间接的Object类的子类,会继承到这个方法

如果自己所写的类中已经重写了equals方法,那么就安装用户自定义的方式来比较俩个对象是否相等,如果没有重写过equal方法,那么会调用父类(Object)中的equals方法进行比较,也就是比较地址值

注意:equals(Object o)方法只能是一个对象来调用,然后参数也是要传一个对象的

所以下面是错误的写法:

int a = 1;

a.equals(1);

因为基本数据类型不是算是对象,不能调用方法

如果是基本数据类型那么就用==比较

如果是引用类型的话,想按照自己的方式去比较,就要重写这个类中的equals方法, 如果没有重写,那么equals和==比较的效果是一样的,都是比较引用的地址值

如果是比较字符串,那么直接用equals就可以了,因为String类里面已经重写了equals方法,比较的时候字符串的内容,而不是引用的地址值了

toString(): Object类中的方法,所以,在每一个java类中,都会有这个方法,因为每一个java类都是直接或者间接的Object类的子类,会继承到这个方法

当前用一个引用指向一个对象的时候,比如:Student s = new Student(),然后如果直接打印这个引用s,其实是调用了s.toString()方法,然后就会把这个引用里面的存放的堆区对象的地址值显示出来

所以我们会常常在类中重写这个toString()方法,然后让这个类的引用按照我们要求来返回内容。

getClass()：Object类中的方法,所以,在每一个java类中，都会有这个方法,并且这个方式final修饰的,不能被子类重写,这个方法可以返回某一个引用在运行的时候指向对象的类型

例如:Person p = new Student()

//会输出:class com.briup.chap06.Student

//说明这个引用p在运行时指向的是Student这个类的对象

//注意这个引用p的类型是Person的(多态)

System.out.println(p.getClass());

八. 基本数据类型对应的包装类型

boolean Boolean

byte Byte

short Short

char Character

int Integer

long Long

float Float

double Double