张帅豪 4.19号上机 18030100101

Ppt15

package exercise;

public class PassTest {

public void changeint(int value) {

value = value+1;

System.*out*.println("value="+value);

}

public static void main(String args[]) {

int val;

PassTest pt = new PassTest();

val=11;

pt.changeint(val);

System.*out*.println("Int value is:"+val);

}

}

运行结果

value=12

Int value is:11

我先按照原代码敲出来，val的结果并未改变，我之后修改changeint方法，加上一个value值的输出语句，运行结果中果然输出了value=55,之后我有修改value=55成value=value+1,这样就和传入的参数有了关系，输出结果为12.

我修改了方法

public class PassTest {

public static int changeint(int value) {

value = value+1;

return value;

}

主函数中System.*out*.println("参数传入对象内部有改变"+*changeint*(val));

刚开始我并没有将方法改为静态，发现有错误，后来将static加入，才可引用。

Ppt16

package exercise;

public class PassTest {

public void changestr(String value) {

value = new String ("different");

}

public static void main(String args[]) {

String str;

PassTest pt = new PassTest();

str = new String("hello");

pt.changestr(str);

System.*out*.println("Str value is:"+str);

// System.out.println("参数传入对象内部有改变"+changeint(val));

}

}

运行结果

Str value is:hello

16和15非常相似

修改方法

public void changestr(String value) {

value = new String ("different");

System.*out*.println("the inner of object "+value);

}

运行结果有the inner of object different

同样的，让方法返回一个字符串

package exercise;

public class PassTest {

public static String changestr(String value) {

value = new String ("different");

System.*out*.println("the inner of object "+value);

return value;

}

public static void main(String args[]) {

String str;

PassTest pt = new PassTest();

str = new String("hello");

pt.*changestr*(str);

System.*out*.println("Str value is:"+str);

System.*out*.println("参数传入对象内部有改变"+*changestr*(str));

}

}

运行结果为

the inner of object different

Str value is:hello

the inner of object different

参数传入对象内部有改变different

Ppt17

package exercise;

public class PassTest {

float ptValue;

public void changeObjectValue(PassTest ref) {

ref.ptValue=99.0f;

}

public static void main(String args[]) {

PassTest pt = new PassTest();

pt.ptValue=101.0f;

pt.changeObjectValue(pt);

System.*out*.println("Pt value is:"+pt.ptValue);

}

}

运行结果

Pt value is:99.0

但是如果我们把 pt.changeObjectValue(pt);删去，输出结果为101.0，这是因为主函数中 pt.changeObjectValue(pt);这句话将对象pt初始化，自然使用了changeObjectValue方法，这个方法改变了赋予对象pt中ptValue值即99.0。而我们 将 pt.changeObjectValue(pt);删去，就不会使用方法。我们可以在对象中加一个方法

public void love() {

System.*out*.println("love you forever");

}

若在主函数中不直接使用，运行结果中就没有love you forever 这句话。我们改变顺序

public void changeObjectValue(PassTest ref) {

ref.ptValue=99.0f;

}

float ptValue=11.0f;

但运行运行结果仍是99.0，这是因为对象中的运行顺序

加载类的实例   
加载类的静态变量   
加载类的静态方法   
实例化一个对象   
初始化时先初始化父类

若是我们再加一个方法

public class PassTest {

float ptValue=11.0f;

PassTest(){

ptValue=11.1f;

}

public void changeObjectValue(PassTest ref) {

ref.ptValue=99.0f;

}

public void change() {

ptValue=22.0f;

}

public void love() {

System.*out*.println("love you forever");

}

public static void main(String args[]) {

PassTest pt = new PassTest();

System.*out*.println("1 "+pt.ptValue);

pt.ptValue=101.0f;

pt.changeObjectValue(pt);

pt.change();

System.*out*.println("Pt value is:"+pt.ptValue);

}

}

运行结果为

1 11.1

Pt value is:22.0

Ppt26

package exercise;

public class DefaultConstructor {

public static void main(String[] args) {

Bird nc = new Bird();

}

}

class Bird {

int i;

}

运行结果并没有什么字符出现。

修改程序

public class DefaultConstructor {

public static void main(String[] args) {

Bird nc = new Bird();

nc.Beaf(3);

}

}

class Bird {

int i;

public void Beaf(int j){

i=j;

System.*out*.println("i="+i);

}

public Bird() {

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

}

}

运行结果为

qwe

i=3

所以在运行过程中，从调试中可以看出Bird nc = new Bird();后就跳转到public Bird() {

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

}

即构造方法，之后跳转到 nc.Beaf(3); 调用Bird对象的beaf方法

Ppt38

先创建两个包pkg1和pkg2

在pkg1中

package pkg1;

public class A {

public A(){

System.*out*.println("A's constructor");

}

void func(){

System.*out*.println("A's method");

}

}

class B{

public B(){

System.*out*.println("B's constructor");

}

}

在pkg2中

package pkg2;

import pkg1.\*;

public class PublicVsPackage {

public static void main(String[] args){

A obj=new A();

}

}

运行pkg2包中的PublicVsPackage类，运行结果为

A's constructor

由程序知，PublicVsPackage引用pkg1包，主函数初始化A的实例化对象obj，从而运行构造方法，即有A's constructor

修改程序

Pkg2

package pkg2;

import pkg1.\*;

public class PublicVsPackage {

public static void main(String[] args){

A obj=new A();

C obj2=new C();

obj.func();

}

}

Pkg1

A类

public class A {

public A(){

System.*out*.println("A's constructor");

B nc=new B();

}

public void func(){

System.*out*.println("A's method");

}

}

class B{

public B(){

System.*out*.println("B's constructor");

}

}

C类

public class C {

public C(){

System.*out*.println("C's constructor");

}

}

运行结果

A's constructor

B's constructor

C's constructor

A's method

从中可知在调用不同包的类时，需要在class前加public

若是pkg1包中有很多类，而pkg2只需要引用A、C类，那么就将

import pkg1.\*;

改为

import pkg1.A;

import pkg1.C;

ppt40

源程序

package pkg2;

public class PrivateVsPackage {

public static void main(String[] args){

C obj=new C();

obj.func2();

}

}

class C{

private void func1(){

System.*out*.println("C's method 1");

}

void func2(){

System.*out*.println("C's method 2");

this.func1();

}

}

运行结果

C's method 2

C's method 1

虽然在同一个类中，但是PrivateVsPackage和C会形成两个文件，源程序中是从func2中引用func1，他们两个属于一个类，这时的private可引用。但若是在主函数中加入

obj.func1();

会出现错误，因为不同类private无法引用

Ppt42

因为原程序没有输出结果，所以加了obj.fun();在主函数里，又因为无法引用c类的protected，因为不在一个包里，所以将c中的protected改为public

源程序

Pkg2

**package** pkg2;

**import** pkg1.C;

**public** **class** ProtectedVsPackageAndPublic {

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

C obj=**new** C();

obj.func();

}

}

**class** CSub **extends** C{

**void** mtd(C parent, CSub sub){

func();

sub.func();

}

}

Pkg1中

**package** pkg1;

**public** **class** C {

**public** **void** func(){

System.***out***.println("Protected method of C");

}

}

运行结果为

Protected method of C

但因为CSub是c的子类，子类可以引用父类。

Ppt51

源程序

**package** pkg2;

**public** **class** OrderOfInit {

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

House h = **new** House();

h.f();

}

}

**class** Window {

Window(**int** m) {

System.***out***.println("window " + m);

}

}

**class** House {

Window w1 = **new** Window(1);

House() {

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

w3 = **new** Window(33);

}

Window w2 = **new** Window(2);

**void** f() {

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

}

Window w3 = **new** Window(3);

}

运行结果为

window 1

window 2

window 3

House

window 33

f()

分析运行步骤

House h = **new** House();

先是到构造方法的House() {，然后进行对象的初始化Window w1 = **new** Window(1);转到

**class** Window {

Window(**int** m) {

System.***out***.println("window " + m);

}

之后两个对象也实例化Window w2 = **new** Window(2); Window w3 = **new** Window(3);，结束之后，跳转到构造方法的System.***out***.println("House"); w3 = **new** Window(33);，最后，主函数的h.f();语句运行。

Ppt61

源代码

**package** pkg2;

**public** **class** TestInheritance {

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

Rectangle rect=**new** Rectangle();

rect.newDraw();

}

}

**class** Shape{

**public** **void** draw(){

System.***out***.println("Draw shape");

}

}

**class** Rectangle **extends** Shape{

**public** **void** draw(){

System.***out***.println("Draw Rectangle");

}

**public** **void** newDraw(){

draw();

**super**.draw();

}

}运行结果

Draw Rectangle

Draw shape

分析步骤，主函数中Rectangle rect=**new** Rectangle();初始化对象，并实例化，rect.newDraw();运行，跳转到rectangle对象的newdraw语句，因为rectangle对象是shape对象的子类，并重写了父类中的draw方法，所以，newdraw中的draw方法为重写后的方法，而super.draw（）为父类中的draw方法，所以运行结果为

Draw Rectangle

Draw shape

若在父类中加一个方法

**public** **void** write() {

System.***out***.println("i love you");

}

主函数中加一个write();，运行结果就发生了改变

Draw Rectangle

i love you

Draw shape

Ppt76

源程序

**package** pkg2;

**public** **class** NotOverriding **extends** Base {

**private** **int** i=2;

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

NotOverriding no=**new** NotOverriding();

no.increase();

System.***out***.println(no.i);

System.***out***.println(no.getI());

}

}

**class** Base{

**private** **int** i=100;

**public** **void** increase(){

**this**.i++;

}

**public** **int** getI(){

**return** **this**.i;

}

}

运行结果

2

101

分析运行步骤，主函数在NotOverriding类中，而NotOverriding是base类的子类，继承了base的方法，NotOverriding no=**new** NotOverriding();no.increase();中，运行了increase（），但this.i是内部的i，所以i++后的结果i=101，System.***out***.println(no.i);语句会索引到NotOverriding定义的 **private** **int** i=2;所以输出结果为2，System.***out***.println(no.getI());会转到父类的getI（）方法，this。I=101.所以输出结果为101.

Ppt77

源程序

**package** pkg2;

**public** **class** PrivOverride {

**private** **void** f() {

System.***out***.println("private f()");

}

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

PrivOverride po = **new** Derived();

po.f();

}

}

**class** Derived **extends** PrivOverride {

**public** **int** a;

**public** **void** f() {

System.***out***.println("public f()");

}

}

运行结果

private f()

分析运行步骤，Derived类是PrivOverride类的子类，而derived重写了父类中的f（）方法，所以主函数中po.f()语句是引用父类中的f()方法，这和子类没有直接关联，所以运行结果为private f()

Ppt85

源程序

**package** pkg2;

**public** **class** PolyConstruct {

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

**new** RoundGlyph(5);

}

}

**class** Glyph {

**void** draw() {

System.***out***.println("Glyph.draw()");

}

Glyph() {

System.***out***.println("Glyph() before draw()");

draw();

System.***out***.println("Glyph() after draw()");

}

}

**class** RoundGlyph **extends** Glyph {

**private** **int** radius = 1;

RoundGlyph(**int** r) {

radius = r;

System.***out***.println("RG.RoundGlyph(), radius = " + radius);

}

**void** draw() {

System.***out***.println("RG.draw(), radius = " + radius);

}

}运行结果

Glyph() before draw()

RG.draw(), radius = 0

Glyph() after draw()

RG.RoundGlyph(), radius = 5

分析运行步骤， RoundGlyph是Glyph的子类，主函数运行**new** RoundGlyph(5); 语句，先是跳转到父类的构造方法语句Glyph() { System.***out***.println("Glyph() before draw()");和draw();，因为子类中重写了draw（）语句，方法重写在父类对象构造之前。所以有跳转到子类的**void** draw() { System.***out***.println("RG.draw(), radius = " + radius); 语句，此后radius默认值为0，在之后继续运行，System.***out***.println("Glyph() after draw()");结束之后，开始传入参数r=5，运行RoundGlyph(**int** r) {

radius = r;

System.***out***.println("RG.RoundGlyph(), radius = " + radius);

}

所以运行结果为

Glyph() before draw()

RG.draw(), radius = 0

Glyph() after draw()

RG.RoundGlyph(), radius = 5

Ppt88

源程序

**package** pkg2;

**public** **class** RTTI {

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

Useful[] x = { **new** Useful(), **new** MoreUseful() };

x[0].f();

x[1].g();

((MoreUseful) x[1]).u();

((MoreUseful) x[0]).u();

}

}

**class** Useful {

**public** **void** f() { }

**public** **void** g() { }

}

**class** MoreUseful **extends** Useful {

**public** **void** f() { }

**public** **void** g() { }

**public** **void** u() { }

**public** **void** v() { }

**public** **void** w() { }

}

运行结果

Exception in thread "main" java.lang.ClassCastException: pkg2.Useful cannot be cast to pkg2.MoreUseful

at pkg2.RTTI.main(RTTI.java:9)

老师，我截了图，结果都找不到，所以只好拍了几张照片，以作考证，真是自己写的。

还有咱们的宠物商店练习，我写了好几次都写不出来，在网上查了些类似的才写出来，所以你可以在照片上看到我写了好多petshop，附件里那个文件是最终的。也就是下面的第一个图。





