## 三元操作符的类型必须一致

三元操作符类型转换规则如下：

1. 二个操作数不可以转换，则不转换，返回Object类型
2. 当一个操作数是表达式，一个是具体的数（变量），或二个都是数或二个都是表达式，则看谁的范围，转换成谁。

**package** cn.first;

**public** **class** Demo01 {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

**int** a = 10;

String s1 = String.*valueOf*(a>100?100:10);

String s2 = String .*valueOf*(a>100?100.0:10);

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

}

}

结果为：false

总之：进行三元操作时，三元操作符的类型必须一致。

## 可变参数的注意事项：

1. 可变参数本质上是数组，当一个方法中已经定义了一个可变参数，调用该方法时可以为可变参数传离散值，也可以传数组对象。
2. 当一个方法中的参数中有可变参数，可变参数必须放在其他参数后面。所以一个方法最多只能有一个可变参数。
3. 可变参数练习：

**package** cn.first;

**public** **class** VarargsDemo {

/\*\*

\* **@param** args

\*/

**public** **static** **int** sum(**int**...nums){

**int** sum = 0;

**for**(**int** num: nums){

sum+=num;

}

**return** sum;

}

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

// **TODO** Auto-generated method stub

System.*out*.println(*sum*(2,3));

System.*out*.println(*sum*(**new** **int**[]{2,6,9}));

}

}

1. 避免带有可变参数的方法重载。这样可能带来意想不到情况。

**package** cn.first;

**import** java.text.NumberFormat;

**public** **class** VarargsDemo01 {

/\*\*

\* **@param** args

\*/

**public** **void** calPrice(**float** price,**float** discount){

**float** knockDownPrice = price\*discount/10;

System.*out*.println("打完折后的花费为："+formatCurrency(knockDownPrice));

}

**public** **void** calPrice(**float** price,**float** ... discount){

**float** knockDownPrice = price;

**for**(**float** f: discount){

knockDownPrice = knockDownPrice\*price/10;

}

System.*out*.println("打折后的价格为："+formatCurrency(knockDownPrice));

}

**public** String formatCurrency(**float** f){

**return** NumberFormat.*getCurrencyInstance*().format(f);

}

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

// **TODO** Auto-generated method stub

VarargsDemo01 vd = **new** VarargsDemo01();

vd.calPrice(120, 7.5F);

}

}

调用的是第一个方法。当含有可变参数的重载方法，一般调用没有可变参数的方法。

### 可变参数注意null和空值时，不要有重载方法。

### 重写可变参数的方法是注意的事项：

重写方法注意点：

重写的方法不能缩小访问权限。

方法原型与父类一样（包形参列表一致，重写方法的形参列表要和父类一样，形式也要一样）。

重写的方法不能抛出新的异常，或者超过父类范围的异常。

当重写带可变参数（可变参数用数组对象表示是）的方法，要父引用指向子实例来调用重写的方法，否则会报错的。

**package** cn.first;

**public** **class** VarargsDemo03 {

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

//向上转型

Base base = **new** Sub();

base.fun(122, 76);

Sub sub = **new** Sub();

sub.fun(122, 89);

}

}

**class** Base{

**void** fun(**int** price,**int** ... discount){

System.*out*.println("base fun");

}

}

**class** Sub **extends** Base{

@Override

**void** fun(**int** price, **int** [] discount) {

// **TODO** Auto-generated method stub

**super**.fun(price, discount);

}

}

形参列表一致，重写方法的形参列表要和父类一样（类型，数目，顺序），形式也要一样

## 警惕自增的陷阱

Count = count++;

自增原理：

Public static int mockAdd(int count){

Int temp = count;

Count = count+1;

Return temp;

}

实例：

**package** cn.first;

**public** **class** AddDemo01 {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

**int** count = 0;

**for**(**int** i = 0;i<10;i++){

count = count++;

}

System.*out*.println(count);

}

}

输出： 0；

## 静态导入事项：

1. 少用静态导入，\*（通配符）全导入。
2. 静态导入二个规则：

\* 不使用\* 通配符，除非是导入静态常量类。

方法名有明确意义的方法。

1. 不要静态导入与本类相同的方法名或变量名：（如果有，则本类的方法，变量被调用），编译器”有个最短路径原则”。如果能够在本类中能找到的方法，属性，就不会到父类，或者其他包中找，已本类的方法优先。

**package** cn.first;

**import** **static** java.lang.Math.*PI*;

**import** **static** java.lang.Math.*abs*;

**public** **class** StaticImportDemo {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

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

System.*out*.println("abs="+*abs*(20));

}

**public** **final** **static** Double *PI* =10.0;

**public** **static** **double** abs(**double** d){

**return** 1.0;

}

}

结果为：

PI=10.0

abs=1.0

## 序列化时，显示声明uid（SerialVersionUID 流标识符，即类的版本定义）

**package** cn.com.serial;

**import** java.io.Serializable;

**public** **class** Student **implements** Serializable{

**private** String name;

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

//定义一个流标识符，当序列化的类增加新的内容时，反序列化会读到这些新的内容。

**private** **final** **static** **long** *serialVersionUID* = 456778L;

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

**return** age;

}

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

**this**.age = age;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

}

**package** cn.com.serial;

**public** **class** Producer {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

Student s = **new** Student();

s.setName("jerk");

s.setAge(19);

SerialUtil.*WriteObject*(s);

}

}

**package** cn.com.serial;

**public** **class** Producer {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

Student s = **new** Student();

s.setName("jerk");

s.setAge(19);

SerialUtil.*WriteObject*(s);

}

}

**package** cn.com.serial;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.io.Serializable;

**public** **class** SerialUtil {

**public** **static** **void** WriteObject(Serializable s){

ObjectOutputStream oos = **null**;

**try** {

oos = **new** ObjectOutputStream(**new** FileOutputStream(**new** File("d:/by.txt")));

oos.writeObject(s);

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **finally**{

**if**(oos!=**null**){

**try** {

oos.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

**public** **static** Object readObject(){

ObjectInputStream ois = **null**;

Object obj = **null**;

**try** {

ois = **new** ObjectInputStream(**new** FileInputStream(**new** File("d:/by.txt")));

obj = ois.readObject();

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (ClassNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **finally**{

**if**(ois!=**null**){

**try** {

ois.close();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**return** obj;

}

}

**package** cn.com.serial;

**public** **class** Custome {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

Student s = (Student) SerialUtil.*readObject*();

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

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

}

}

# Instanceof注意事项：

1：instanceof 左边和右边必须要继承关系，否则会报错的

**package** cn.first;

**public** **class** InstanceOfDemo01 {

/\*\*

\* **@param** args

\*/

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

// **TODO** Auto-generated method stub

System.*out*.println("dfsd"**instanceof** Object);

System.*out*.println(**new** String () **instanceof** Object);

System.*out*.println(**new** Object () **instanceof** String);

// System.out.println('a' instanceof Character);//编译不通过，八大原始型没有对象 instanceof是判断某个对象是否属于某个类的实例

System.*out*.println(**null** **instanceof** String);//instanceof只要左边为null(不管null是否强转)就为false。

System.*out*.println((String)**null** **instanceof** String );

// System.out.println(new String() instanceof Date);instanceof左右二边没有继承关系，则编译不通过。

}

}

# 不要只替换一个类（final 修饰的基本类型和引用类型的区别）：

1. 对于final修饰的基本类型和String类型，编译器认为它是稳定的，所以在编译时就直接把值编译到字节码中，以提高代码的效率。
2. 对象final修饰的引用型，编译器认为它是不稳定的，所以在编译时建立的是引用关系。
3. 在（IDE中看不到这种效果）