JAVA编程思想

第一章：对象导论

汇编语言是对底层机器的轻微抽象，接着出现了许多所谓命令式语言(FORTRAN,BASIC,C等)等是对汇编语言的轻微抽象。

它们所做的主要抽象仍要求在解决问题时要基于计算机结构，而不是基于所要解决的问题的结构来考虑。

面向对象方式通过向程序员提供表示问题空间中的元素的工具而更进了一步，使得程序员不会受限于任何特定类型的问题，我们将问题空间中的元素及其在解空间中表示称为-对象。

这种思想的实质是：成员可以通过添加新类型的对象使自身适用于某个特定的问题。所以OOP允许根据问题来描述问题，而不是根据运行解决方案的计算来描述问题。

1.万物皆为对象：将对象视为奇特的变量，既可以存储数据，也可以在自身上执行操作。

2.程序是对象的集合，他们通过发送消息来告知彼此所要做的：可以把消息想象为对某个特定对象方法的调用请求

3.每个对象都有自己的由其他对象所构成的存储：可以通过现有的对象来创建新的对象，在程序中构建复杂的的体系

4.每个对象都拥有其类型：每个类最重要的区别于其他类的特性就是“可以发送什么呀的消息给它”

5.某一特定类型的所有对象都可以接受同样的消息：接口特性，产生可替代性是OOP中最强有力的概念之一

每个对象都有一个接口，接口确定了对某一特定对象所能发出的请求。

因为类描述了具有相同特性(数据元素)和行为(功能)的对象集合，所以一个类实际上就是一个数据类型。

将对象视为服务提供者

public private protected：protected关键字与Private作用相当，差别仅在于继承的类可以访问protected成员，但是不能访问private成员

还有一种默认的访问权限，就是当没有指定中三者中的一个时，在同一个包中可以访问，其他包相当于private一样不能访问

创建新类时，首先考虑的是组合，然后才是继承，组合带来了极大的灵活性

导出类也是一个基类，将包含基类的所有属性和方法，覆盖和创建新方法能将导出类和基类区分开来

"是一个"就是纯粹的继承，不添加新方法，"像是一个"则是添加了新的方法

在OOP中，程序直到运行时才能确定代码的地址，所以当消息发送到一个泛化对象时，必须采用其他的机制，就是后期绑定的概念，编译器确保被调用的方法存在，并对调用参数和返回值执行类型检查(弱类型的语言)，此处说的便是多态(可扩展性)。

多态神奇的地方在于编译器在编译时是不知道具体要执行的方法的，而取决导出类的向上转型

单根继承：使垃圾回收器的实现变的容易很多，而垃圾回收器正是JAVA相对C++的重要改进之一。由于所有对象都保证具有其类型信息，因此不会因无法确定对象类型而陷入僵局，这对于系统级操作(如异常处理)显得尤其重要，并且给编程带来了更大的灵活性

List(用户存储序列)，Map(用来建立对象之间的关联)，Set(每种对象类型只持有一个)

ArrayList和LinkedList，经常改动插入删除的用LinkedList，经常查询的用ArrayList

在容器中，只存储Object对象，取出来时需要向下转型，而向下转型是非常危险的，需要确保转换的类型是可行的，就里就引入了泛型的概念

ArrayList<Shape> shapes = new ArrayList<Shape>(); JAVASE5新特性

垃圾回收器知道对象何时不再被使用，并自动释放对象所占用的内存。这一点同所有对象都是继承自单根基类Object以及只能以一种方式创建对象(在堆上创建)这两个特性结合起来，使得用java编程的过程较之用C++编程要简单得多。

并发的隐患在于共享资源，JAVASE5增加了大量额外的库支持

第二章：一切皆是对象

String s;创建了一个引用，不能说是指针

寄存器、堆栈(向上释放内存，向下分配内存，java对象并不存在于此)、堆(RAM)、常量存储(ROM只读存储器)、非RAM存储(流对象和持久化对象)

基本类型存储在堆栈中

boolean char(0,2^16-1) byte(-128,+127) short(-2^15,+2^15) int(-2^31,+2^31) long(-2^63,+2^63) float(IEEE754) double(IEEE754)

BigInteger与BigDecimal支持任意精度的整数和浮点数

作用域完全取决于{}的位置

当变量属于类的成员时，java会赋予它一个默认的变量，但是如果是局部变量，则必须给它赋初值。

String中的每个字符都是16位或2个字节，以此来提供对Unicode字符集的支持

通过import来准确低告诉编译器你想要的类是什么

static域独立于类之外，而且永远只创建一份

初始化和创建都是在构造器里完成的，默认构造器只有在没有构造器的时候JAVA才会创建，如果有构造器了则不会

public protected 默认 private：private定义的只有本来中可见，new新对象也不行，protected在同包中相当于public，不同包相当于private

protected处理的是继承的概念，也支持包访问权限，默认的权限即使继承了也不能访问

继承一个类后可以直接使用它的方法，就像在本类里一样

对于类的访问权限，仅有两个选择：包访问权限或Public

类中的方法分为：构造方法，普通方法，静态方法，每种访问的权限都不一样

在构造方法私有化的同时在静态方法中返回已经被static的类引用，那就是单例模式了，永远只new一次

在定义一个类的时候，只要不用public就能摆脱类名的限制了，但是必须要同包下才能使用它，使用的是默认权限

import的类必须是public声明的类

重要的一点就是，不管是protected还是public，只要不是静态的方法，调用的时候需要严格的class.fun()来调用

最重要的一点，protected在被继承的时候会变成public，不被继承的时候相当于默认包权限(protected static是个神奇的东西)

所有的public private protected都只需要关注一个问题：能不能看到

被继承的类的构造方法至少是protected权限，不然子类将无法访问

第7章：复用现有类

1.在新的类中产生现有类的对象，新的类由现有类对象所组成，这种方法称为组合(组合的设计模式)

2.另外一种就是继承

惰性初始化是指类中有的属性不需要一开始就赋默认值，而等到使用这个对象之前再去初始化

当创建了一个导出类的对象时，该对象包含了一个基类的子对象。这个子对象与你用基类直接创建的对象是一样的。二者区别在于，后者来自于外部，而基类的子对象被包装在导出类对象内部

导出类构造方法中总是会调用基类的构造器

导出类构造方法被调用的时候，会默认调用基类的无参构造方法，如果需要使用有参的，要用到super(args)

如果基类中不存在无参构造方法，则需要导出类中明确声明super(args)

private Class class = new Class(); public void fun(){class.fun()}; 是组合的基本特征

编译器强制导出类去初始化基类，并且要求导出类在构造器起始处就要这么做，但是它并不监督必须将成员对象也初始化

Class[] class = new Class[3]; 可以直接new出三个对象，但是都是没有初始化的

调用清理方法时，是先调用导出类的清理方法，然后再调用基类的清理方法

垃圾回收器可能永远也无法被调用，即使被调用，它也可能以任何它想要的顺序来回收对象。最好的办法是除了内存以外，不能依赖垃圾回收器去做任何事情。如果需要进行清理，最好编写自己的处理方法，但不要使用finalize()。

组合和继承的选择：组合技术通常用于想在新类中使用现有类的功能而且它的接口这种情形。也许只用其他的一个方法(Build设计模式)

继承是使用某个现有类，并开发一个它的特殊版本。是一个的关系用继承来表达，有一个的关系则用组合来表达，一个最清晰的判断办法就是问一问自己是否需要从新类向基类进行向上转型

向上转型可以用"新类是现有类的一种类型"加以概括

由于向上转型是从一个较专用的类型向较通用的类型转换，所以总是很安全的

一个既是static又是final的域只占据一段不能改变的存储空间

而用于对象引用，final使引用恒定不变。一旦引用被初始化指向一个对象，就无法再把它改为指向另一个对象。然而，对象自身却是可以被修改的，java并未提供使任何对象恒定不变的途径(但可以自己编写实现)

如果final一开始没有被初始化，则一定需要在constructor中对其进行初始化

被定义成final的在类中必须已经被初始化好了，如果传递了final或传递参数中用了final，则只能读，不能写

用final来定义方法是不能得到什么优化效果的

用private final的方法，是不会被覆盖的，因为导出类根本就看不到，即使可以向上转型，也不能用这些方法，这仅仅是因为private

定义是final的类将不能被继承，但可以使用，因为被类被指定成final时，里面的方法也全是final了，而final方法不支持继承

类的任何static成员都是在被访问时才初始化的，构造方法也是static的

第8章 多态

继承允许对象视为它自己本身的类型或其基类型来加以处理：将多种类型(从同一基类导出的)视为同一类型来处理，只要同一份代码也就可以无差别的运行在这些不同类型之上了

多态也称为(动态绑定、后期绑定或运行时绑定)，对象既可以作为它自己本身的类型使用，也可以作为它的基类型使用

导出类向基类向上转型的时候可能会"缩小"接口，但不会比基类的全部接口更窄

应用多态，我们在编写代码的时候只需要跟基类打交道，而不需要考虑其众多的导出类

Java中除了static方法和final方法(private方法属于final方法)之外，其他所有的方法都是后期绑定

向上转型后得到的基类只能用自己的方法，是不可能用导出类所定义的新方法的，如果基类定义了private方法，那就不能被继承

一个基类在使用时，一定会先调用自己的方法，然后如果发现此方法被覆写了，会进入子类去调用子类的方法，基础数据类型和静态方法不会被覆写，构造器也是static的，没有多态性

在导出类的构造主体重，如果没有明确指定调用某个基类构造器，它就会"默默"的调用默认构造器。如果不存在默认构造器，编译器就会报错

构造器初始化的时候，会先初始化静态变量与方法，然后是属性变量，最后才是构造器初始化(先基类静态方法或属性，然后是子类静态方法和属性，其次是基类一般属性和初始化块跟构造器，最后是子类的是属性和初始化跟构造器)，其中的静态与静态需要初始化的时候只分代码上的先后。初始化代码时，一开始全部内存都是0，然后马上覆盖方法。

协变返回类型是指return指能直接返回导出类型给基类

用继承表达行为间的差异，并用字段表达状态上的变化：状态模式

向下转型中只要增加一个导出类的强制转型就能保证所有的方法转型成功

第9章 接口

抽象类(abstract class)是普通的类与接口之间的一种中庸之道，如果以个类包含一个或多个抽象方法，该类必须被限定为抽象的

如果抽象类中的导出类不实现基类的抽象方法，则它必须也是抽象类，当然也可以创建没有抽象方法的抽象类

一个接口的表示：所有实现了该特定接口的类看起来都像这样

复用代码的第一种方式是客户端程序员遵循该接口来编写他们自己的类

适配器模式是使新的类继承一个接口实现所需要的功能：接受你所拥有的接口，然后生成所需要的方法体

可以继承任意多个接口，并可以向上转型为每个接口，因为每一个接口都是独立类型

如果一个类继承了一个基类和许多的接口，那基类中恰巧有一个方法与接口中的一个方法一致，则不再需要覆写此方法

使用接口的核心原因：为了能够向上转型为多个基类(以及由此带来的灵活性)

如何确定使用抽象类还是接口：如果知道某事物应该成为一个基类，那么第一选择应该是使它成为一个接口

接口是可以继承接口的，用extends关键字 interface new extends old1,old2 ，允许用extends继承多个接口

在接口中，继承此接口的导出类中如果方法与接口中的一致而返回类型又不一样是不允许的

public class AdaptedRandomDoubles extends RandomDoubles 适配器模式

如果在接口中这样定义域 int RANDOM\_INT = new Random(47).nextInt(10); 则实际在用的时候已经初始化成实际的数值了，不属于接口的一部分

在类中嵌套接口的语法是相当显而易见的，就像非嵌套接口一样，可以拥有Public和包访问两种可视性，嵌套接口可以使用private来定义

工厂模式经常会有一个方法getClass(autowired)来获得实际来操作的对象

第10章 内部类

内部类允许你把一些逻辑相关的类组织在一起，并控制位于内部的类的可视性

实例化内部的两种方式，在内部时直接new，在外部时先new出外部的类，然后在调用外部类的方法来实例化内部

当生成一个内部类的对象时，此对象与制造它的外围对象之间就有了一种联系，所以它能访问其他外围对象的所有成员，而不需要任何特殊条件。此外，内部还拥有其外围类的所有元素的访问权限

当某个外围类的对象创建了一个内部类对象时，此内部类对象必定会秘密的捕获一个指向外围类对象的引用。然后，在你访问此外围类的成员时，就是用那个引用来选择外围类的成员。

如果内部类中想产生外部类的对象，可直接使用 外围类.this，想直接new内部类的时候，需要使用 外围类.new Inner();其实这才是基础方法

在拥有外部类对象之前是不可能创建内部类对象的。这是因为内部类对象会暗暗地连接到创建它的外部类对象上。但是没如果你创建的是嵌套类(静态内部类)，那么它就不需要对外部类对象的引用

内部类是可以定义在方法内的，变成只有这个方法能访问中个类

return new Class{};是匿名内部类一个重要的应用，在此，匿名类没有名字，也不可能有构造器

返回的匿名内部类中，返回的对象默认是一种继承，如果return返回的是个接口，则是implements，如果是基类，则是extends，然后回根据传入的参数调用父类的构造器

从外部传入给匿名内部类的参数，必须是final的

匿名内部类与正规的继承相比有些受限，因为匿名内部类既可以扩展类，也可以实现接口，但是不能两者兼备。而且如果是实现接口，也只能实现一个。

工厂模式的意义：定义一个方法，产生特定的产品，如果那个方式直接对应一个内部类的return 则非常简洁

优先使用类而不是接口

如果不需要内部类对象与其外围类对象之间的联系，那么可以将内部类声明为static：称为嵌套类。

这样的结果的new的时候变得容易理解，但是不能再引用外围类的域了

如果定义一个Interface却可以让他输出，很明显就是使用内部类，定义在inerface内的内部类都是public和static的

1. 通过异常处理错误

异常的基本概念是用名称代表发生的问题。并且异常的名称应该可以望文知意。

异常处理的原则是：只有在你知道如何处理的情况下才捕获异常

当异常产生的生活，可能在当前环境中还没有足够的信息来解决这个问题，所以把这个问题提交到一个更高级别的环境中，在这里将作出正确的决定。

普通问题是指，在当前环境下能够得到足够的信息，总能处理这个错误。而对于异常情形，就不能继续下去了，当前环境信息不足，异常处理则是将此问题提交给上一级环境。

Java支持终止模型，表明错误已无法挽回，也不能回来继续执行；另一种是恢复模型，异常处理程序的工作是修正错误，然后重新尝试调用出问题的方法，并认为第二次能成功，此时不能抛出异常，应该调用方法来修正错误。

对于Throwable类来说，getMessage()和toString()方法差不多

在编译时被强制检查的异常成为被检查的异常

Exception，Error，RuntimeException继承了Throwable，一般有getMessage()；getLocalizedMessage()；toString()；而Throwable的方法为printStackTrace()；printStackTrace(PrintStream)；getMessage()是指输出msg这个参数

异常被catch捕获后还可以用throw重新抛出 throw e;同一级的catch将会被忽略

RuntimeException被称为不受检查异常，这种异常属于错误，将被自动捕获

finally关键字表示此段异常处理代码一定会被执行，finally的使用可以导致丢失异常，在一个异常还未处理的时候抛出另一个

new RuntiemException(e)将检测型异常转换为运行时异常抛出

第13章 字符串

String对象是不可变的，具有只读特性，String类的每一次改变，其实都是创建了一个全新的String对象

对于一个方法而言，参数是为该方法提供信息的，而不是想让该方法改变自己的，这样会使得代码易于编写和阅读

String的"+"和"+="是java仅有的两个重载过的操作符，而java不允许程序员重载任何操作符

StringBuilder方法集：insert()，replace()，substring()，reverse()，最常用的append()和toString()，还有delete()方法

StringBuffer是线程安全的，StringBuilder是javase 5引入的

如果想用toString()打印对象的内存地址，不能用this，而应该用Object.toString()，就是super.toString()；

正则表达式：\\\\表示需要反斜杠(-|\\+)?\\d+ 正数或是负数

在java中,\\的意思是我要插入一个正则表达式的反斜线

字符类

. 任意字符

[abc] 包含a、b和c的任何字符(和a|b|c作用相同)

[^abc] 除了a、b和c之外的任何字符(否定)

[a-zA-Z] 从a到z或从A到Z的任意字符(范围)

[abc[hij]] 任意a、b、c、h、i和j字符(与a|b|c|h|i|j作用相同)(合并)

[a-z&&[hij]] 任意h、i或j(交)

\s 空白符(空格、tab、换行、换页和回车)

\S 非空白符([^\s])

\d 数字[0-9]

\D 非数字

\w 词字符[a-zA-Z0-9]

\W 非词字符[^\w]

边界匹配符

^ 一行的起始

$ 一行的结束

\b 词的边界

\B 非词的边界

\G 前一个匹配的结束

组概念：组号为0表示整个表达式，组号1表示被第一对括号括起来的组，依次类推。

第14章 类型信息

RTTI(Run-Time Type Identification)运行时详细的类型信息，使用反射，就是Class类实现

特殊的编程问题——如果能够知道某个泛化引用的确切类型，就可以使用最简单的方式去解决它，从List里取出父类是不明确

Class的方法Class.forName(package.class);Class.class更加简单高效。引用.class的不会被初始化，而forName一定会

cc.getName()、cc.getSimpleName()、cc.getCanonicalName()

cc.getInterfaces()获得继承的所有接口，List对象

cc.getSuperclass()获得继承的父类

cc.newInstance()默认无参构造

Class<? extends super>表示存储的是super的子类 Class<? super sub>表示存储的是sub的超类，这样返回的instance只能是Object

基本数据类型中boolean.class = Boolean.Type

所有的类都是在对其第一次使用时，动态加载到JVM中的。当程序创建第一个对类的静态成员的引用时，就会加载这个类。

当使用.class来创建对Class对象的引用时，不会自动地初始化该Class对象。

如果一个static final值是”编译器常量”，那么这个值不需要对Initable类进行初始化就可以被读取。

向下转型如果失败，会抛出ClassCastException，在转型之前可以用instanceOf去检测。

人们想要在运行时获取类的信息的另一个动机，便是希望提供在跨网络的远程平台上创建和运行对象的能力：这被称为远程方法调用。(传统的RTTI的字面变量和Class.forName()都是已知的本地类)

在通过反射与一个未知类型的对象打交道时，JVM知是简单地检查这个对象，看它属于哪个特定的类(就像RTTI那样)。在用它做其他事情之前必须先加载那个类的Class对象。因此，那个类的.class文件对于JVM来说是必须是可获取的：要么在本机上，要么可以通过网络获取。

RTTI和反射之间的真正的区别只在于，对RTTI来说，编译器的编译时打开和检查.class文件。而对于反射机制来说，.class文件是在编译时不可获取的，所以是在运行时打开和检查.class文件。

# 泛型(参数化类型)

Final类不能扩展

理解边界所在，你才能成为程序高手。因为只有知道了某个技术部能做到什么，你才能更好地做到所能做的。

泛型的主要目的之一就是用来指定容器要持有什么类型的对象，而且由编译器来保证类型的正确性。

在字段上加入final，能提供和private同样的安全性

### Java I/O系统

对程序语言的设计者来说，创建一个好的输入/输出(I/O)是一项艰巨的任务：不仅存在各种I/O源端和想要与之通讯的接受端(文件、控制台、网络链接等)，而且还需要以多种不同的方式与他们进行通信(顺序、随机存取、缓冲、二进制、按字符、按行、按字等)

策略模式：在org.chapter18.DirList中，策略的目的就是提供了代码行为的灵活性。

InputStream：ByteArrayInputStream、StringBufferInputStream、FileInputStream、PipedInputStream、SequenceInputStream(合并多个InputStream)、FilterInputStream(作为装饰器的接口)。

OutputStream：ByteArrayOutputStream、FileOutputStream、PipedOutputStream、FilterOutputStream(装饰器接口)。

装饰器必须具有和它所装饰的对象相同的接口，但它也可以扩展接口，而这种情况只发生到个别filter类中。

但是，装饰器模式也有一个缺点：在编写程序时，它给我们提供了相当多的灵活性(因为我们可以很容易地混合和匹配属性)，但是它同时也增加了代码的复杂性。Java I/O类库操作不便的原因在于：我们必须创建许多类——”核心”I/O类型加上所有的装饰器，才能得到我们所希望的单个I/O对象。

PrintStream最初的目的便是为了以可视化格式打印所有的基本数据类型以及String对象。这和DataOutputStream不同，后者的目的是将数据置于”流”中，使DataInputStream能够可移植地重构他们。

InputStream和OutputStream在以面向字节形式的I/O中可以提供极有价值的功能，Reader和Writer则提供兼容Unicode与面向字符的I/O功能。

有时我们必须把来自于”字节”层次结构中的类和”字符”层次结构中的类结合起来使用。为了实现这个目的，要用到适配器(adapter)类：InputStreamReader可以把InputStream转换为Reader，而OutputStreamWriter可以把OutputStream转换为Writer。

设计Reader和Writer继承层次结构主要是为了国际化。老的I/O流继承层次结构仅支持8位字节流，并且不能很好的处理16位Unicode字符。由于Unicode用于字符国际化(Java本身的char也是16位的Unicode)，所以添加Reader和Writer继承层次结构就是为了在所有的I/O操作中都支持Unicode。

PrintStream InputStream PrintWriter Writer

要读取格式化数据，可以使用DataInputStream，使用ByteArrayInputStream必须提供字节数组；使用DataInputStream需要使用available检验是否到最后了

PrintWriter是能提供某种格式化的写入

DataInputStream和DataOutputStream可用于数据的存储和恢复，具有平台无关性。

使用RandomAccessFile，类似于组合使用DataInputStream和DataOutputStream(因为他们实现了相同的接口：DataInput和DataOutput)

管道流PipedInputStream、PipedOutputStream、PipedReader及PipedWriter的价值体现在多线程上，用于任务之间通讯。

标准I/O的意义在于：我们可以很容易地把程序串联起来，一个程序的标准输出可以成为另一个程序的标准输入。System.in(InputStream)、System.out(PrintStream)、System.err。

新I/O：目的是为了提交速度，速度的提高在文件I/O和网络I/O都有可能发生。速度的提高来自于所使用的结构更接近于操作系统执行I/O的方式：通道和缓冲区。只要把缓冲器送到通道，通道要么从缓冲器获得数据，要么想缓冲器发送数据。唯一直接与通道交互的缓冲器是ByteBuffer——也就是说，可以存储未加工字节的缓冲器。没有办法输出或读取对象，即使是字符串对象也不行。

旧I/O类库中有三个类被修改了，用以产生FileChanel。这三个被修改的类是FileInputStream、FileOutputStream以及用于既读又写的RandomAccessFile。Reader和Writer这种字符模式类不能用于产生通道；但是java.nio.channels.Channels类提供了实用的方法，用以在通道中产生Reader和Writer。

一旦调用read()来告知FileChannel向ByteBuffer存储字节，就必须调用缓冲器上的flip()，让它做好让别人读取字节的准备，这样做适用于获取最大速度。而ByteBuffer中的读取完之后，需调用clear()方法清理。-1是Unix和C的分界符。

缓冲器容纳的是普通的字节，为了把他们转换成字符，我们要么在输入他们的时候对其进行编码(这样，他们输出时才具有意义)，要么在将其从缓冲器输出时对他们进行解码。

对象序列化：如果对象能在程序不运行的情况依然能存在并保存其信息，那将非常有用。这样，在下次运行程序时，该对象将被重建并且拥有的信息与程序上次运行时它所拥有的信息相同。轻量级持久性。

对象序列化特别”聪明”的一个地方是它不仅保存了对象的”全景图”，而且能够追踪对象内所包含的所有引用，并保存那些对象；接着又能对对象内包含的每个引用进行追踪；依次类推。这种情况有时被称为”对象网”。

第20章 注解

注解(也称为元数据)为我们在代码中添加信息提供了一种形式化的方法，使我们可以在稍后某个时刻非常方便的使用这些数据

注解可以用来生成描述符文件，甚至或是新的类定义，并且有助于减轻编写"样板"代码的负担

@Override，@Deprecated(声明不赞成使用的类或方法)，@SupressWarnings关闭不当的编译器警告信息

元注解

@Target(ElementType.METHOD) CONSTRUCTOR，FIELD(域)，LOCAL\_VARIABLE(局部变量)，PACKAGE，PARAMETER，TYPE(类，接口)

@Retention(RetentionPolicy.RUNTIME) SOURCE源码中(被编译器丢弃)，CLASS类文件中(被VM丢去)，RUNTIME运行时

@Documented 将此注解包含在javaDoc中

@Inherited允许子类继承父类的注解