## Servlet的生命周期

Servlet默认是在第一次访问服务器的时候创建,还可以通过配置webxml,添加load-on-start 来设置成在服务器启动的时候创建.

创建了servlet对象后执行init方法.在每一次请求的时候访问service方法,在销毁的时候执行destory方法.

## Servletcontext

在每一个web服务启动的时候都会有个servletcontext对象,该对象可以存储参数,获取参数,以及移除参数

## 成员变量和局部变量

位置不同:成员变量是在方法外边命名,局部变量是在方法里边命名

内存位置不同: 成员变量是在堆中,局部变量是在栈中

生命周期不同:成员变量随着对象的创建和销毁而创建销毁,局部变量随着方法代码块的的执行和结束来创建和销毁.

默认初始值:成员变量有默认初始值,局部变量没有默认初始值.

## a=a+b与a+=b的区别

如果a和b是同一个数据类型,那么没有区别,假如不是同一个数据类型,a+=b能够自动强转,转换的类型为+=号左边的数据类型.

## 字符串设计成static 加final的类型原因

字符串存储在公共的存储池中,优点就是编译器可以让字符串共享,java设计者认为共享的效率远远高于提取拼接字符串所带来的低效率,

## 类之间的关系

类之间的关系分成三种 依赖、聚合、继承

例如账户、订单、商品项 这三个类来举例

订单依赖于账户，需要有账户类的实例来记录订单信息，订单中的方法会用到账户类信息，而商品项相对于订单来说是聚合，一个订单可能会有多个商品。依赖是 uses-a的关系，而聚合是has-a的关系，继承是is –a 的关系。

## 更改器方法和访问器方法

更改器方法：描述的方法是，对象调用该方法，对象的属性值会发生改变。

访问器方法：意思是对象调用该方法，会返回一个新的同类型的对象，切原有对象不会发生改变。

属性最好不要使用更改器类型的对象，防止在调用get，set方法的时候把原属性修改掉，必须使用的话需要先克隆，在新的对象的上做处理。

## Java的变量更改注意事项

如果在一个方法中传入的不是对象类型的参数，是简单类型的数据，那么传参使用的是拷贝功能，方法中无论怎么改都不会影响原参数。而引用类型的对象，加入在方法中修改它的属性，那么方法外边的对象的属性也是会被修改掉的

**public** **class** Test {

**private** String name;

**public** **static** **void** init (Test test){

test.name =test.name + 3;

System.***out***.println(test.name);

}

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

Test test = **new** Test();

test.name = "aaaa";

*init* ( test);

System.***out***.println(test.name);

}

}

输出

aaaa3

aaaa3

**public** **class** Test {

**private** String name;

**public** **static** **void** init (**int** a){

a =a + 4;

System.***out***.println(a );

}

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

**int** a = 4;

*init* ( a);

System.***out***.println(a);

}

}

输出

8

4

方法不能让对象参数引用一个新的对象，即在现有对象a的情况下，传入方法中，将引用改为对象b，但是方法不返回重新赋值对象a，那么对象a依然是对象a不会变成对象b。

经过分析得出，方法中传入参数，变量获取的是内存的地址值，方法内的变量所代表的地址值发生变化，不会影响方法外边的变量所记录的地址值，而方法修改对象的属性能成功是通过地址值来修改内存的堆中的属性，所以能保存成功。

## 方法的重载

方法的重载只是于方法的名称，和参数有关，与返回值无关。

## 初始化代码块

就是类里边用大括号括起来的一段代码，先于构造器执行。

## 类的设计思想

1：一定要保证数据的私有性：不要破坏类的封装性，要编写一个访问器和方法更改器。

2：一定要对数据初始化：可以不对局部变量进行初始化，但是对对象的实例域要进行初始化，最好不要依赖系统的默认值。

3：不要在类中使用过多的基本类型：

4：不是所有的域都需要独立的域访问器和域更改器：比如一些加了finall的域。

5：将职责过多的类进行拆解。

6：类名和方法名要体现他们的职责。

7：优先使用不可变的类。以处理多线程导致的错误。

## 多态

一个对象变量可以指示多种实际类型的现象被称为多态，在运行的时候能够自动选择调用哪个方法的现象叫做动态绑定。

## 多态类型数组方法引用

用基类声明的数组不能用 基类[i].实现类特有的方法。这样做是不正确的，因为基类中没有该方法。

## 抽象类

抽象类是不能被实例化的，因为里边有可能存在着抽象方法，但是抽象类使用于多态。

## 接口

接口方法不需要声明权限，自动是public的，且必须是public

在接口中可以创建静态方法，但是不鼓励。

接口中可用default 修饰方法，这样就可以给接口的中的方法写实体了，并且在新的实现类中不用覆盖这个方法，可以直接调用。保证老的接口能添加新的方法，做到与之前实现这个接口的实现类能够完美兼容。

接口默认的方法和父类的方法冲突，父类方法优先，接口1和接口2方法冲突，必须要通过方法覆盖来解决冲突。

## 内部类

内部类可以使用外边类的属性，私有属性也ok。

局部内部类访问的局部变量必须要声明成final的。

## 异常与程序的运行

程序中如果报异常，程序会停止运行，加入try catch 把异常吃掉，那么程序会继续运行，但是在try catch抛异常，那么程序仍然会停止执行。加入catch中的异常和实际出现的异常不一致，相当于没有catch。

在catch中可以包装一下发生的异常，从而抛出别的类型的异常。

## Finally中的return

当finally中有了返回语句，return，那么原有的return将会被覆盖，不再执行。

## 带资源的try语句

**try** (Scanner in = **new** Scanner(**new** FileInputStream("afasdfasf"))){

**return** a\*a;

} **catch** (Exception e) {

**throw** **new** Exception(e.getMessage());

}

主要应用于防止关闭数据流失败，即finally中的操作出现异常的情况，try语句中的资源可以自己主动关闭流，并且自带trycatch。

## 异常的使用规则

1：尽可能不适用异常，因为异常很耗费时间。

2：不要过分的细化异常，即每个异常都有trycatch，可以将多个异常放到一个try中，多catch几次。

3：利用异常的层次结构。

4：不要压制异常，假如有可能有异常，那么久使用trycatch。

5：异常可以向上抛出。

## 泛型

泛型类：

Public class Pair<T> {

Private T t;

}

泛型类的作用是在创建的时候定义该类中属性的类型。

泛型方法：

Public static <T> T getT (T t){

}

泛型类需要验证是否是某个类的子类的时候，可以用

Public static <T entends Comparable & Serializable> T get T (T t){};多个限定类型用& 隔开。在泛型类和泛型方法上都适用。

## 不能创建参数化类型的数组

例如： Pair<String>[] a = new Pair<String>[10];因为泛型擦除会导致非泛型类型也可以保存，所以不允许创建这样的数组。

虽然不能创建这样的数组，但是可以声明变量。

还有通过方法，给参数化类型的的数组添加变量，就不会报错。

## 泛型类属性与构造器

一个泛型类的内部属性，使用了泛型变量，构造器内部不能直接new这个变量来初始化。构造器nebula要想初始化这个属性，需要调用方法来进行初始化。

不能再静态域或方法中引用类型变量。因为擦除的原因。

## 泛型与catch

Catch中不能使用类型变量，不合法。但是在catch中可以使用类型变量，例如继承Throwable 的子类。

Public static <T extends Throwable> void doWork<T t>{

Try{

} catch(Throwable throwable){

t.方法

}

}

## 泛型类型的继承规则

泛型变量b继承a 那么带有泛型b的类不能转换成带有a的泛型类。

## 关于泛型继承设置属性问题

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

Pair<B> pB = **new** Pair<B>();

Pair<? **extends** A> pV = pB;//没有问题

pV.setT(**new** A());//编译报错

}

因为编译器只是知道是属于A的某个子类，但是具体是哪个类没有办法指定，这里需要指定具体类才行。所以会报错，但是很费解。

带有超类限定的通配符可以向泛型对象写入，带有子类型限定的通配符可以从泛型对象读取。

## 无限定通配符

可以调用获取值得方法，返回的是个Object，但是不能修改值。