初始化与清理——Java 编程思想第五章

作者：importGuitar
原文链接：https://ld246.com/article/1571474758839
来源网站：链滴

初始化与清理

构造器

构造器是一种特殊类型的方法，因为它没有返回值。这与返回值为空（void）明显不同。对于空返回，尽管方法本身不会自动返回什么，但仍可选择让它返回别的东西。构造器则不会返回任何东西，（new表达式确实返回了对新建对象的引用，但构造器本身并没有返回任何值）

默认构造器

默认构造器又名无参构造器，作用是创建一个默认对象，如果我们写的类中没有构造器，则编译器会动创建一个默认构造器。但如果我们已经创建了一个构造器（无论是否有参数），编译器便不再创建认构造器。

此外，为了让方法名相同而形式参数不同的构造器同时存在，必须用到**方法重载**

区分重载方法

每个重载方法都必须有一个独一无二的参数列表。

参数顺序不同也足以区分两个方法。

** 但不要根据返回值来区分重载方法，因为有时我们会忽略返回值，比如：

```
void f(){}
int f(){ return 1;}
```

调用时我们只写了f()，那么编译器和别人根本不知道调用的是哪一个。

涉及基本类型的重载

基本类型能从一个较小的类型自动提升至一个较大类型，此过程一旦涉及到重载（参数列表中声明的数类型和实际传入的参数类型大小不一致），可能会造成一些混淆。如果某个重载方法恰好接收传入参数，则会调用这个恰好接受的重载方法；如果传入的参数小于每一个重载方法中声明的参数类型，传入的参数类型会被提升。反过来，如果传入的实际参数类型大于每一个重载方法中声明的参数类型则必须进行强制类型转换，否则编译器会报错。

重载方法与重写方法

重载与重写，总是容易混，可得记住咯

重写方法的规则：

1.参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。

2.返回的类型必须一直与被重写的方法返回类型相同

3.访问修饰符的限制一定要大于被重写方法的访问修饰符。（public>protected>default>private）。

4.重写方法一定不能抛出新的检查异常或者比重写方法申明更加宽泛的检查型异常。

重载方法的规则：

1.必须具有不同的参数列表。

2.可以有不同的返回类型，只要参数列表不同就可以。

3.可以有不同的访问修饰符。

4.可以抛出不同异常。

总结：重载判断比较简单，一条就够了，就是**参数列表必须不同**，其余的相不相同无所谓（当然函数肯定得相同）；而重写方法也就是重新实现该方法，所以除实现部分之外的部分都应该保持不变，但问修饰符可以发生变化，只要大于被重写方法就可以

至于第四条，重写方法一定不能抛出新的检查异常或者比重写方法申明更加宽泛的检查型异常，这一不太理解，加粗待更新吧

@Override注解

可以确保你是在重写方法而不是重载方法（在有次注解的前提下如果不小心重载了，编译器会生成一错误信息）

this关键字

当我们调用某个对象的方法时，编译器会暗自把**所操作对象的引用**作为第一个参数传递给我们要调用方法。如果我们希望在方法内部获得对当前对象的引用，就可以是使用关键字：this。

注意：

this关键字只能在方法内部使用，表示对“调用方法的那个对象的引用”。this的用法和其他对象的用并无不同，但要注意，如果在方法内部调用同一个类的另一方法，就不必使用this，直接调用即可也可以写，但无此必要，编译器会自动添加）。

在构造器中调用构造器

在构造器中，如果为this添加了参数列表，这将产生对符合此参数列表的某个构造器的明确调用。

但是，尽管可以用this调用一个构造器，但不能调用两个。此外，必须将构造器调用置于最起始处，则编译器会报错。

还有，除构造器之外，编译器禁止在其他任何方法中调用构造器

this还有另一种用法，比如：

```
package Unity5;

public class flower {

    String s="hello";
    public void test(String s) {

        System.out.println(this.s);
        this.s=s;
        System.out.println(this.s);
    }
    public static void main(String[] args) {
            flower f=new flower();
            f.test("world");

    }
}
```

参数s的名称与数据成员s的名字相同，直接写s会产生歧义。this.s代表数据成员s（因为this代表当前象的引用嘛），直接写s则是方法的参数列表中的s。

再谈static

static方法可以不创建对象就可调用，即，static方法不需要对象的引用，所以也就没有this关键字。在tatic方法内部不能直接调用非静态方法（通常来讲，在同一个类中的方法，可以直接调用，不需this不需创建对象，但static不同，不能直接调用，需要先创建对象，然后通过对象的引用来调用），反来可以。

还是上面的代码，稍加修改：

```
package Unity5;

public class flower {

    String s="hello";
    public void test(String s) {

        System.out.println(this.s);
        this.s=s;
        System.out.println(this.s);
    }
    public static void main(String[] args) {
        //flower f=new flower();
        //f.test("world");
```

```
        test("world");//编译器报错，因为不能在static方法中调用非static方法，提示在test方法前加stat
c

    }
}
```

按编译器提示，加上static之后：

```
package Unity5;

public class flower {

    String s="hello";
    public static void test(String s) {

        System.out.println(this.s);//报错Cannot use this in a static context
        this.s=s;//报错Cannot use this in a static context
        System.out.println(this.s);//报错Cannot use this in a static context
    }
    public static void main(String[] args) {
        //flower f=new flower();
        //f.test("world");
        test("world");//编译器报错，因为不能在static方法中调用非static方法，提示在test方法前加stat
c

    }
}
```

成员初始化

所有变量在使用前都能得到恰当的初始化。对于方法的局部变量，Java以编译时错误的形式来贯彻这保证。

```
void f(){
    int i;
    i++;//错误，变量i没有初始化
}
```

对于类的数据成员，每个基本类型都会有其默认值，对象的引用默认为null；

构造器初始化

可以用构造器来进行数据成员的初始化，但：**无法阻止自动初始化的进行，它将在构造器被调用之发生。如：

```
public class Counter{
    int i;
    Counter(){
        i=7;
    }
}
```

i首先会被置0，然后置7。

初始化顺序

变量定义的先后顺序决定了初始化的顺序。即使变量定义散布于方法定义之间，它们仍旧会在任何方（包括构造器）被调用之前得到初始化。

静态数据的初始化

无论创建多少个对象，静态数据都只占用一份存储区域。static关键字不能应用于局部变量，因此它能作用于域。

静态数据的初始化与非静态数据没什么区别，但带有继承初始化顺序要注意：

1.根基类中的static域

2.导出类中的static域

3.创建导出类对象

3.1基本类型设为默认值，引用设为null

3.2基类成员变量按顺序初始化

3.3调用基类构造器

3.4导出类成员变量按次序初始化

3.5导出类构造器

3.6构造器其余部分执行

这是第七章7.9.1继承与初始化的内容，代码到第七章总结再贴吧

数组初始化

int[] a1;

或

int a1[];

编译器不允许指定数组的大小，所以像int[9] a1;这样写编译器会报错。

为了给数组创建响应的存储空间，必须写初始化表达式。

对于数组，初始化动作可以出现在代码的任何地方，但也可以使用一种特殊的初始化表达式，它必须创建数组的地方出现。这种特殊的初始化是由一对花括号括起来的值组成的。在这种情况下，存储空的分配（等价于使用new）将由编译器负责。

```
int[] a={1,2,3,};//列表最后的逗号是可选的
```

用new创建数组：

```
int[]  a;
a=new int[9];
```

Java中可以将一个数组赋值给另一个数组，如a2=a1；

其实真正做的只是复制了一个引用，即a1和a2是同一个数组的别名，所以通过a2做的修改，a1也可到。

所有数组均有一个固有成员length，数组下标从0开始，最大为length-1，如果访问越界则会出现运时错误。

可变参数列表

就是这种：

```
package Unity5;

public class KeBianCanShuLieBiao {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        KeBianCan.s("importGuitar","code");
        KeBianCan.s();//此处说明参数个数为0也可以
        KeBianCan.s("importGuitar","code","Guitar");



    }
    static class KeBianCan{
        static void s(String...strings) {//可变参数列表中的参数类型可以选择任意一种，包括基本类型
引用类型
            for(String s:strings) {
                System.out.print(s+" ");
            }
            System.out.println();
        }


    }

}
```

运行结果：

importGuitar code

importGuitar code Guitar

注意：可变参数列表不依赖自动包装机制，实际使用的还是基本类型，但自动包装机制可以和可变参列表和谐共处（当可变参数列表为包装器类型时，自动包装机制将基本类型参数提升为包装器类型）。

枚举类型enum

```
public enum Test{
    NOT,MILD,MEDIUM,HOT,FLAMING//末尾的分号可加可不加
}
```

enum是一个类，并且有自己的方法

可以将enum放在switch语句内使用：

```
package Unity5;

public class enumMeiJu {
```

```
    public enum Test{
        NOT,MILD,MEDIUM,HOT,FLAMING
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Test test;
        test=Test.NOT;
        switch(test) {
        case NOT:
            System.out.println("not");
            break;
        case MILD:
            System.out.println("MILD");
            break;
        default:
            System.out.println("other");

        }
    }

}
```

运行结果：not

感觉这一章好多啊，清理部分我没有写，因为目前还不太明白，可能是因为接触的太少吧，以后理解再来记录。