常识

JVM – JDK -JRE关系？（JDK包含JRE，JRE包含JVM）

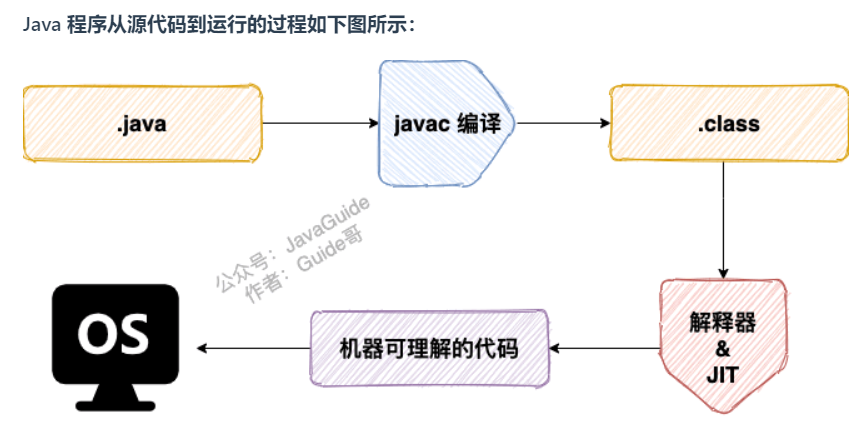
1. Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。
2. JDK 是 Java Development Kit 缩写，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器（javac）和工具（如 javadoc生成api文档 和 调试工具jdb）。它能够创建和编译程序。
3. JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机（JVM），Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

什么是字节码？

在 Java 中，JVM 可以理解的代码就叫做字节码（即扩展名为 .class 的文件），它不面向任何特定的处理器，只面向虚拟机。Java 语言通过字节码的方式，**在一定程度上解决了传统解释型语言执行效率低的问题**，**同时又保留了解释型语言可移植的特点**。

编译型语言源代码---编译器---一次性编译---得到机器可以理解的代码（开发效率低，执行效率高）

解释型语言源代码---解释器---一句句解释---得到机器可以理解的代码（开发效率高，执行速度慢）



Java和c++的区别：

虽然，Java 和 C++ 都是面向对象的语言，都支持封装、继承和多态，但是，它们还是有挺多不相同的地方：

* **Java 不提供指针来直接访问内存**，程序内存更加安全
* **Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口可以多继承。**
* **Java 有**自动内存管理**垃圾回收机制(GC)**，不需要程序员手动释放无用内存。
* **C ++同时支持方法重载和操作符重载**，但是 **Java 只支持方法重载**（操作符重载增加了复杂性，这与 Java 最初的设计思想不符）。

语法基础

注释有哪几种形式？

Java 中的注释有三种：

* 单行注释
* 多行注释
* 文档注释。

标识符和关键字的区别是什么？

* 在我们编写程序的时候，需要大量地为程序、类、变量、方法等取名字，于是就有了 标识符 。简单来说， **标识符就是一个名字**。
* 有一些标识符，Java 语言已经赋予了其特殊的含义，只能用于特定的地方，这些特殊的标识符就是 关键字 。简单来说，**关键字是被赋予特殊含义的标识符** 。

例子：新开的发廊名（标识符）不能叫警察局（关键字）。

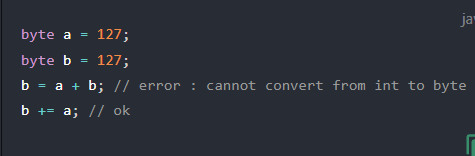
自增自减运算符？

“符号在前就先加/减，符号在后就后加/减”。

当 b = ++a 时，先自增（自己增加 1），再赋值（赋值给 b）；当 b = a++ 时，先赋值(赋值给 b)，再自增（自己增加 1）。

a = a + b 与 a += b 的区别？

+= 隐式的将加操作的结果类型强制转换为持有结果的类型。如果两这个整型相加，如 byte、short 或者 int，首先会将它们提升到 int 类型，然后在执行加法操作。



因为 a+b 操作会将 a、b 提升为 int 类型，所以将 int 类型赋值给 byte 就会编译出错

continue、break 和 return 的区别是什么？

* continue ：指**跳出当前的这一次循环，继续下一次循环**。
* break ：指**跳出整个循环体，继续执行循环下面的语句**。
* return： 用于**跳出所在方法，结束该方法的运行**。return 一般有两种用法：

return：直接使用 return 结束方法执行，用于没有返回值函数的方法

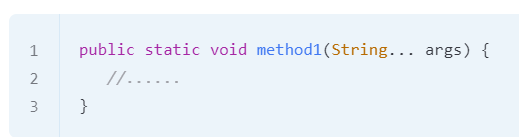
return value ：return 一个特定值，用于有返回值函数的方法

java中有三种移位运算符

* << :左移运算符,x << 1,相当于x乘以2(不溢出的情况下),低位补0
* >> :带符号右移,x >> 1,相当于x除以2,正数高位补0,负数高位补1
* >>> :无符号右移,忽略符号位,空位都以0补齐

可变长参数？

从 Java5 开始，Java 支持定义可变长参数，所谓可变长参数就是允许在调用方法时传入不定长度的参数。就比如下面的这个 printVariable 方法就可以接受 0 个或者多个参数。



另外，可变参数只能作为函数的最后一个参数，但其前面可以有也可以没有任何其他参数。

遇到方法重载的情况怎么办呢？会优先匹配固定参数还是可变参数的方法呢？

---答案是会优先匹配固定参数的方法，因为固定参数的方法匹配度更高。

Switch 判断语句支持什么数据类型？

在jdk1.5之前switch判断语句只支持byte short char int四种数据类型

在jdk1.5时，switch判断语句新增支持枚举类Enum 和 byte short char int的包装类（Byte，Short，Character，Integer）。之所以支持枚举类，是因为枚举类中有一个ordinal方法，该方法实际上返回一个int型的序号。

从jdk7开始， switch 判断语句支持使用字符串，但这仅仅是一个语法糖。内部实现在 switch 中使用字符串的hash code，而该hash code是一个int型。

对equal()和hashCode()的理解?

* 为什么在重写 equals 方法的时候需要重写 hashCode 方法?

因为有强制的规范指定需要同时重写 hashcode 与 equal 方法，许多容器类，如 HashMap、HashSet 都依赖于 hashcode 与 equals 的规定。

* 有没有可能两个不相等的对象有有相同的 hashcode?

**有可能**，两个不相等的对象可能会有相同的 hashcode 值，这就是为什么在 hashmap 中会有冲突。相等 hashcode 值的规定只是说如果两个对象相等，必须有相同的hashcode 值，但是没有关于不相等对象的任何规定。

* 两个相同的对象会有不同的的 hash code 吗?

**不能**，根据 hash code 的规定，这是不可能的。

final、finalize 和 finally 的不同之处?

* final 是一个修饰符，可以修饰变量、方法和类。如果 final 修饰变量，意味着该变量的值在初始化后不能被改变。
* Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的，但是什么时候调用 finalize 没有保证。
* finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常。

===========================================================

========================================================

面向对象基础

面向对象和面向过程的区别：

* 面向过程把解决问题的过程拆成一个个方法，通过一个个方法的执行解决问题。
* 面向对象会先抽象出对象，然后用对象执行方法的方式解决问题。另外，面向对象开发的程序一般更易维护、易复用、易扩展。

面向对象特性：封装、继承、多态

封装：

含义：利用抽象数据类型将数据和基于数据的操作封装在一起，使其构成一个不可分割的独立实体。数据被保护在抽象数据类型的内部，尽可能地隐藏内部的细节，只保留一些对外接口使之与外部发生联系。用户无需知道对象内部的细节，但可以通过对象对外提供的接口来访问该对象。目标：高内聚、低耦合；将对象封装为 高度自治 和 相对封闭 的个体。

优点：

 **减少耦合**: 可以独立地开发、测试、优化、使用、理解和修改

 **减轻维护的负担**: 可以更容易被程序员理解，并且在调试的时候可以不影响其他模块

 **有效地调节性能**: 可以通过剖析确定哪些模块影响了系统的性能

 **提高软件的可重用性**

 **降低了构建大型系统的风险**: 即使整个系统不可用，但这些独立模块却有可能是可用的

-------------------------------------------------------------------------------

继承：在定义和实现一个类的时候，可以在一个已经存在的类的基础上进行，把已存在的类的内容作为自己的内容，并可以加入新的内容或者修改原来的方法，来满足自己的需求，这就是继承。

优点：提高了软件的可重用性和可扩展性。

关于继承如下 3 点请记住：

* 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，只是拥有。
* 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
* 子类可以用自己的方式实现父类的方法。

-------------------------------------------------------------------------------

多态：

多态分为**编译时多态**和**运行时多态**:

* 编译时多态主要指**方法的重载**
* 运行时多态指程序中定义的对象引用所指向的具体类型在运行期间才确定（具体表现为父类的引用指向子类的实例。）

运行时多态有三个条件:

* 继承
* 覆盖(重写)
* 向上转型

优点：提高了软件的灵活性和可扩展性

===========================================================

创建一个对象用什么运算符?对象实体与对象引用有何不同?

**new 运算符**，new 创建对象实例（对象实例在堆内存中），对象引用指向对象实例（对象引用存放在栈内存中）。

对象的相等与指向他们的引用相等,两者有什么不同?

* 对象的相等一般比较的是内存中存放的内容是否相等。
* 引用相等一般比较的是他们指向的内存地址是否相等。

静态方法为什么不能调用非静态成员?

* 静态方法是属于类的，在类加载的时候就会分配内存，可以通过类名直接访问。而非静态成员属于实例对象，只有在对象实例化之后才存在，需要通过类的实例对象去访问。
* 在类的非静态成员不存在的时候静态成员就已经存在了，此时调用在内存中还不存在的非静态成员，属于非法操作。

静态方法和实例方法有何不同？

1. 调用方式

在外部调用静态方法时，可以使用 **类名.方法名** 的方式（建议），也可以使用 对象.方法名 的方式（不建议）

而实例方法**只能**用 **对象.方法名** 来调用。

1. 访问类成员是否存在限制

静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），不允许访问实例成员（即实例成员变量和实例方法）

而实例方法不存在这个限制。

接口和抽象类的共同点和区别？

共同点：

* 都不能被实例化。
* 都可以包含抽象方法。

区别：

* 一个子类只能继承一个抽象类, 但能实现多个接口
* 抽象类可以有构造方法, 接口没有构造方法
* 抽象类可以有普通成员变量, 接口没有普通成员变量
* 抽象类和接口都可有静态成员变量, 抽象类中静态成员变量访问类型任意，接口只能public static final(默认)
* **抽象类可以没有抽象方法, 抽象类可以有普通方法；接口在JDK8之前都是抽象方法，在JDK8可以有default方法，在JDK9中允许有私有普通方法**
* **抽象类可以有静态方法；接口在JDK8之前不能有静态方法，在JDK8中可以有静态方法，且只能被接口类直接调用（不能被实现类的对象调用）**
* **抽象类中的方法可以是public、protected; 接口方法在JDK8之前只有public abstract，在JDK8可以有default方法，在JDK9中允许有private方法**

**抽象类：可以没抽象方法，可以有普通方法、静态方法；权限：可以是public、protected**

**接口：jdk8之前都是抽象方法，jdk8之后可以有静态方法，jdk9可以有（私有）普通方法；权限：jdk8之前只有public abstract，jdk8有default，jdk9可以有private**

this() & super()在构造方法中的区别？

* 调用super()必须写在子类构造方法的第一行, 否则编译不通过
* super从子类调用父类构造, this在同一类中调用其他构造均需要放在第一行
* 尽管可以用this调用一个构造器, 却不能调用2个
* this和super不能出现在同一个构造器中, 否则编译不通过
* this()、super()都指的对象,不可以在static环境中使用
* 本质this指向本对象的指针。super是一个关键字

======================================================

重载与重写

重载（overload）：

重载是什么？

在同一个类中（或者父类和子类之间），可以有多个方法名相同，参数列表不同（类型不同、个数不同、顺序不同）的方法。此外，方法返回值和访问修饰符可以不同。

注意：

* **构造方法能够被重载（构造方法能重载，不能重写）**
* 对继承来说，如果某个方法在父类用private修饰，那么子类就不能对其进行重载。如果在子类中定义的话，也只是定义了一个新的方法，不会起到重载效果。

重写是什么？（重点）

重写发生在**运行期**，是子类对父类的允许访问的方法的实现过程进行重新编写。

* **方法名、参数列表必须相同，子类方法返回值类型应比父类方法返回值类型更小或相等，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类**。
* 如果父类方法访问修饰符为 **private/final/static** 则子类就不能重写该方法，但是被 static 修饰的方法能够被再次声明。
* **构造方法无法被重写（因为构造器不能被继承，所以不能被重写）**

综上：**重写就是子类对父类方法的重新改造，外部样子不能改变，内部逻辑可以改变。**

方法的重写遵从“**两同两小一大**“

* “两同”即**方法名相同、形参列表相同**；
* “两小”指的是子类方法返回值类型应比父类方法**返回值类型更小或相等**，子类方法声明抛出的异常类应比父类方法声明**抛出的异常类更小或相等**；
* “一大”指的是子类方法的访问权限应比父类方法的**访问权限更大或相等**。

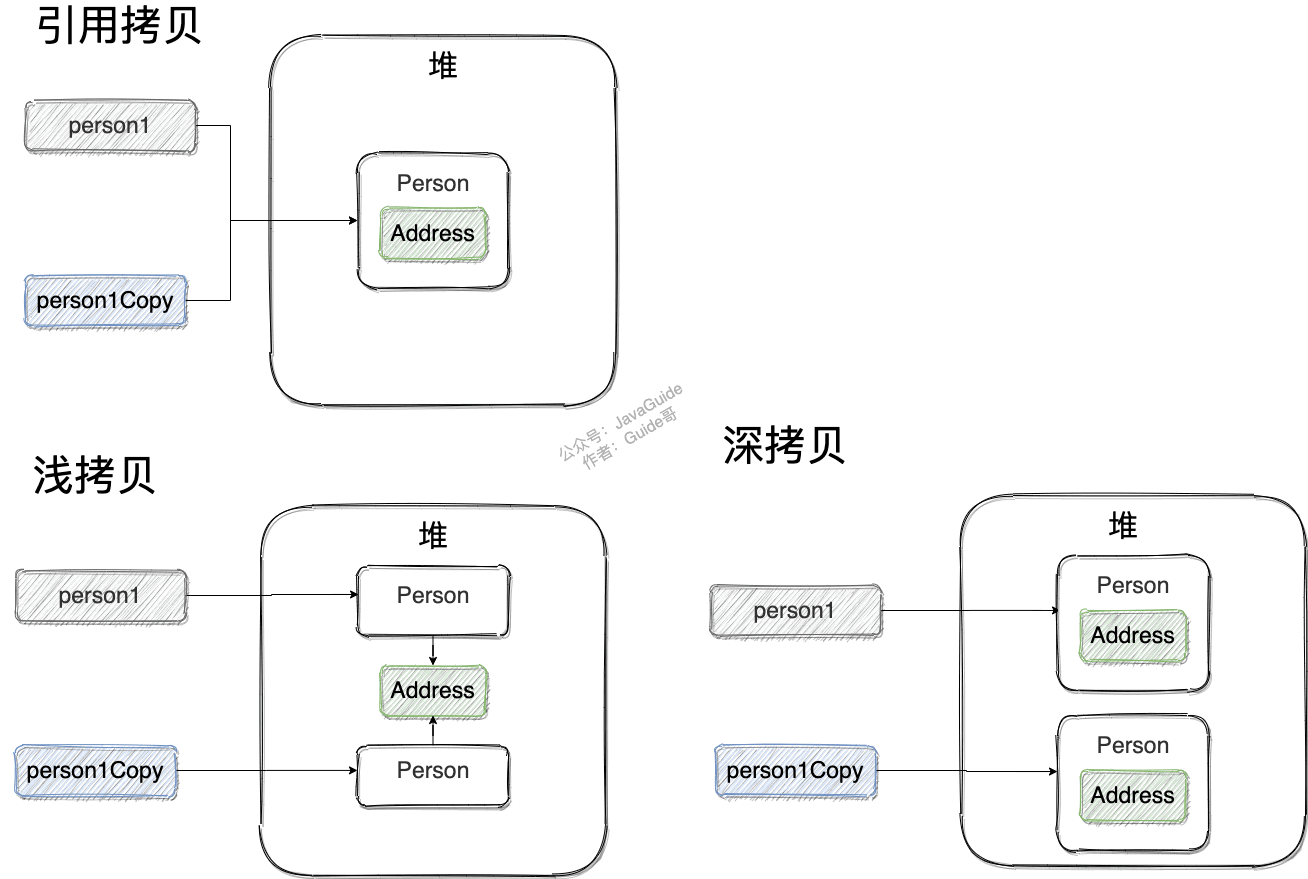
但注意：如果方法的返回类型是 void 和基本数据类型，则返回值重写时不可修改。但是如果方法的返回值是引用类型，重写时是可以返回该引用类型的子类的。

重载和重写的区别：



深拷贝和浅拷贝区别了解吗？

* **浅拷贝**：浅拷贝会在堆上创建一个新的对象（区别于引用拷贝的一点），不过，如果原对象内部的属性是引用类型的话，浅拷贝会直接复制内部对象的引用地址，也就是说拷贝对象和原对象共用同一个内部对象。
* **深拷贝** ：深拷贝会完全复制整个对象，包括这个对象所包含的内部对象。



Object类---是所有类的父类

equals（）方法要点

==和equals（）的区别：

（1）== 对于基本类型和引用类型的作用效果是不同的：

* 对于基本数据类型来说，== 比较的是值。
* 对于引用数据类型来说，== 比较的是对象的内存地址。

注意：Java只有值传递，基本数据类型装的是数据值，引用数据类型装的是对象所在内存地址值。

（2）equals() 不能用于判断基本数据类型的变量，只能用来判断两个对象是否相等。equals()方法存在于Object类中，而Object类是所有类的直接或间接父类，因此所有的类都有equals()方法。

equals() 方法存在两种使用情况：

* **类没有重写 equals()方法** ：通过equals()比较该类的两个对象时，等价于通过“==”比较这两个对象，使用的默认是 Object类equals()方法。
* **类重写了 equals()方法** ：一般我们都重写 equals()方法来比较两个对象中的属性是否相等；若它们的属性相等，则返回 true(即，认为这两个对象相等)。

注意：String 中的 equals 方法是被重写过的，因为 Object 的 equals 方法是比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值。

--------------------------------------------------------------------------

hashCode（）方法要点

hashCode() 有什么用？

hashCode() 的作用是获取哈希码（int 整数），也称为散列码。这个哈希码的作用是确定该对象在哈希表中的索引位置。

hashCode()定义在 JDK 的 Object 类中，这就意味着 Java 中的任何类都包含有 hashCode() 函数。

**那为什么两个对象有相同的 hashCode 值，它们也不一定是相等的？**

总结下来就是 ：

* 如果两个对象的hashCode 值相等，那这两个对象不一定相等（哈希碰撞）。
* 如果两个对象的hashCode 值相等并且equals()方法也返回 true，才认为这两个对象相等。
* 如果两个对象的hashCode 值不相等，我们就可以直接认为这两个对象不相等。

为什么重写 equals() 时必须重写 hashCode() 方法？

因为两个相等的对象的 hashCode 值必须是相等。也就是说如果 equals 方法判断两个对象是相等的，那这两个对象的 hashCode 值也要相等。

如果重写 equals() 时没有重写 hashCode() 方法的话就可能会导致 equals 方法判断是相等的两个对象，hashCode 值却不相等。

总结 ：

* equals 方法判断两个对象是相等的，那这两个对象的 hashCode 值也要相等。
* 两个对象有相同的 hashCode 值，他们也不一定是相等的（哈希碰撞）。

==========================================================

==========================================================

字符串知识！！！

String、StringBuffer与StringBuilder的区别？

**第一点: 可变和适用范围。**

String对象是不可变的，而StringBuffer和StringBuilder是可变字符序列。

每次对String的操作相当于生成一个新的String对象，而对StringBuffer和StringBuilder的操作是对对象本身的操作，而不会生成新的对象，所以对于频繁改变内容的字符串避免使用String，因为频繁的生成对象将会对系统性能产生影响。

StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串，不过没有使用 final 和 private 关键字修饰，最关键的是这个 AbstractStringBuilder 类还提供了很多修改字符串的方法比如 append 方法。

**第二点: 线程安全。**

String由于有final修饰（实际上是因为private），是不可变的，是线程安全的。StringBuilder不保证同步所以是非线程安全的， StringBuffer在每个方法中加了synchronized同步锁所以是线程安全的。

按照应用场景可以这样用：

字符串修改和拼接次数少可以用---String

单线程下操作字符串可以用---StringBuilder----（效率更高）

多线程下操作字符串可以用---StringBuffer----（保证线程安全）

注意：

1. 三者在执行速度方面的比较：StringBuilder > StringBuffer > String
2. String类重写了equals和hashcode方法；StringBuilder 和 StingBuffer 不重写equals和hashcode方法，所以将StringBuffer对象存储到java集合中会出问题。

**String为什么不可变？（重点）**

---表面上是因为String 类中使用 final 关键字修饰字符数组来保存字符串，实际上

String 真正不可变有两点原因：

* 保存字符串的数组被 final 修饰且为私有的，并且String 类没有提供/暴露修改这个字符串的方法。
* String 类被 final 修饰导致其不能被继承，进而避免了子类破坏 String 不可变。

字符串拼接用“+” 还是 StringBuilder?

Java 语言本身并不支持运算符重载，“+”和“+=”是专门为 String 类重载过的运算符，也是 Java 中**仅有的两个重载过的元素符**。**String对象引用和“+”的字符串拼接方式，实际上是通过 StringBuilder 调用 append() 方法实现的，拼接完成之后调用 toString() 得到一个 String 对象 。**不过，在循环内使用“+”进行字符串的拼接的话，存在比较明显的缺陷：编译器不会创建单个 StringBuilder 以复用，会导致创建过多的 StringBuilder 对象。StringBuilder 对象是在循环内部被创建的，这意味着每循环一次就会创建一个 StringBuilder 对象。

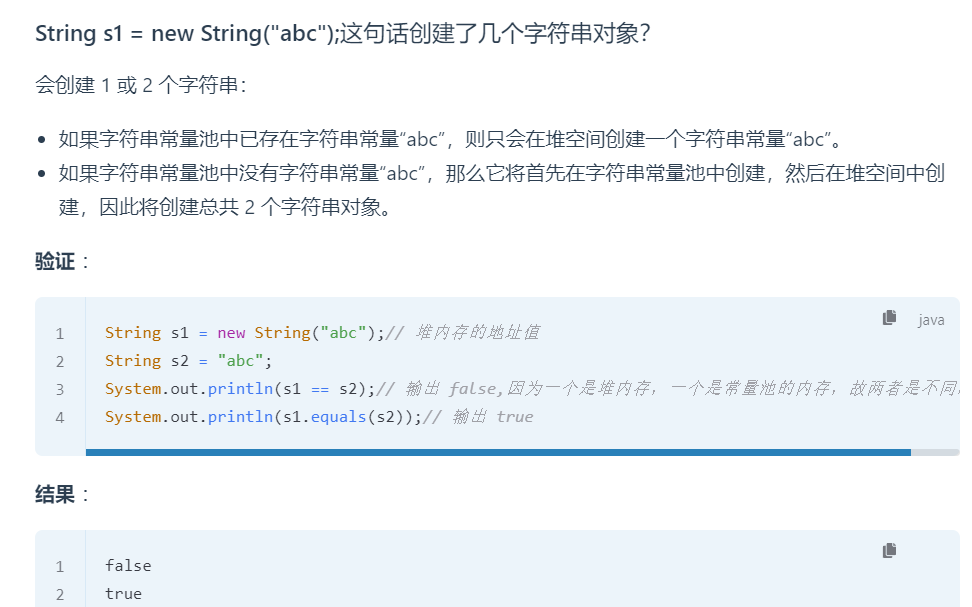
**直接使用 StringBuilder 对象进行字符串拼接**的话，就不会存在这个问题了。

注意：通常是：确定值.equals(变量)，而最好别用 变量.equals(值)；确定值.equals(变量)可以避免空指针异常！！！

------------------------------------------------------------------------

字符串常量池的作用了解吗？

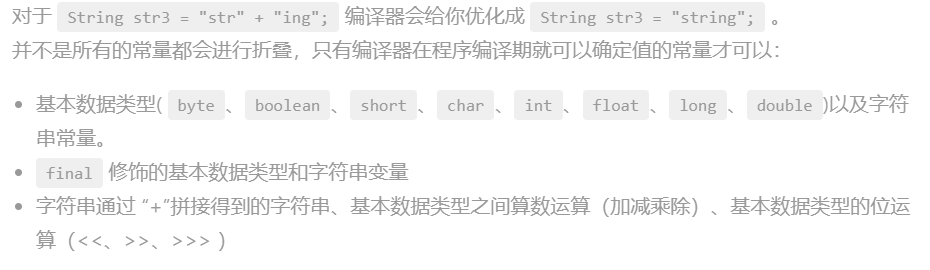
字符串常量池 是 JVM 为了提升性能和减少内存消耗针对字符串（String 类）专门开辟的一块区域，主要目的是为了避免字符串的重复创建。**JDK1.7 之前运行时常量池逻辑包含字符串常量池存放在方法区**。**JDK1.7 的时候，字符串常量池被从方法区拿到了堆中**。

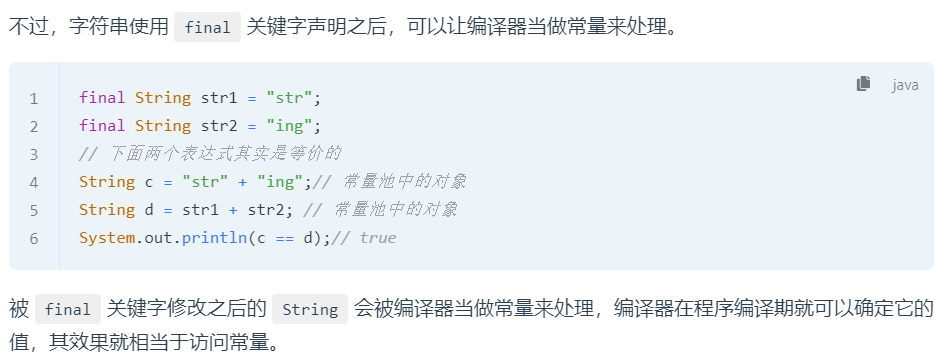


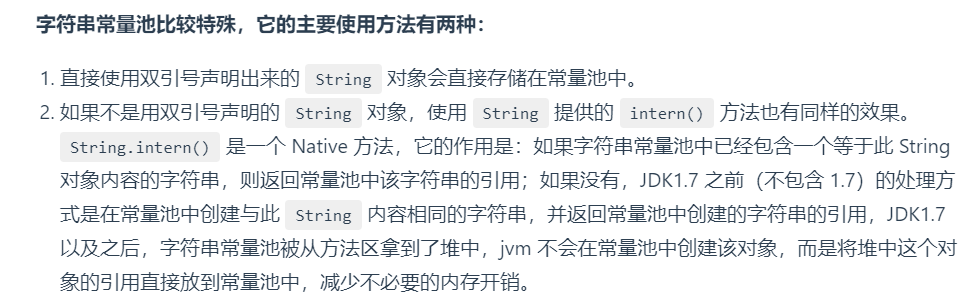
String 类型的变量和常量做“+”运算时发生了什么？



对于编译期可以确定值的字符串，也就是常量字符串 ，jvm 会将其存入字符串常量池。字符串常量池 是 JVM 为了提升性能和减少内存消耗针对字符串（String 类）专门开辟的一块区域，主要目的是为了避免字符串的重复创建。**JDK1.7 之前运行时常量池逻辑包含字符串常量池存放在方法区。JDK1.7 的时候，字符串常量池被从方法区拿到了堆中。**并且，**字符串常量拼接得到的字符串常量在编译阶段就已经被存放字符串常量池**，这个得益于编译器的优化。所谓的优化就叫“**常量折叠**“--**常量折叠会把常量表达式的值求出来作为常量嵌在最终生成的代码中。**







字符串总结：

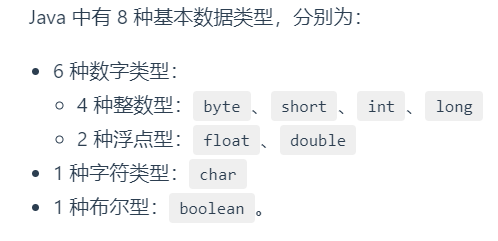
1. 对于基本数据类型来说，==比较的是值。对于引用数据类型来说，==比较的是对象的内存地址。
2. 在编译过程中，Javac 编译器（下文中统称为编译器）会进行一个叫做 常量折叠(Constant Folding) 的代码优化。常量折叠会把常量表达式的值求出来作为常量嵌在最终生成的代码中，这是 Javac 编译器会对源代码做的极少量优化措施之一(代码优化几乎都在即时编译器中进行)。
3. 一般来说，我们要尽量避免通过 new 的方式创建字符串。使用双引号声明的 String 对象（ String s1 = "java" ）更利于让编译器有机会优化我们的代码，同时也更易于阅读。
4. 被 final 关键字修改之后的 String 会被编译器当做常量来处理，编译器程序编译期就可以确定它的值，其效果就相当于访问常量。

===========================================================

===========================================================

基本数据类型

基本数据类型是什么？



注意：

1. Java 里使用 long 类型的数据一定要在数值后面加上 L，否则将作为整型解析。
2. char a = 'h' char :单引号，String a = "hello" :双引号。

这八种基本类型都有对应的包装类分别为：Byte、Short、Integer、Long、Float、Double、Character、Boolean 。

基本数据类型和包装类的区别

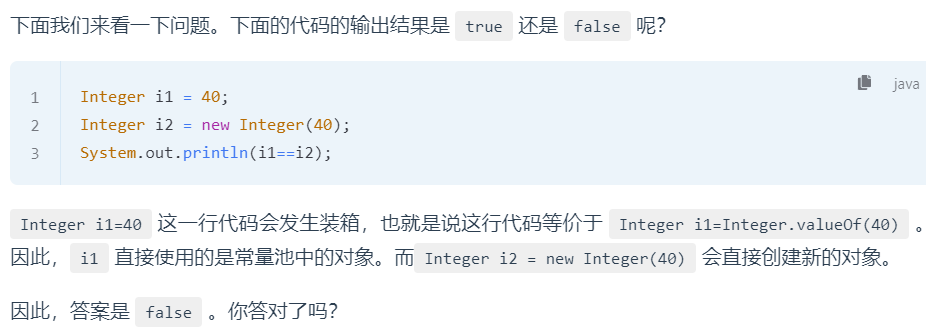
* 包装类型不赋值就是 null ，而基本类型有默认值且不是 null。
* 包装类型可用于泛型，而基本类型不可以。
* 基本数据类型的**局部变量**存放在 Java 虚拟机栈中的局部变量表中，基本数据类型的**成员变量（未被 static 修饰 ）**存放在 Java 虚拟机的堆中（包装类型属于对象类型，对象存放在堆内存中）。注意 ： **基本数据类型存放在栈中是一个常见的误区！** 基本数据类型的成员变量如果没有被 static 修饰的话（不建议这么使用，应该要使用基本数据类型对应的包装类型），就存放在堆中。
* 相比于对象类型， 基本数据类型占用的空间非常小。

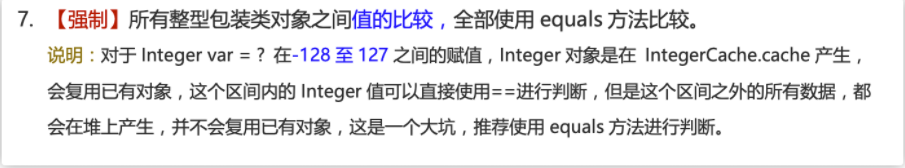
包装类型的常量池技术了解么？

Java 基本类型的包装类的大部分（除了两个浮点型）都实现了**常量池技术**。

Byte, Short, Integer, Long 这 4 种包装类默认创建了数值 [-128，127] 的相应类型的缓存数据，Character 创建了数值在 [0,127] 范围的缓存数据，Boolean 直接返回 True or False。

两种浮点数类型的包装类 Float, Double **并没有**实现常量池技术。



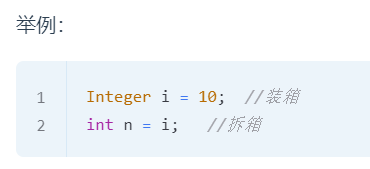


记住：**所有整型包装类对象之间值的比较，全部使用 equals 方法比较**。

自动装箱与拆箱了解吗？原理是什么？

什么是自动拆装箱？

* 装箱：将基本类型用它们对应的引用类型包装起来；
* 拆箱：将包装类型转换为基本数据类型；



装箱其实就是调用了 包装类的valueOf()方法，拆箱其实就是调用了 xxxValue()方法。

因此，

* Integer i = 10 等价于 Integer i = Integer.valueOf(10)
* int n = i 等价于 int n = i.intValue();

注意：**如果频繁拆装箱的话，也会严重影响系统的性能。我们应该尽量避免不必要的拆装箱操作。**

===========================================================

泛型

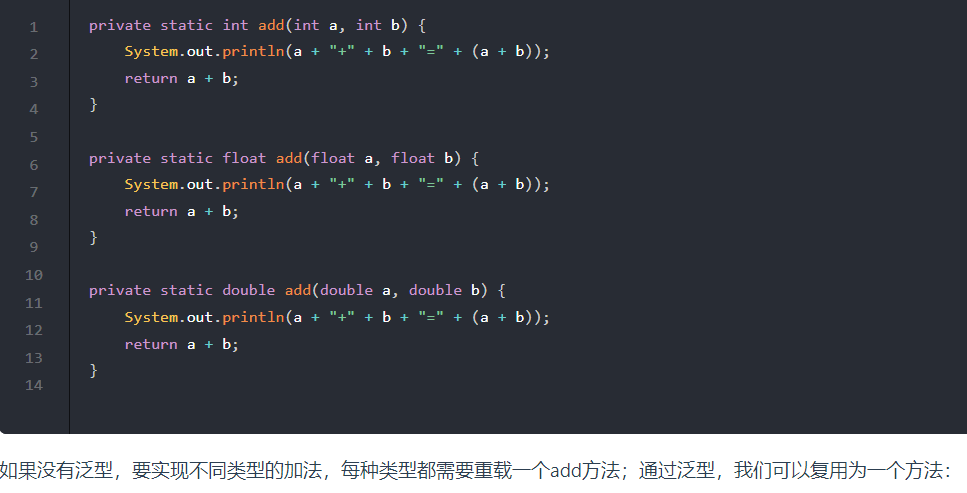
泛型是什么？

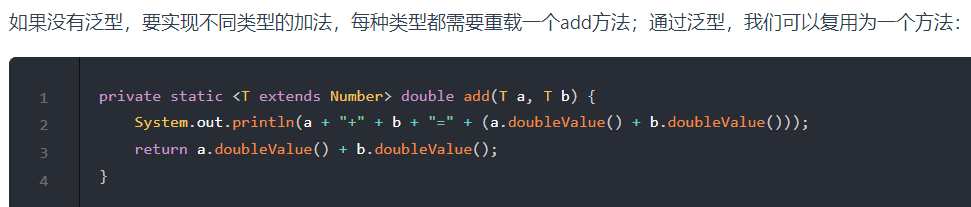
Java 泛型（generics） 是 **JDK 5** 中引入的一个新特性, 泛型提供了**编译时类型安全检测机制**，该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

为什么需要泛型？

1. **适用于多种数据类型执行相同的代码**

泛型中的类型在使用时指定，不需要强制类型转换（类型安全，编译器会检查类型）

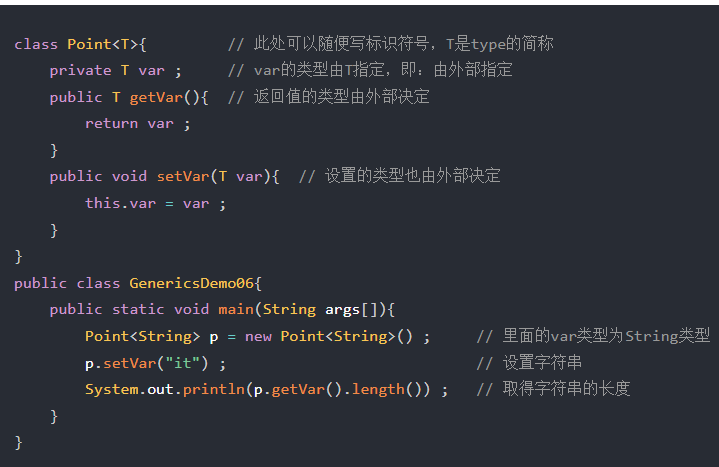




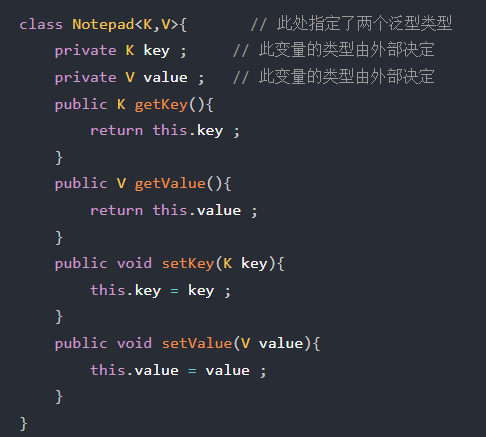


泛型的三种使用方式：泛型类、泛型接口、泛型方法

1. 泛型类如何定义使用？
   1. 简单泛型类



（2）多元泛型



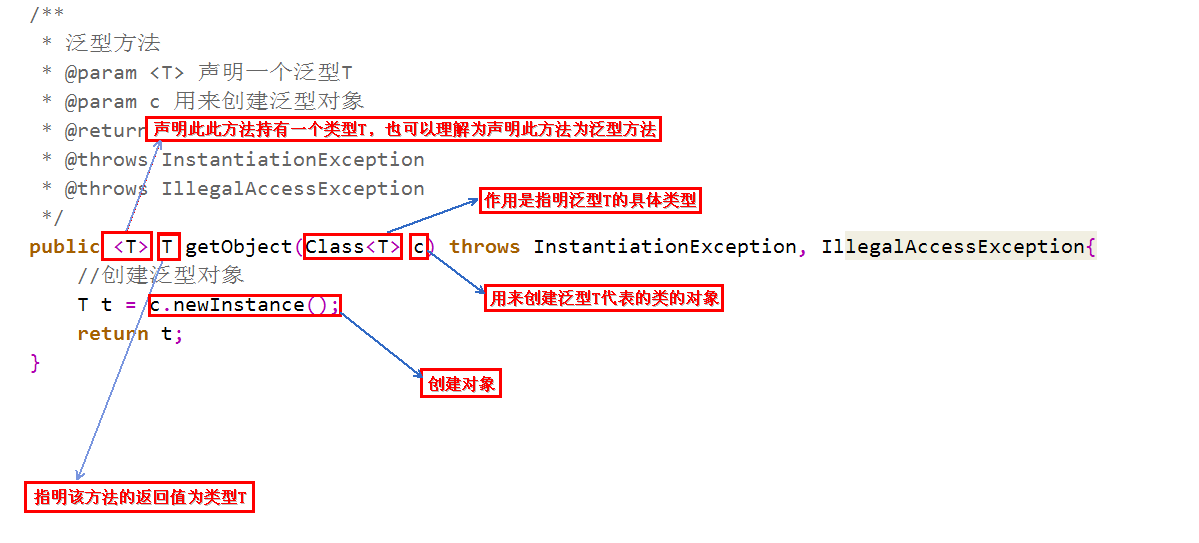
1. 泛型接口如何定义使用



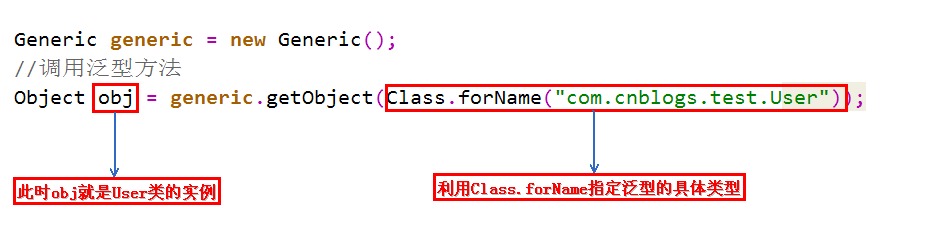
1. 泛型方法如何定义使用

泛型方法，是在调用方法的时候指明泛型的具体类型。

定义泛型方法的语法格式：



调用泛型方法的语法格式：



说明一下，定义泛型方法时，必须在返回值前边加一个<T>，来声明这是一个泛型方法，持有一个泛型T，然后才可以用泛型T作为方法的返回值。

Class<T>的作用就是指明泛型的具体类型，而Class<T>类型的变量c，可以用来创建泛型类的对象。

为什么要用变量c来创建对象呢？既然是泛型方法，就代表着我们不知道具体的类型是什么，也不知道构造方法如何，因此没有办法去new一个对象，但可以利用变量c的newInstance方法去创建对象，也就是利用反射创建对象。

泛型方法要求的参数是Class<T>类型，而Class.forName()方法的返回值也是Class<T>，因此可以用Class.forName()作为参数。其中，forName()方法中的参数是何种类型，返回的Class<T>就是何种类型。在本例中，forName()方法中传入的是User类的完整路径，因此返回的是Class<User>类型的对象，因此调用泛型方法时，变量c的类型就是Class<User>，因此泛型方法中的泛型T就被指明为User，因此变量obj的类型为User。

当然，泛型方法不是仅仅可以有一个参数Class<T>，可以根据需要添加其他参数。

**所以，为什么要使用泛型方法呢？**----因为泛型类要在实例化的时候就指明类型，如果想换一种类型，不得不重新new一次，可能不够灵活；而泛型方法可以在调用的时候指明类型，更加灵活。

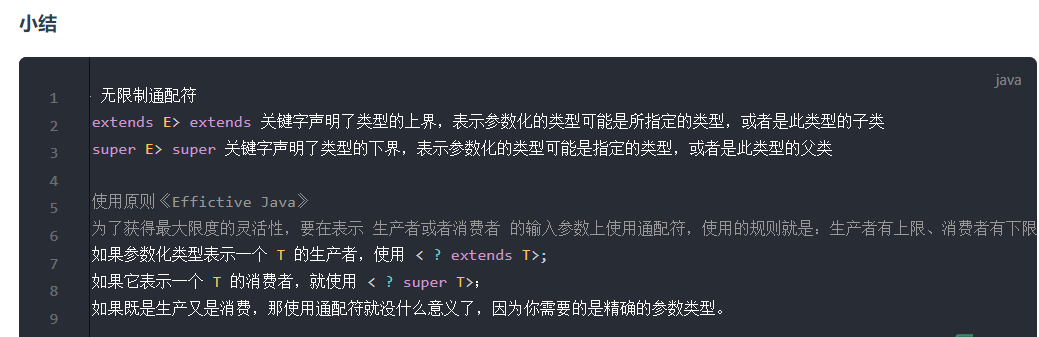
泛型的上限和下限？

在使用泛型的时候，我们可以为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。

注意： ？ 可在使用泛型时代表一切类型

上下限：

1. **？ extends Car ------？必须为Car或其子类，泛型上限**
2. **？ super Car ----- ？ 必须是Car或其父类，泛型下限**



**如何理解Java中的泛型是伪泛型？**

Java泛型这个特性是从**JDK 1.5**才开始加入的，因此为了兼容之前的版本，Java泛型的实现采取了“**伪泛型**”的策略，即**Java在语法上支持泛型，但是在编译阶段会进行所谓的“类型擦除**”（Type Erasure），**将所有的泛型表示（尖括号中的内容）都替换为具体的类型**（其对应的原生态类型），就像完全没有泛型一样。

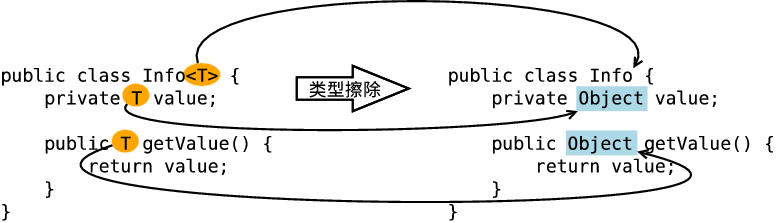
泛型的类型擦除原则是：

* 消除类型参数声明，即删除<>及其包围的部分。
* **根据类型参数的上下界推断并替换所有的类型参数为原生态类型：如果类型参数是无限制通配符或没有上下界限定则替换为Object，如果存在上下界限定则根据子类替换原则取类型参数的最左边限定类型（即父类）。**
* 为了保证类型安全，必要时插入强制类型转换代码。
* 自动产生“桥接方法”以保证擦除类型后的代码仍然具有泛型的“多态性”。

那么如何进行擦除的呢？

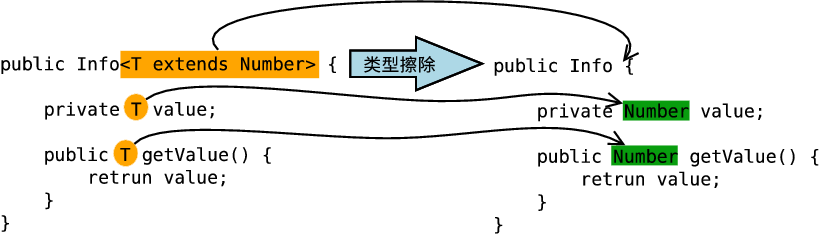
* 擦除类定义中的类型参数---无限制类型擦除

当类定义中的类型参数没有任何限制时，在类型擦除中直接被替换为Object，即形如<T>和<?>的类型参数都被替换为Object。



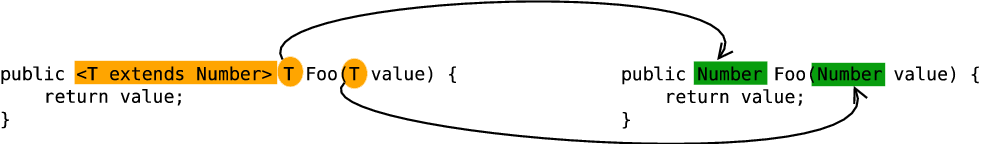
* 擦除类定义中的类型参数---有限制类型擦除

当类定义中的类型参数存在限制（上下界）时，在类型擦除中替换为类型参数的上界或者下界，比如形如<T extends Number>和<? extends Number>的类型参数被替换为Number，<? super Number>被替换为Object。



* 擦除方法定义中的类型参数

擦除方法定义中的类型参数原则和擦除类定义中的类型参数是一样的，这里仅以擦除方法定义中的有限制类型参数为例。



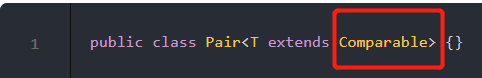
注意：虽然有泛型对数据类型进行限制，可是仍可以通过反射添加其它类型元素

----------------------------------------------------------------

如何理解类型擦除后保留的原始类型？

**原始类型 就是擦除去了泛型信息，最后在字节码中的类型变量的真正类型**，无论何时定义一个泛型，相应的原始类型都会被自动提供，类型变量擦除，并使用其限定类型（无限定的变量用Object）替换。



而Pair这样声明的话：

那么原始类型就是Comparable。

在调用泛型方法时，可以指定泛型，也可以不指定泛型:

* 在不指定泛型的情况下，泛型变量的类型为该方法中的几种类型的同一父类的最小级，直到Object
* 在指定泛型的情况下，该方法的几种类型必须是该泛型的实例的类型或者其子类

----------------------------------------------------------------

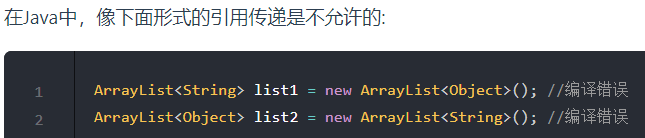
如何理解泛型的编译期检查？既然说类型变量会在编译的时候擦除掉，那为什么我们往 ArrayList 创建的对象中添加整数会报错呢？

🡪 Java编译器是通过**先检查代码中泛型的类型，然后在进行类型擦除，再进行编译**。

另外，**类型检查就是针对引用的**，谁是一个引用，用这个引用调用泛型方法，就会对这个引用调用的方法进行类型检测，而无关它真正引用的对象。

-------------------------------------------------------------

泛型中参数话类型为什么不考虑继承关系？



如何理解泛型的多态？泛型的桥接方法？

->**类型擦除会造成多态的冲突，而JVM解决方法就是桥接方法**。虚拟机巧妙的使用了桥方法，来解决了类型擦除和多态的冲突。

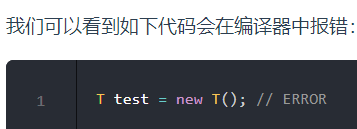
如何理解基本类型不能作为泛型类型？比如，我们没有ArrayList<int>，只有ArrayList<Integer>, 为何？

->**因为当类型擦除后，ArrayList的原始类型变为Object，但是Object类型不能存储int值，只能引用Integer的值。**

另外需要注意，我们能够使用list.add(1)是因为**Java基础类型的自动装箱拆箱操作**。

如何理解泛型类型不能实例化？

不能实例化泛型类型, 这本质上是由于类型擦除决定的

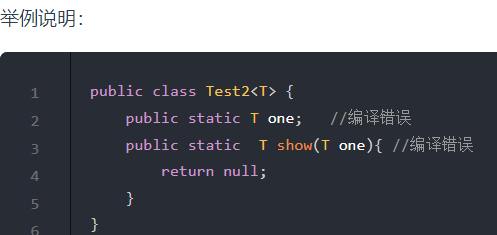


**因为在 Java 编译期没法确定泛型参数化类型，也就找不到对应的类字节码文件**，所以自然就不行了，**此外由于T 被擦除为 Object，如果可以 new T() 则就变成了 new Object()，失去了本意。**     如果我们确实需要实例化一个泛型，应该如何做呢？可以通过**反射**实现

---------------------------------------------------------------------------

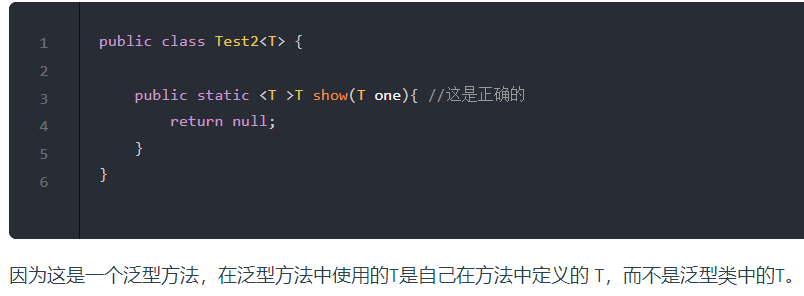
如何理解泛型类中的静态方法和静态变量？

**泛型类中的静态方法和静态变量不可以使用泛型类所声明的泛型类型参数**



**因为泛型类中的泛型参数的实例化是在定义对象的时候指定的，而静态变量和静态方法不需要使用对象来调用。对象都没有创建，如何确定这个泛型参数是何种类型**，当然是错误的。

但是要注意区分下面的一种情况：

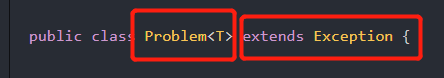


--------------------------------------------------------------------

如何理解异常中使用泛型？

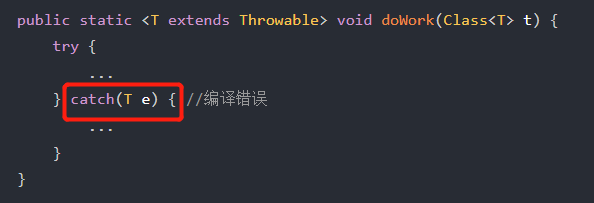
* **不能抛出也不能捕获泛型类的对象**。

事实上，泛型类扩展Throwable都不合法。例如：下面的定义将不会通过编译：



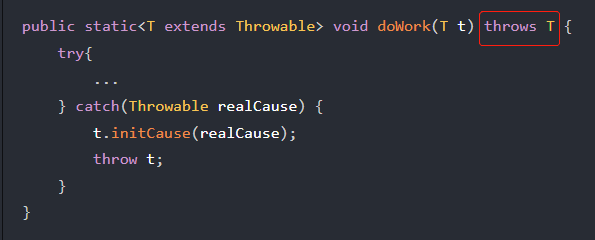
为什么不能扩展Throwable，因为异常都是在**运行时**捕获和抛出的，而在编译的时候，泛型信息全都会被擦除掉

* **不能在catch子句中使用泛型变量**



因为泛型信息在编译的时候已经变为原始类型，也就是说上面的T会变为原始类型Throwable，那么如果可以再catch子句中使用泛型变量。根据异常捕获的原则，一定是子类在前面，父类在后面，那么上面就违背了这个原则。

* **但是在异常声明中可以使用类型变量。**



如何获取泛型的参数类型？既然类型被擦除了，那么如何获取泛型的参数类型呢？

->可以通过反射（java.lang.reflect.Type）获取泛型。

（了解：java.lang.reflect.Type是Java中所有类型的公共高级接口, 代表了Java中的所有类型. Type体系中类型的包括：数组类型(GenericArrayType)、参数化类型(ParameterizedType)、类型变量(TypeVariable)、通配符类型(WildcardType)、原始类型(Class)、基本类型(Class), 以上这些类型都实现Type接口。）

==========================================================

========================================================

注解

注解是什么？

注解是JDK1.5版本开始引入的一个特性，用于对代码进行说明，可以对包、类、接口、字段、方法参数、局部变量等进行注解。

注解的作用：

主要的作用有以下四方面：

* **生成文档**，通过代码里标识的元数据生成javadoc文档。
* **编译检查**，通过代码里标识的元数据让编译器在编译期间进行检查验证。
* **编译时动态处理**，编译时通过代码里标识的元数据动态处理，例如**动态生成代码**。
* **运行时动态处理**，运行时通过代码里标识的元数据动态处理，例如**使用反射注入实例**。

注解的常见分类：

1. Java自带的标准注解，包括**@Override**、**@Deprecated**和**@SuppressWarnings**，分别用于标明重写某个方法；标明某个类或方法过时；标明要忽略的警告；用这些注解标明后编译器就会进行检查。
2. 元注解，元注解是用于定义注解的注解，包括@Retention、@Target、@Inherited、@Documented

* @Retention用于标明注解被保留的阶段（三种策略定义在RetentionPolicy中）
* @Target用于标明注解使用的范围 （取值范围定义在ElementType枚举中）
* @Inherited用于标明子类是否可继承父类中的注解（注解本身不支持继承）（父类标注，子类自动继承该注解）
* @Documented用于标明是否生成javadoc文档（用javadoc为类生成帮助文档时，是否保留注解信息）

---java8中的元注解：

* @Repeatable 重复注解，允许在同一申明类型(类，属性，或方法)的多次使用同一个注解，与java8之前的重复使用注解的方案相比，增强了可读性
* @Native：使用 @Native 注解修饰成员变量，则表示这个变量可以被本地代码引用，常常被代码生成工具使用。

1. 自定义注解，可以根据自己的需求定义注解，并可用元注解对自定义注解进行注解。

注解与反射接口，定义注解后，如何获取注解中的内容呢？

反射包java.lang.reflect下的AnnotatedElement接口提供这些方法。这里注意：只有注解被定义为RUNTIME后，该注解才能是运行时可见，当class文件被装载时被保存在class文件中的Annotation才会被虚拟机读取。

AnnotatedElement 接口是所有程序元素（Class、Method和Constructor）的父接口，所以程序通过反射获取了某个类的AnnotatedElement对象之后，程序就可以调用该对象的方法来访问Annotation信息。

===========================================================

===========================================================

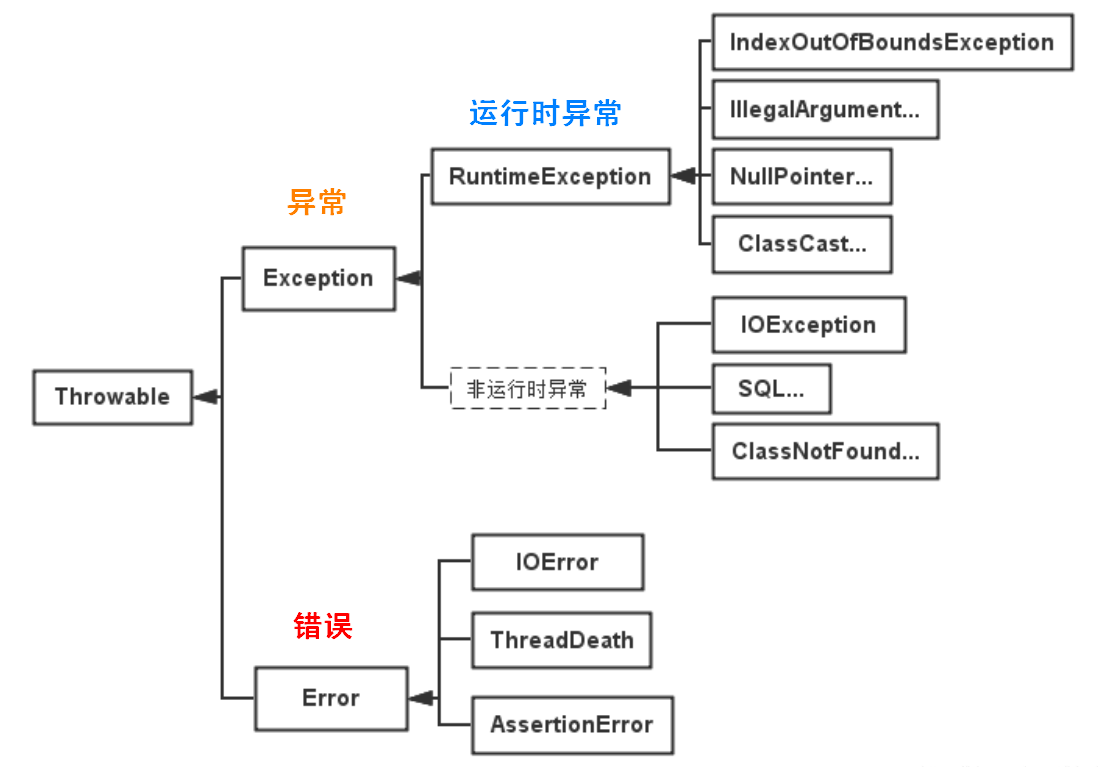
异常

Java异常类层次结构？

Throwable 是 Java 语言中所有错误与异常的超类。

* Error 类及其子类：程序中无法处理的错误，表示运行应用程序中出现了严重的错误。
* Exception 程序本身可以捕获并且可以处理的异常。Exception 这种异常又分为两类：运行时异常RuntimeException 和 编译时异常。

注意：运行时异常的特点是Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch语句捕获它，也没有用throws子句声明抛出它，也会编译通过。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。



可查的异常（checked exceptions）和不可查的异常（unchecked exceptions）区别？

可查异常---受检异常---编译时异常---编译器要求必须处理的异常

->处理：要么用try-catch语句捕获它，要么用throws子句声明抛出，否则编译不通过

不可查异常---非受检异常---运行时异常+错误Error---编译器不要求强制处理的异常

->不要求处理

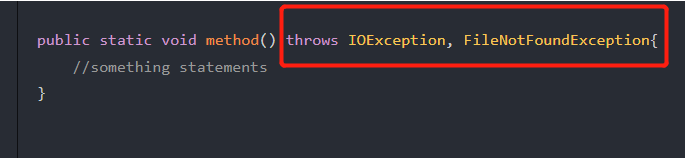
异常关键字

* try – 用于监听。将要被监听的代码(可能抛出异常的代码)放在try语句块之内，当try语句块内发生异常时，异常就被抛出。
* catch – 用于捕获异常。catch用来捕获try语句块中发生的异常。
* finally – finally语句块总是会被执行。它主要用于回收在try块里打开的物力资源(如数据库连接、网络连接和磁盘文件)。只有finally块，执行完成之后，才会回来执行try或者catch块中的return或者throw语句，如果finally中使用了return或者throw等终止方法的语句，则就不会跳回执行，直接停止。
* throw – 用于抛出异常。
* throws – 用在方法签名中，用于声明该方法可能抛出的异常。

throw和throws的区别？

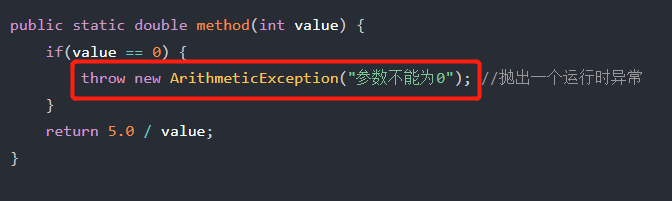
* 异常的申明---throws

在Java中，当前执行的语句必属于某个方法，Java解释器调用main方法执行开始执行程序。若方法中存在检查异常，如果不对其捕获，那必须在方法头中显式声明该异常，以便于告知方法调用者此方法有异常，需要进行处理。 在方法中声明一个异常，方法头中使用**关键字throws**，后面接上要声明的异常。若声明多个异常，则使用**逗号分割**。



* 异常的抛出---throw

如果代码可能会引发某种错误，可以**创建**一个合适的**异常类实例**并**抛出**它，这就是抛出异常。

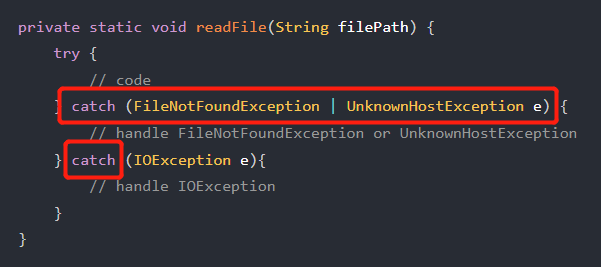


异常捕获处理的方法通常有：

* try-catch
* try-catch-finally
* try-finally
* try-with-resource

1. 了解try-catch？

在一个 try-catch 语句块中可以捕获多个异常类型，并对不同类型的异常做出不同的处理。同一个 catch 也可以捕获多种类型异常，用 | 隔开。



Try可以跟多个catch

1. 了解try-catch-finally？

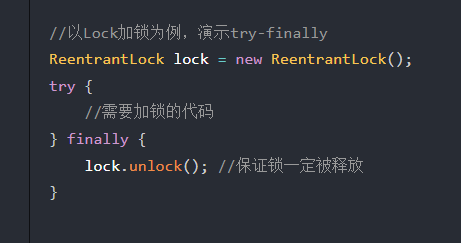
执行的顺序 ---分三种情况

* 当try没有捕获到异常时：try语句块中的语句逐一被执行，程序将跳过catch语句块，执行finally语句块和其后的语句；
* 当try捕获到异常，catch语句块里没有处理此异常的情况：当try语句块里的某条语句出现异常时，而没有处理此异常的catch语句块时，此异常将会抛给JVM处理，finally语句块里的语句还是会被执行，但finally语句块后的语句不会被执行；
* 当try捕获到异常，catch语句块里有处理此异常的情况：在try语句块中是按照顺序来执行的，当执行到某一条语句出现异常时，程序将跳到catch语句块，并与catch语句块逐一匹配，找到与之对应的处理程序，其他的catch语句块将不会被执行，而try语句块中，出现异常之后的语句也不会被执行，catch语句块执行完后，执行finally语句块里的语句，最后执行finally语句块后的语句；

1. 了解try-finally吗？

**try块中引起异常，异常代码之后的语句不再执行，直接执行finally语句。** try块没有引发异常，则执行完try块就执行finally语句。

**try-finally可用在不需要捕获异常的代码，可以保证资源在使用后被关闭**。例如IO流中执行完相应操作后，关闭相应资源；使用Lock对象保证线程同步，通过finally可以保证锁会被释放；数据库连接代码时，关闭连接操作等等。

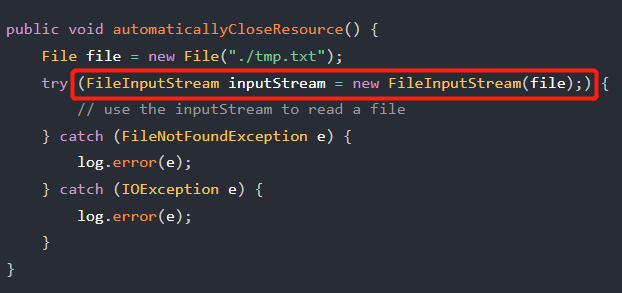


finally遇见如下情况不会执行

* 在前面的代码中用了System.exit()退出程序。
* finally语句块中发生了异常。
* 程序所在的线程死亡。
* 关闭CPU。

1. 了解Java 7 的 try-with-resource吗?---在java7中引入

大多数的java标准资源都继承了AutoCloseable接口，当用的资源继承了这个接口，就可以用try-with-resource这个语法。当你在 try 子句（try后面的圆括号）中打开资源，资源会在 try 代码块执行后或异常处理后自动关闭。



--------------------------------------------------------------------

异常基础总结

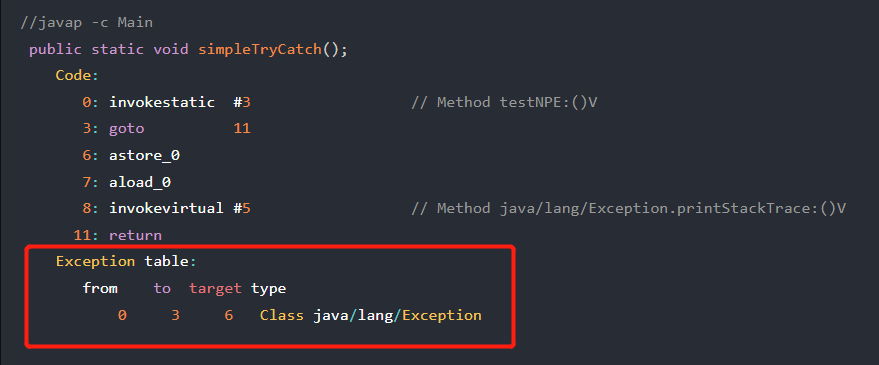
* try、catch和finally都不能单独使用，只能是try-catch、try-finally或者try-catch-finally。
* try语句块监控代码，出现异常就停止执行下面的代码，然后将异常移交给catch语句块来处理。
* finally语句块中的代码一定会被执行，常用于回收资源 。
* throws：声明一个异常，告知方法调用者。
* throw ：抛出一个异常，至于该异常被捕获还是继续抛出都与它无关。

------------------------------------------------------------------------

异常的底层？？

提到JVM处理异常的机制，就需要提及Exception Table---称为异常表。

对处理异常的方法代码先用javac编译再用javap反编译就能看到该方法中显示异常表。



异常表中包含了一个或多个**异常处理者(Exception Handler)**的信息，这些信息包含如下

* from---可能发生异常的起始点
* to ---可能发生异常的结束点
* target ---上述from和to之前发生异常后的异常处理者的位置
* type ---异常处理者处理的异常的类信息

常见的运行时异常RuntimeException：

* java.lang.ArrayIndexOutOfBoundsException 数组索引越界异常。
* java.lang.NullPointerException 空指针异常。
* ClassCastException 类型转换异常类
* java.lang.IllegalArgumentException 非法参数异常
* java.lang.ArithmeticException 算术条件异常。譬如：整数除零等。

常见的IOException

* IOException：操作输入流和输出流时可能出现的异常。
* EOFException 文件已结束异常
* FileNotFoundException 文件未找到异常

异常实践：

1. **不要捕获 Throwable 类**。Throwable 是所有异常和错误的超类。你可以在 catch 子句中使用它，但是你永远不应该这样做！如果在 catch 子句中使用 Throwable ，它不仅会捕获所有异常，也将捕获所有的错误。
2. **不要忽略异常**。很多时候，开发者很有自信不会抛出异常，因此写了一个catch块，但是没有做任何处理或者记录日志。
3. **不要记录并抛出异常**。这个处理逻辑看着是合理的。但这经常会给同一个异常输出多条日志。因此，仅仅当想要处理异常时才去捕获，否则只需要在方法签名中声明让调用者去处理。
4. **包装异常时不要抛弃原始的异常**
5. **不要使用异常控制程序的流程**。不应该使用异常控制应用的执行流程，例如，本应该使用if语句进行条件判断的情况下，你却使用异常处理，这是非常不好的习惯，会严重影响应用的性能。
6. **不要在finally块中使用return**。try块中的return语句执行成功后，并不马上返回，而是继续执行finally块中的语句，如果此处存在return语句，则在此直接返回，无情丢弃掉try块中的返回点。

======================================================

反射

什么是反射？

🡪 JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java的反射机制。

反射的使用？

在Java中，Class类与java.lang.reflect类库一起支持反射技术。

反射包中，常用的类：

* Constructor类表示的是Class 对象所表示的类的构造方法，利用它可以在运行时动态创建对象
* Field表示Class对象所表示的类的成员变量，通过它可以在运行时动态修改成员变量的属性值(包含private)
* Method表示Class对象所表示的类的成员方法，通过它可以动态调用对象的方法(包含private)



getName、getCanonicalName与getSimpleName的区别?

* getSimpleName：只获取类名
* getName：类的全限定名，jvm中Class的表示，可以用于动态加载Class对象，例如Class.forName。
* getCanonicalName：返回更容易理解的表示，主要用于输出（toString）或log打印，大多数情况下和getName一样，但是在内部类、数组等类型的表示形式就不同了。