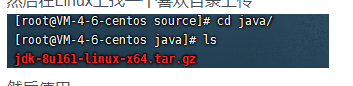
# Linx操作系统中安装java

## 1、下载jdk

## 2、在Linux上新建一个目录上传



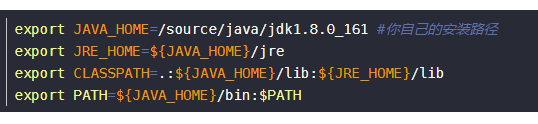
## 3、解压文件



## 4、配置环境变量

vim /etc/profile

在文件的最下面添加：



export JAVA\_HOME=/software/jdk1.8.0\_421

export JRE\_HOME=${JAVA\_HOME}/jre

export CLASSPATH=.:${JAVA\_HOME}/lib:${JRE\_HOME}/lib

export PATH=${JAVA\_HOME}/bin:$PATH

## 5、执行命令生效

. /etc/profile

## 6、查看版本

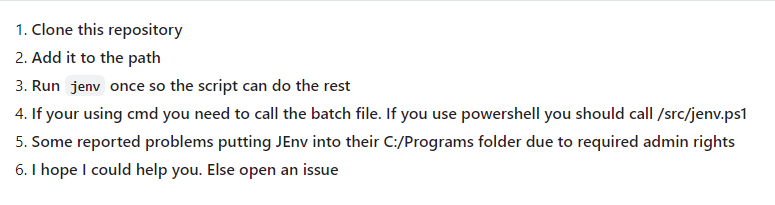
java -version

# JDK版本管理工具

## 网址

https://github.com/FelixSelter/JEnv-for-Windows

## 安装教程



## 使用方法

https://github.com/FelixSelter/JEnv-for-Windows

# 环境变量配置

## 变量名：JAVA\_HOME

变量值：C:\Program Files\Java\jdk1.7.0\_51

## 变量名：Path

变量值：;%JAVA\_HOME%\bin;

## 说明

jdk可以安装在任何位置，不一定安装在C盘。

# 版本查看方法

命令：java –version

# Java命令

## javac 类名.java

编译java文件

## java 类名

运行class文件

## Java -XX:+PrintCommandLineFlags -version

查看默认的垃圾收集器

# jar文件

1、JAR（Java Archive，Java 归档文件）是与平台无关的文件格式，它允许将许多文件组合成一个压缩文件。

2、JAR 文件格式以流行的 ZIP 文件格式为基础，与 ZIP 文件不同的是，JAR 文件不仅用于压缩和发布，而且还用于部署和封装库、组件和插件程序，并可被编译器和 JVM 这样的工具直接使用。在 JAR 中包含特殊的文件，如 manifests 和部署描述符，用来指示工具如何处理特定的 JAR文件。

3、使用范围：一个 JAR 文件可以用于

（1） 用于发布和使用类库。

（2）作为应用程序和扩展的构建单元。

（3）作为组件、applet 或者插件程序的部署单位。

（4）用于打包与组件相关联的辅助资源。

# jdk1.8采用的编码方式

## 基本说明

在 Java 1.8 及其早期版本中，JDK 使用了多种编码方式来处理字符数据。主要的编码方案包括：

UTF-16：Java 的内部字符串编码

系统默认编码：用于 I/O 操作的默认字符编码

## UTF-16 编码

### 内部表示

Java 中的 String 和 char 类型使用 UTF-16 编码。这意味着 Java 使用 16 位的 char 类型来表示字符：

**基本多语言平面（BMP）**：绝大多数常用字符（例如 ASCII 字符、汉字等）都在 BMP 中，这些字符用一个 16 位的 char 表示。

**辅助平面**：一些较少见的字符（例如表情符号和某些历史字符）位于 BMP 之外，这些字符由一对 16 位的 char（称为代理对）表示。

### 代理对

对于 UTF-16 编码的字符集，某些字符需要两个 16 位的 char（即两个 char 对象），这称为代理对（surrogate pairs）。代理对的范围是从 U+D800 到 U+DFFF。

例如，表情符号 "😊" 使用一个代理对来表示：

代理对的高位代理（High Surrogate）：U+D83D

代理对的低位代理（Low Surrogate）：U+DE0A

## 系统默认编码

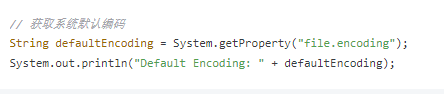
### 文件 I/O 和控制台 I/O

在文件读写和控制台输入输出等操作中，Java 还涉及到系统默认编码。这种编码在不同操作系统和 Java 环境中可能有所不同。

**Windows**：通常使用 Cp1252（也称为 Windows-1252）作为默认编码，但有时也可能使用其他编码，例如 Cp936（简体中文）。

**Linux 和 macOS**：通常使用 UTF-8 编码。

在 Java 1.8 中，你可以通过以下方法获取和设置默认字符编码：



### 设置编码

你可以在 Java 应用程序启动时，通过 -Dfile.encoding 选项来设置系统的默认编码，例如：



## 编码转换

在 Java 1.8 中，处理编码转换通常使用 java.nio.charset.Charset 类。你可以使用它来对不同的字符编码进行转换：



## 总结

1、内部编码：Java 使用 UTF-16 编码来表示 String 和 char。

2、默认编码：在 I/O 操作中，Java 使用系统默认编码，这可能依赖于操作系统和 JVM 设置。

3、编码转换：可以使用 Charset 类进行字符编码和解码的转换。

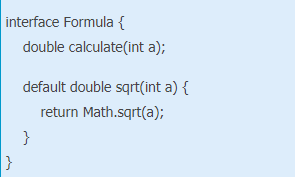
这些编码机制确保了 Java 程序能够处理各种语言和字符集的文本数据，同时也提供了与其他编码格式兼容的功能。

# jdk1.8特性

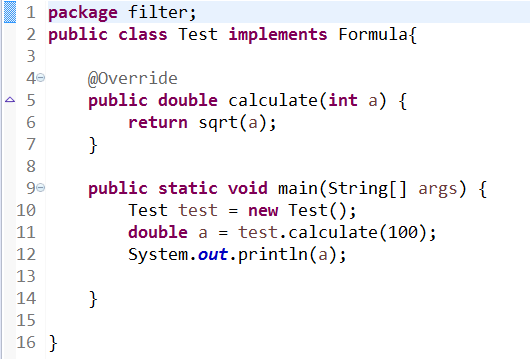
## 1、接口的默认方法

Java8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法。（在接口中可以添加扩展方法）

代码示例：



Formula接口在拥有calculate方法之外同时还定义了sqrt方法，实现了Formula接口的子类只需要实现一个calculate方法，默认方法sqrt将在子类上可以直接使用。



说明：在Java中只有单继承，如果要让一个类赋予新的特性，通常是使用接口来实现。

## 接口的静态方法

### 基本概念

接口中的静态方法和类中定义的静态方法一样，不属于特定对象，所以它们不是实现接口的api的一部分，必须使用InterfaceName.staticMethod来调用它们。

### 特性

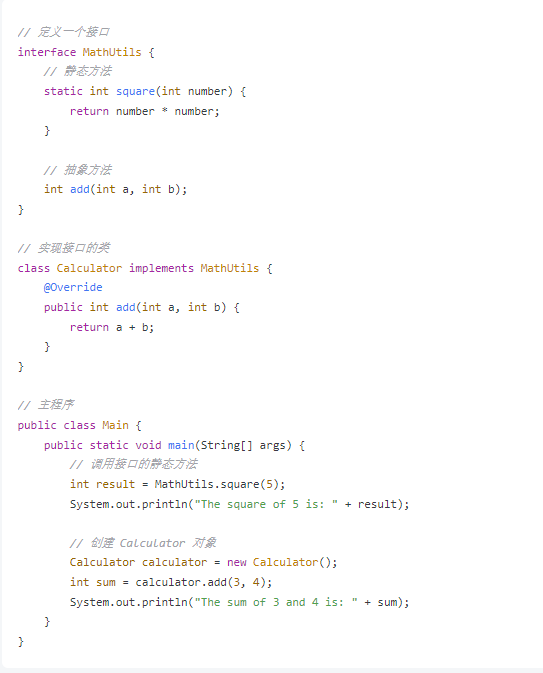
1、定义: 静态方法在接口中使用 static 关键字定义，可以在接口上直接调用。

2、访问: 通过接口名来调用静态方法，而不是通过实例或实现类。

3、不继承: 接口的静态方法不会被实现类继承，也不能被实现类重写。

4、用途: 常用于提供与接口相关的实用工具方法或功能。

### 示例代码



接口的静态方法提供了一种与接口相关的工具，而不是依赖于实现类的实例。

## 2、Lambda 表达式

### 基本概念

Lambda 是一个匿名函数,我们可以把 Lambda 表达式理解为是一段可以传递的代码(将代码像数据一样进行传递)。可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使 Java的语言表达能力得到了提升。

Lambda表达式的写法：

Comparator<Integer> comparator = (x, y) -> Integer.compare(x, y);

Runnable runnable = () -> System.out.println("Lambda表达式");

我们会发现Lambda表达式的写法更加的简洁、灵活。它只关心参数和执行的功能（具体需要干什么，比如->后的Integer.compare(x, y)）。

### 语法

Lambda 表达式在Java 语言中引入了一个新的语法元素和操作符。这个操作符为 “->” , 该操作符被称为 Lambda 操作符或剪头操作符。它将 Lambda 分为两个部分：

（1）左侧：指定了Lambda表达式需要的所有参数。

（2）右侧：指定了Lambda体,即Lambda表达式要执行的功能。

可以是如下的写法：

(params) -> expression

(params) -> statement

(params) -> { statements }

#### 语法格式一：无参，无返回值，Lambda 体只需一条语句

Runnable runnable = () -> System.out.println("lamda表达式");

#### 语法格式二：Lambda 只需要一个参数

Consumer<String> consumer=(x)->System.out.println(x);

#### 语法格式三：Lambda 只需要一个参数时，参数的小括号可以省略

Consumer<String> consumer=x->System.out.println(x);

#### 语法格式四：Lambda 需要两个参数

Comparator<Integer> comparator = (x, y) -> Integer.compare(x, y);

#### 语法格式五：当 Lambda 体只有一条语句时，return 与大括号可以省略

BinaryOperator<Integer> binaryOperator=(x,y)->(x+y);

#### 语法格式六：数据类型可以省略,因为可由编译器推断得出，称为“类型推断"

BinaryOperator<Integer> bo=(x,y)->{

System.out.println("Lambda");

return x+y;};

类型推断

上述 Lambda 表达式中的参数类型都是由编译器推断得出的，Lambda 表达式中无需指定类型，程序依然可以编译，这是因为 javac 根据程序的上下文，在后台推断出了参数的类型。Lambda 表达式的类型依赖于上下文环境，是由编译器推断出来的，这就是所谓的 “类型推断”。

### 示例

#### 1、用lambda表达式实现Runnable接口

new Thread( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();

#### 2、循环迭代List

在[使用](http://javarevisited.blogspot.sg/2012/03/how-to-loop-arraylist-in-java-code.html)lambda或不使用lambda表达式的情况下迭代列表（List）。你可以看到列表现在有了一个 forEach()方法，它可以迭代所有的对象，并将你的lambda代码应用在其中。

List features = Arrays.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");

features.forEach(n -> System.out.println(n));

使用Java 8的方法引用更方便，方法引用由::双冒号操作符标识。

features.forEach(System.out::println);

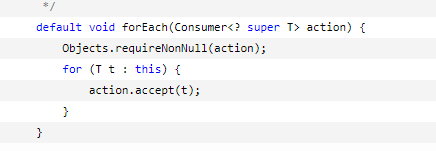
说明：列表循环的最后一个例子展示了如何在Java 8中使用方法引用（method reference）。

## 3、函数式接口

### 基本概念

函数式接口@FunctionalInterface，简称FI，简单的说FI就是指仅含有一个抽象方法的接口，以@Functionalnterface标注，注意这里的抽象方法指的是该接口自己特有的抽象方法，而不包含它从其上级继承过来的抽象方法。

在JDK8中，接口Iterable中默认实现了forEach方法，调用了 JDK8中增加的接口Consumer内的accept方法，执行传入的方法参数。JDK源码如下

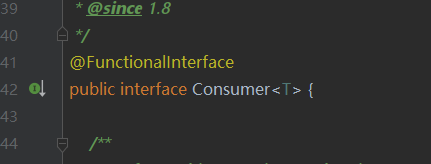


### Consumer<T>

#### 基本概念

代表了接受一个输入参数并且无返回的操作

#### 源码



#### 作用

处理数据。通过定义一个Consumer对象，我们可以将处理逻辑封装成一个函数，然后在需要使用该处理逻辑的地方直接调用该函数。

#### 方法

##### void accept(T t);

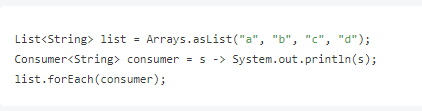
接受一个参数并执行操作

##### default Consumer<T> andThen(Consumer<? super T> after)

默认方法，用于将多个 Consumer 组合成一个链式操作

#### 代码示例

要使用Consumer接口，我们需要实现其accept()方法，该方法接受一个参数，表示需要进行处理的数据。例如：



在上述代码中，我们定义了一个名为list的字符串列表，并定义了一个Consumer对象consumer。在consumer.accept()方法中，我们定义了打印参数的逻辑。接着我们调用forEach()方法，并将consumer对象作为参数传入，实现了对列表中所有元素进行处理的目的。

#### 使用场景

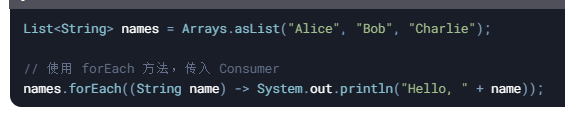
1、遍历集合并对每个元素执行操作。

2、日志记录、数据转换等无返回值的操作。

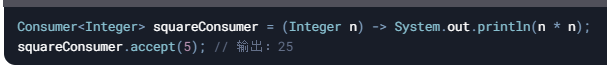
3、回调函数

4、组合多个操作。

遍历集合：

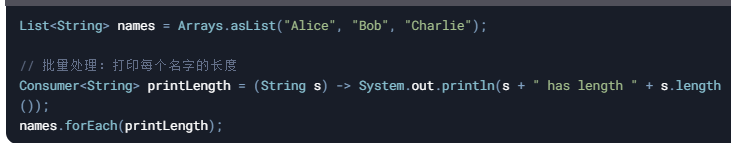


自定义消费逻辑：

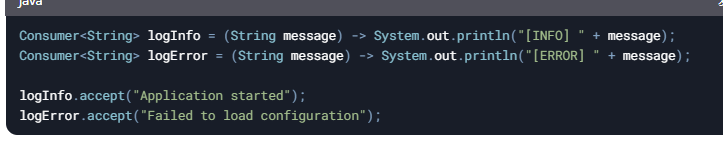


#### 实际应用

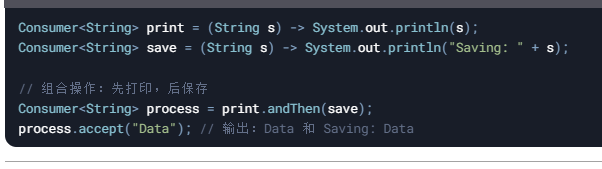
##### 1、批量处理数据



##### 2、日志记录



##### 3、组合操作



### BiConsumer<T, U>

#### 基本概念

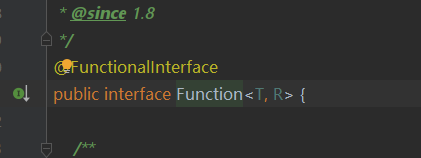
BiConsumer<T, U> 是 Consumer 的扩展版本，接受**两个**参数。

### Function<T,R>

#### 基本概念

它定义了一个具有一个输入参数和一个输出参数的函数，我们可以使用Function接口来定义一些简单的转换操作。

#### 源码



#### 方法

##### R apply(T t)

参数：

t：函数的参数

R：函数的结果

将函数应用于给定的参数

##### default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)

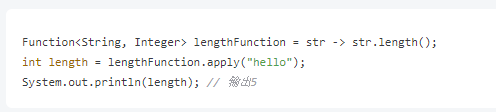
返回一个组合函数，首先应用当前函数，然后应用指定的后续函数。

##### compose(Function<? super V, ? extends T> before)

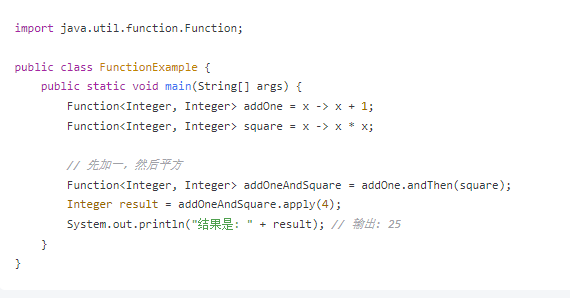
返回一个组合函数，首先应用指定的前置函数，然后应用当前函数。

#### 代码示例

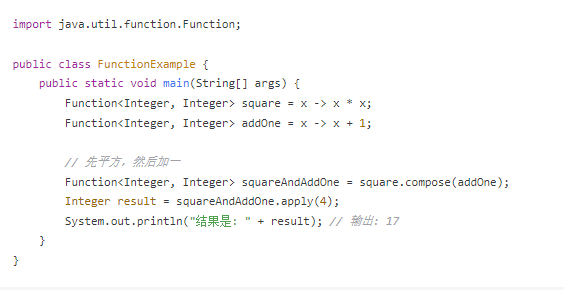
##### 1、将一个字符串转换为其长度



##### 2、组合函数



##### 3、使用 compose 方法



#### 适用场景

1、数据转换

2、流处理中的映射操作

3、作为方法参数传递的回调函数

#### 总结

Function<T, R> 接口提供了一种灵活的方法来处理输入和输出之间的转换，可以有效地与 Java 8 的流和其他函数式编程特性结合使用，提升代码的可读性和可维护性。

### BiFunction

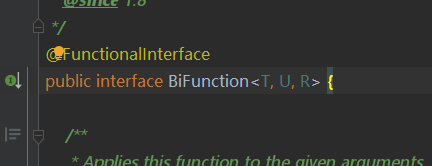
#### 基本概念

BiFunction 是 Java 中的一个函数式接口，它定义了一个接收两个参数并返回结果的函数。具体来说，BiFunction 接口有以下方法签名：

R apply(T t, U u)

其中，T 和 U 是输入参数的类型，R 是返回结果的类型。apply 方法接收一个类型为 T 的参数和一个类型为 U 的参数，并返回一个类型为 R 的结果。

#### 源码



#### 方法

##### R apply(T t, U u);

##### default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after)

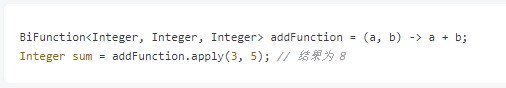
返回一个新的函数，该函数首先对输入应用 BiFunction，然后对结果应用指定的函数。

#### 作用

允许开发者自定义逻辑来处理两个输入参数，并产生一个结果。它可以用于各种场景，例如数据转换、聚合操作、复杂计算等。

##### 数据转换

BiFunction 可以将两个输入参数转换为一个输出结果。例如，将两个整数相加得到它们的和：



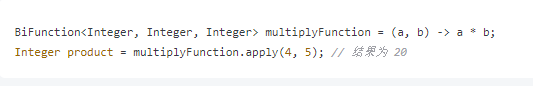
##### 聚合操作

BiFunction 可以用于聚合操作，将多个值合并为一个值。例如，将两个字符串拼接在一起：



##### 复杂计算

BiFunction 可用于处理复杂的计算逻辑，根据输入参数生成一个结果。例如，计算两个数字的乘积：

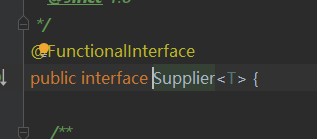


### Supplier<T>

#### 基本概念

无参数，返回一个结果。用于表示一个供应者。它没有任何参数，但可以提供一个类型为 T 的返回值。Supplier<T> 接口只定义了一个抽象方法 T get()，该方法用于获取一个值。

#### 源码



#### 作用

在于延迟生成或提供一个值，通过调用 get() 方法来获取这个值。

#### 使用场景

##### 惰性求值

在某些情况下，我们希望推迟计算或生成某个值，直到真正需要它的时候再进行计算。使用 Supplier<T> 可以将值的生成逻辑延迟到需要的时候才执行。

##### 依赖注入

在某些情况下，我们希望将值的获取逻辑从调用方解耦出来，使用 Supplier<T> 可以将值的提供者抽象化，由外部传入。

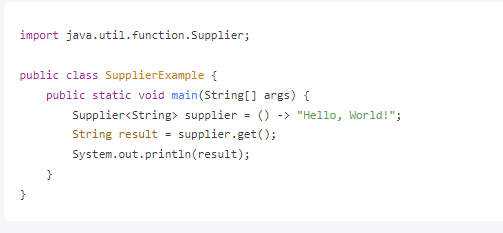
##### 延迟初始化

当我们需要在对象创建或初始化时，动态地提供某个值，而不是在一开始就立即确定该值。使用 Supplier<T> 可以在需要的时候获取该值。

#### 方法

##### T get();

#### 代码示例

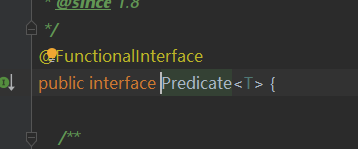


### Predicate

#### 基本概念

用于表示一个断言，即接受一个参数并返回一个布尔值。Predicate<T> 接口定义了一个抽象方法 boolean test(T t)，该方法用于对给定的输入值进行判断，并返回一个布尔结果。

#### 源码



#### 方法

##### boolean test(T t);

接受一个参数并返回一个布尔值。

##### default Predicate<T> and(Predicate<? super T> other)

返回一个新的 Predicate，表示当前 Predicate 和 other 的逻辑与（AND）

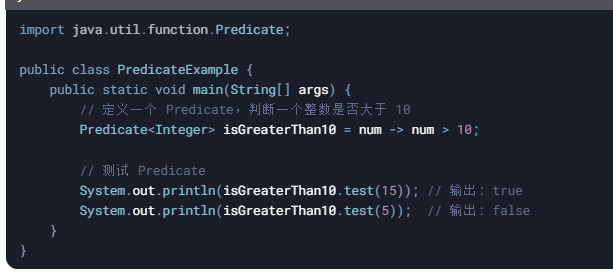
##### default Predicate<T> negate()

##### default Predicate<T> or(Predicate<? super T> other)

##### static <T> Predicate<T> isEqual(Object targetRef)

返回一个 Predicate，用于判断输入对象是否与 target 相等

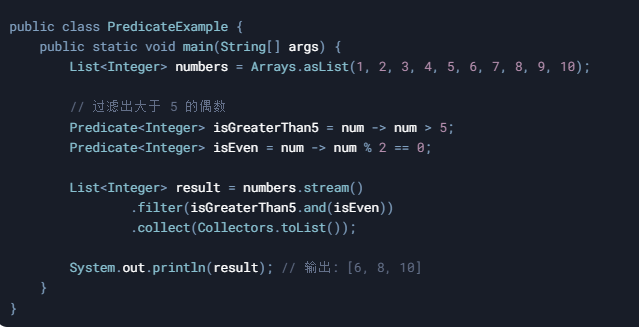
#### 代码示例



#### 使用场景

##### 1、过滤集合

Predicate 常用于结合 Stream API 对集合进行过滤。



##### 2、条件判断

Predicate 可以用于封装复杂的条件判断逻辑。



##### 3、动态条件组合

通过组合多个 Predicate，可以动态构建复杂的条件逻辑。



#### 注意事项

##### 1、空值处理

如果输入参数可能为 null，需要在 test 方法中显式处理，避免 NullPointerException。

##### 2、性能

对于复杂的条件判断，Predicate 的性能与传统的 if-else 语句相当，但代码更简洁。

##### 3、组合逻辑

使用 and、or 和 negate 方法时，注意逻辑运算的优先级和顺序。

### Comparator<T>

#### 基本概念

Comparator<T> 接口是用于定义对象的比较规则的接口。它允许我们根据自定义的排序逻辑对对象进行比较和排序。

#### 方法

##### int compare(T o1, T o2)

用于比较两个对象 o1 和 o2 的顺序。该方法返回一个负整数、零或正整数，具体的返回值意义如下：

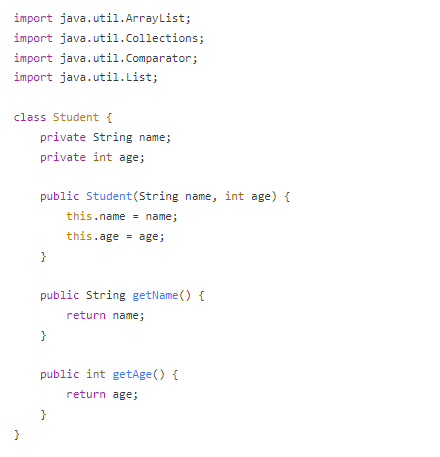
返回负整数：表示 o1 小于 o2，即 o1 应该排在 o2 之前。

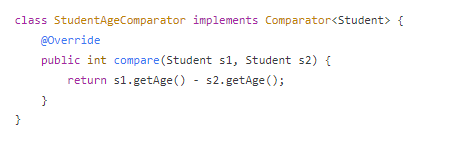
返回零：表示 o1 等于 o2，它们的顺序不变。

返回正整数：表示 o1 大于 o2，即 o1 应该排在 o2 之后。

通过实现 Comparator<T> 接口，我们可以根据自己的需求来定义对象的比较规则。这使得我们可以按照任意的方式对对象进行排序，而不仅限于对象自身的默认排序规则（比如实现 Comparable<T> 接口）。

#### 示例







上面的例子中，我们定义了一个 Student 类，包含姓名和年龄两个属性。然后，我们实现了一个 Comparator<Student> 接口的实现类 StudentAgeComparator，通过比较学生的年龄来定义排序规则。最后，在主函数中，我们创建了一个学生列表，使用 Collections.sort() 方法和自定义的比较器对学生列表按照年龄进行排序，然后打印排序结果。通过自定义比较器，我们可以根据不同的需求来对对象进行排序，而不需要修改对象本身的类或实现标准的自然排序接口。

除了在集合中排序，Comparator<T> 接口还可以在其他需要比较对象的地方使用，比如搜索算法、优先队列等。

## 4、方法与构造函数引用

### 基本概念

方法引用是一种简化Lambda表达式的方式，它可以直接引用已有的Java方法或构造函数，从而避免重复编写Lambda表达式。方法引用可以使代码更加简洁、易于理解。

### 分类

#### 1、静态方法引用

可以引用一个类中的静态方法，例如：ClassName::staticMethodName

#### 2、实例方法引用

可以引用一个特定对象中的实例方法，例如：object::instanceMethodName

#### 3、构造函数引用

可以引用一个类的构造函数，例如：ClassName::new

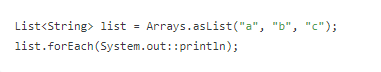
#### 4、超类方法引用

可以引用超类中定义的方法，例如：super::methodName

### 使用场景

方法引用可以用于任何函数式接口，只需要满足参数类型、返回值类型和函数式接口中抽象方法的参数类型、返回值类型相同即可。使用方法引用可以使代码更加简洁、易于理解，特别是对于重复的Lambda表达式而言，方法引用可以避免代码的重复。

### 代码示例



在上述代码中，我们使用了方法引用语法System.out::println，避免了编写Lambda表达式。该语法指定了调用System.out的println方法，并将其作为参数传递给forEach()方法，从而实现了对列表中所有元素进行输出的目的。

### 注意事项

1、方法引用只能用于函数式接口，接口中必须只有一个抽象方法。

2、使用方法引用可以使代码更加简洁、易于理解，特别是对于重复的Lambda表达式而言，方法引用可以避免代码的重复。

3、方法引用还可以和Lambda表达式一起使用，对于需要进行多次转换的场景更加方便。

## 5、Lambda作用域

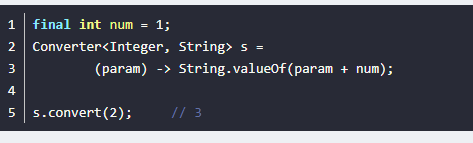
### 基本概念

你可以直接访问标记了final的外层局部变量，或者实例的字段以及静态变量。Lambda表达式不会从超类（supertype）中继承任何变量名，也不会引入一个新的作用域。Lambda表达式基于词法作用域，也就是说lambda表达式函数体里面的变量和它外部环境的变量具有相同的语义（也包括lambda表达式的形式参数）。此外this关键字及其引用，在Lambda表达式内部和外部也拥有相同的语义。

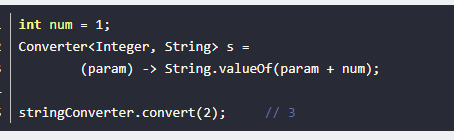
## 6、访问局部变量

### （1）可以直接在lambda表达式中访问外层的局部变量

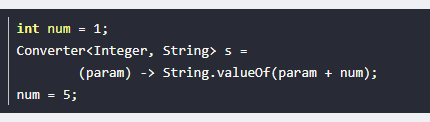
例如：



但是和匿名对象不同的是，lambda表达式**的局部变量**（eg:num）**可以不用声明为final。**

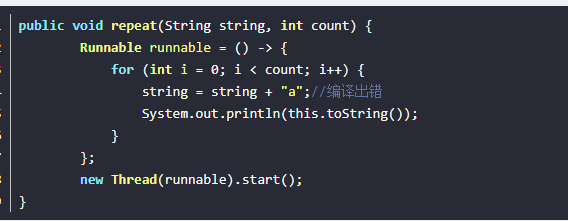


不过这里的局部变量(eg:num)必须不可被后面的代码修改（即隐性的具有final的语义），下面的代码无法编译：

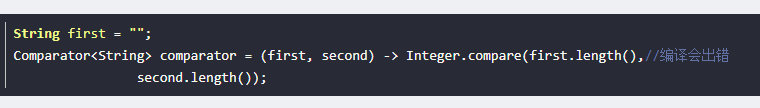


在Lambda表达式中试图修改局部变量是不允许的。

### **（2）在 Lambda 表达式当中被引用的变量的值不可以被更改**。****



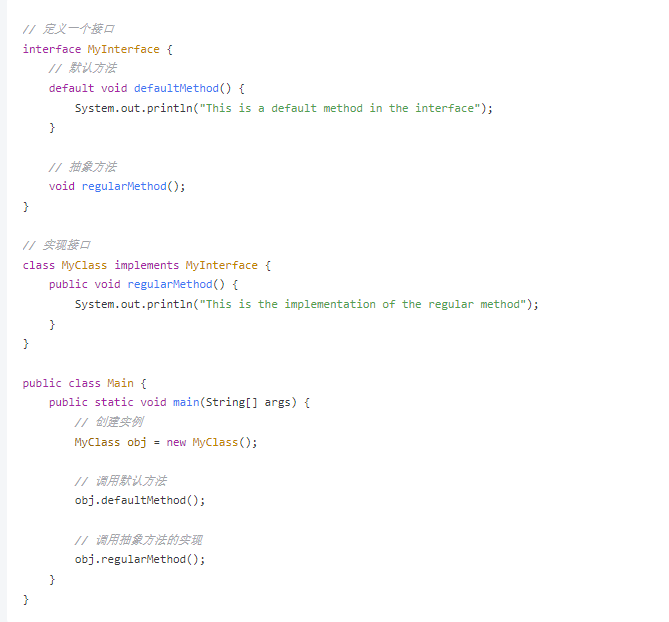
### （3）**在 Lambda 表达式当中不允许声明一个与局部变量同名的参数或者局部变量。**



## 7、访问对象字段与静态变量

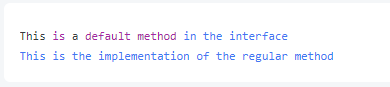
## 8、访问接口的默认方法

代码示例：



在上面的示例中，MyInterface接口包含一个默认方法defaultMethod和一个抽象方法regularMethod。MyClass类实现了MyInterface接口，并提供了对regularMethod方法的实现。在Main类的main方法中，我们创建了MyClass的实例，并通过该实例调用了默认方法和抽象方法的实现。

运行代码输出以下内容：



这样就成功地访问了接口的默认方法。

## 9、Date API

Java 8 在包java.time下包含了一组全新的时间日期API。新的日期API和开源的Joda-Time库差不多，但又不完全一样。

### （1）Clock时钟

Clock类提供了访问当前日期和时间的方法，Clock是时区敏感的，可以用来取代 System.currentTimeMillis() 来获取当前的微秒数。某一个特定的时间点也可以使用Instant类来表示，Instant类也可以用来创建老的java.util.Date对象。

代码示例：

Clock clock = Clock.systemDefaultZone();

long mill = clock.millis();

System.out.println(mill);

Instant instant = clock.instant();

Date date = Date.from(instant);

System.out.println(date);

### （2）Timezones 时区

在新API中时区使用ZoneId来表示。时区可以很方便的使用静态方法of来获取到。时区定义了到UTS时间的时间差，在Instant时间点对象到本地日期对象之间转换的时候是极其重要的。

代码示例：

System.out.println(ZoneId.getAvailableZoneIds());  
// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");  
ZoneId zone2 = ZoneId.of("Brazil/East");  
System.out.println(zone1.getRules());  
System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]  
// ZoneRules[currentStandardOffset=-03:00]

### （3）LocalTime本地时间

LocalTime 定义了一个没有时区信息的时间，例如晚上10点或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。

### （4）LocalDate 本地日期

## 10、Annotation注解

# jdk17特性

## 基本概念

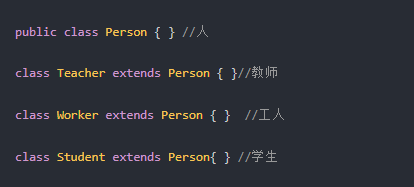
JDK 17 在 2021 年 9 月 14 号正式发布了！根据发布的规划，这次发布的 JDK 17 是一个长期维护的版本（LTS)。Java 17 提供了数千个性能、稳定性和安全性更新，以及 14 个 JEP（JDK 增强提案），进一步改进了 Java 语言和平台，以帮助开发人员提高工作效率。JDK 17 包括新的语言增强、库更新、对新 Apple (Mx CPU)计算机的支持、旧功能的删除和弃用，并努力确保今天编写的 Java 代码在未来的 JDK 版本中继续工作而不会发生变化。

## 密封的类和接口（正式版）

### 基本概念

封闭类可以是封闭类和或者封闭接口，用来增强 Java 编程语言，防止其他类或接口扩展或实现它们。这个特性由Java 15的预览版本晋升为正式版本。因为我们引入了sealed class或interfaces，这些class或者interfaces只允许被指定的类或者interface进行扩展和实现。使用修饰符sealed，您可以将一个类声明为密封类。密封的类使用reserved关键字permits列出可以直接扩展它的类。子类可以是最终的，非密封的或密封的。

之前我们的代码是这样的。



但是我们现在要限制 Person类 只能被这三个类继承，不能被其他类继承，需要这么做。



很强很实用的一个特性，可以限制类的层次结构。

## JEP 306：恢复始终严格的浮点语义

Java 编程语言和 Java 虚拟机最初只有严格的浮点语义。从 Java 1.2 开始，默认情况下允许在这些严格语义中进行微小的变化，以适应当时硬件架构的限制。这些差异不再有帮助或必要，因此已被 JEP 306 删除。

https://pdai.tech/md/java/java8up/java17.html

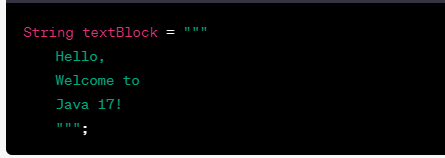
## 文本块

### 基本概念

Java 17 引入了一项新的语言特性，即文本块（Text Blocks），也称为多行字符串。文本块允许在代码中更方便地编写多行字符串，而无需使用传统的转义序列或字符串连接操作符。

文本块在编写包含大段文本或格式化字符串的代码时非常有用，例如 SQL 查询、HTML 片段、JSON 数据等。它简化了代码的编写，并提高了可读性和维护性。

文本块由三个双引号（"""）开始和结束，并且可以包含多行文本。下面是一个简单的示例：



在上面的示例中，textBlock 变量包含了一个文本块，其中包含了三行文本。文本块中的缩进和换行符都会被保留，因此文本块的输出结果将保持与代码中定义的格式一致。

### 特点

1、缩进控制：文本块中的每行文本会保留与代码中相同的缩进级别。这样可以使多行文本保持良好的可读性，并且不需要额外的空格或制表符。

2、转义字符：文本块中的转义字符（如 \n、\t 等）不会被解析，而会被直接当作普通字符处理。这样可以避免在字符串中使用大量的转义序列。

3、行尾空白：文本块中每行的行尾空白（包括空格和制表符）会被保留，除非在行的结尾显式使用了 \ 进行行续行。

4、引号处理：文本块中的双引号不需要进行转义，因为文本块是由三个双引号进行界定的。

### 怎么转义双引号

在 Java 17 的文本块中，如果你需要在文本块中包含双引号，你可以使用双引号进行转义。具体来说，你可以使用两个双引号（""）来表示一个双引号字符。

## instanceof操作符的模式匹配

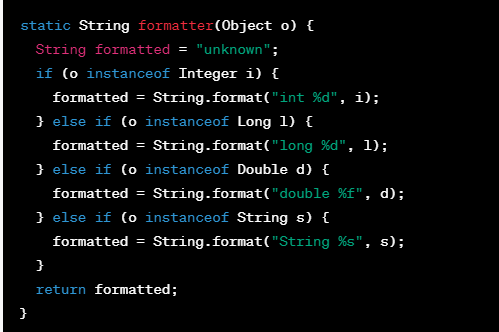
## switch表达式

### 基本概念

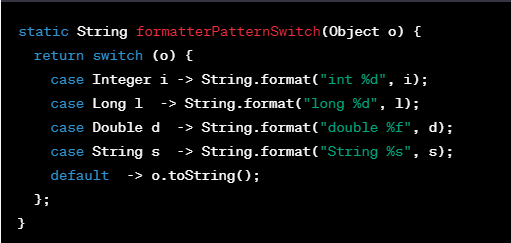
switch表达式与switch语句有所不同。switch语句只能作为语句使用，只能对数据进行求值，而不能返回值。而switch表达式作为表达式使用，可以返回值，并且可以在case中使用模式匹配。

### 示例代码

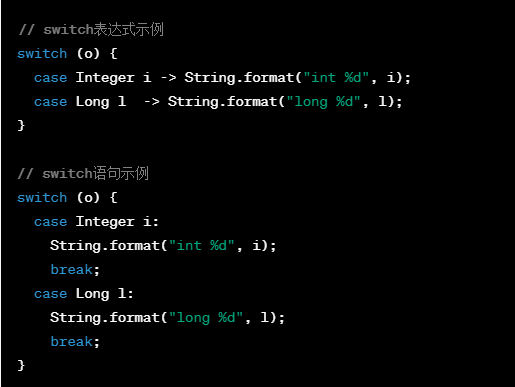
下面是一个使用if/else if和instanceof的示例代码



上述代码可以使用switch表达式进行改写



switch表达式与switch语句在使用上有一些区别。首先是在case后面使用的操作符，表达式中使用箭头（->），而语句中仍然使用冒号（:）。例如：



需要注意的是，使用switch语句时仍然需要使用break语句来终止switch流程，否则将继续执行下一个代码块。

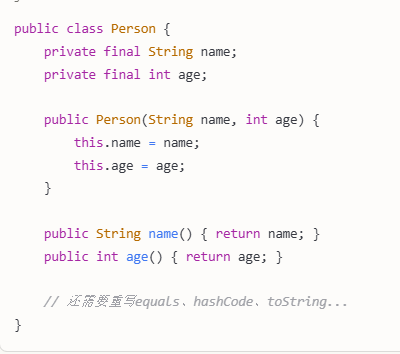
## Record

### 基本概念

Record是一种特殊的类，专门用来存储不可变数据，自动生成常用方法。

### 传统写法 vs Record

#### 传统写法



#### Record写法



### Record自动提供什么

1、构造方法

2、getter方法（方法名就是字段名）

3、equals()、hashCode()、toString()方法

4、所有字段都是final的

### 使用示例



### 限制

1、Record是final的，不能被继承

2、不能声明实例字段（除了Record组件）

3、所有字段都是不可变的

### 总结

Record就是专门用来装数据的"盒子"，Java帮你自动生成所有基础方法。

# 理解面向对象

1、Java语言是纯碎的面向对象的程序设计语言，这主要表现为Java完全支持面向对象的三种基本特征：封装、继承、多态。Java语言完全以对象为中心，Java语言的最小程序单位是类，整个Java程序由一个一个的类组成。

2、Java完全支持使用对象、类、继承、封装、消息等基本概念来进行程序设计，允许从现实世界中客观存在的事物（即对象）出发来构造。软件系统，在系统构造中尽可能运用人类的自然思维方式，实际上，这些优势是所有面向对象编程语言的共同特征，面向对象的方式实际上由OOA(面向对象分析)、OOD（面向对象设计）、OOP（面向对象编程）三个部分有机组成，其中OOA、OOD的结构需要使用一种方式来描述并记录，目前统一采用UML（统一建模语言）来描述并记录OOA和OOD的结果。

3、采用面向对象方式开发的软件系统，其最小的程序单元是类，这些类可以生成系统中的多个对象，而这些对象则直接映像成客观世界的各种事物。

4、面向对象的软件系统由多个类组成，类代表了客观世界中具有某种特征的一类事物，这类事物往往有一些内部的状态数据，当然程序没有必要记录该事物所有的状态数据，程序只要记录业务关心的状态数据即可。

5、面向对象的语言不仅使用类来封装一类事物的内部状态数据，而且类会提供操作这些状态数据的方法，还会为这类事物的行为特征提供相应的实现，这种实现也是方法，因此可以得到如下基本公式：

状态数据+方法=类定义，从这个等式来看，面向对象比面向过程的编程粒度要大，面向对象的程序单位是类，面向过程的程序单位是函数（相当于方法），因此面向对象比面向过程更简单、易用。

6、抽象也是面向对象的重要部分，抽象就是忽略一个主题中与当前目标无关的那些方面，以便更充分的注意与当前目标有关的方便，抽象并不打算了解全部问题，而只是考虑部分问题，例如，需要考察Person对象时，我们不可能在程序中把Person的所有细节都定义出来，通常只能定义Person的部分数据、部分行为特征，而这些数据、行为特征是软件系统所关心的部分。

7、面向对象支持如下几个功能：

（1）对象是面向对象方法中最基本的概念，它的基本特点有：标识唯一性、分类性、多态性、封装性、模块独立性。

（2）类是具有共同属性、共同方法的一类事物，类是对象的抽象，对象是类的实例，而类是整个软件系统最小的程序单元，类的封装性将各种信息细节隐藏起来，并通过公用方法来暴露该类对外提供的功能，从而提高了类的内聚性，降低了对象之间的耦合性。

（3）对象间的这种相互合作需要一个机制协助进行，这样的机制称为“消息”，消息是一个实例与另一个实例之间相互通信的机制。

（4）在面向对象的方法中，类之间共享属性和操作的机制称为继承，继承具有传递性。

8、通常而言，“基于对象”也使用了对象，但是无法利用现有的对象模板产生新的对象类型，继而产生新的对象，也就是说，“基于对象”没有继承的特点，而“多态”则需要继承，没有了继承的概念也就无从谈论“多态”，面向对象方法的三大基本特征（封装、继承、多态）缺一不可，例如：js语言就是基于对象的，它使用了一些封装好的对象，调用对象的方法，设置对象的属性，但是它们无法让开发者派生新的类，开发者只能使用现有对象的方法和属性。

9、判断一门语言是否是面向对象的，通常可以使用继承和多态加以判断，“面向对象”和“基于对象”都实现了“封装”的概念，但是面向对象实现了“继承和多态”，而“基于对象”没有实现这些。

10、对象就是面向对象程序设计的中心，对象是人们要进行研究的任何事物，从最简单的整数到复杂的飞机等均可看作对象，它不仅能表示具体的事物，还能表示抽象的规则、计划或事件。

11、对象具有状态，一个对象用数据值来描述它的状态，Java通过为对象定义属性来描述对象的状态，对象还有操作，这些操作可以改变对象的状态，对象的操作也被称为对象的行为，Java通过为对象定义方法来描述对象的行为。

12、对象实现了数据和操作的结合，对象把数据和对数据的操作封装成一个有机的整体，因此面向对象提供了更大的编程粒度，因此对程序员来说，更易于掌握和使用。

13、对象是Java程序的核心，所以Java里的对象具有唯一性，每个对象都有一个标识来引用它，如果某个对象失去了标识，这个对象将变成垃圾，只能等着系统垃圾回收机制来回收它。Java语言不允许直接访问对象，而是通过对对象的引用来操作对象。

14、开发者定义了Java类之后，就可以使用new关键字来创建指定类的对象了，每个类可以创建任意多个对象，多个对象的属性值可以不同--这表现为不同对象的数据存在差异。

# 类

一组有相同属性和方法的**对象**。类实际上是一种数据类型，是我们根据需要创建的复合数据类型。

# 封装

## 基本概念

指隐藏对象的属性和实现细节，仅对外提供公共访问方式，将代码及其处理的数据绑定在一起的一种编程机制，保证了程序和数据都不受外部干扰且不被误用。封装的基本单位是类。

## 好处

将变化隔离，便于使用，提高重用性，安全性。

## 原则

将不需要对外提供的内容都隐藏起来，把属性都隐藏，提供公共方法对其访问。

# 继承

一个对象获得另一个对象的属性和方法的过程。

## 好处

　　1：提高了代码的复用性。

　　2：让类与类之间产生了关系，提供了多态的前提。

# 多态

## 基本概念

允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）实现多态的技术称为动态绑定，是指在执行期间判断所引用对象的实际类型，根据其实际类型调用其相应的方法。

多态的举例：比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是AS3的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在Windows下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。

## 作用

消除类型之间的耦合关系。

## 多态存在的三个必要条件

（1）要有继承

（2）要有重写

（3）父类引用指向子类对象

## 多态的好处

### （1）可替换性

多态对已存在代码具有可替换性。

### （2）可扩充性

多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作，实际上新增加子类更容易获得多态功能。

### （3）接口性

多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。

### （4）灵活性

它在应用中体现了灵活多样的操作，提高了使用效率。

### （5）简化性

多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

## Java中多态的实现方式

1、接口实现

2、子类继承父类进行方法重写

3、同一个类中进行方法重载。

## 多态的形式

### 方法重载

在一个类中创建多个方法，它们具有相同的名字，但是具有不同的参数个数或类型，调用方法时通过传递给它们的不同参数个数和参数类型来决定具体使用哪个方法，这就是多态性。重载的时候，方法名要一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同，不能以返回类型作为重载函数的区分标准。方法重载是一个类中多态性的一种表现。

### 方法重写

父类与子类之间的多态性，对父类的函数进行重新定义，如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写。在Java中，子类可继承父类中的方法，而不需要重新编写相同的方法。但是有时子类并不想原封不动的继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数列表，则新方法将覆盖原有的方法，如果需要使用父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类。子类函数的访问修饰权限不能小于父类。

多态性是面向对象编程的一种特性，和方法无关，简单的说，就是同样的一个方法根据输入数据的不同，做出不同的处理，即方法的重载--有不同的参数列表（静态多态性），而当子类继承自父类的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类方法。即在子类中重写该方法-相同参数，不同实现（动态多态性）。重写的主要优点是能够定义某个子类特有的特征。

## 重写的规则

（1）方法名相同，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）返回类型必须与被重写的方法的返回类型相同，否则不能称为重写而是重载。

（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被重写方法申明更加宽泛的检查型异常。例如：

父类的一个方法申明了一个检查异常IOException，在重写这个方法就不能抛出Exception,只能抛出IOException的子类异常，可以抛出非检查异常。

## 重载的规则

（1）方法名相同，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了。

（3）可以有不同的访问修饰符。

（4）可以抛出不同的异常。

## 重写和重载的区别

调用被重载过的方法可以大大减少代码的输入量，同一个方法名只要往里面传递不同的参数就可以拥有不同的功能或返回值。用好重写和重载可以设计一个结构层次清晰而简洁的类，可以说重写和重载在编写代码过程中的作用非同一般。

# 驼峰命名法

## 基本概念

驼峰命名法是一种编程中常用的命名规则，它的特点是将多个单词连接在一起，并且除了第一个单词外，其他单词的首字母都大写。

## 形式

### 1、小驼峰命名法(camelCase)

第一个单词的首字母小写，之后每个单词的首字母大写。

例如：userName、firstName、mobilePhone、studentScore

### 2、大驼峰命名法(PascalCase)

每个单词的首字母都大写。

例如：UserName、FirstName、MobilePhone、StudentScore

## 使用建议

1、选择有意义的单词，避免使用缩写（除非是广为人知的缩写）

2、保持团队命名风格的一致性

3、遵循所使用语言或框架的命名约定

4、避免使用过长的名称，一般不超过3-4个单词组合

# Snake\_Case命名规范

## 基本概念

Snake\_Case(蛇形命名法)是一种编程和数据库命名约定，特点是使用下划线(\_)连接单词，所有字母通常小写。单词之间通过下划线分隔，形似蛇形蜿蜒，因此得名。

## 命名规则

所有字母小写

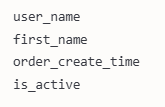
单词之间用下划线(\_)连接

不使用空格

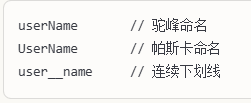
避免使用连续下划线

## 示例

### 正确示例



### 错误示例



## 应用场景

### 数据库命名

数据库中最常使用Snake\_Case，尤其是在：

表名

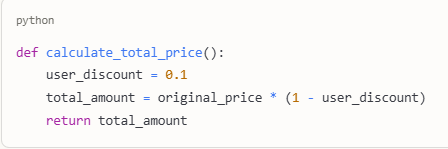
字段名

存储过程名

视图名

### 编程语言中的应用

Python推荐在变量、函数、方法名中使用Snake\_Case



# 变量

内存中的一个存储空间，用于存储常量数据。

作用：方便运算。因为有些数据不确定，所以确定该数据的名词和存储空间。

特点：变量空间可以重复使用。

什么时候定义变量？只要是数据不确定的时候，就定义变量。

# 数据类型

## 基本数据类型（8个）

byte、short、int、long、float、double、char、boolean

### byte

byte 是一种数据类型，它用于表示整数值。byte 数据类型占用 8 个位（即一个字节），可以存储的值范围是从 -128 到 127（包括 -128 和 127）需要注意的是，当对 byte 类型的变量进行运算时，会自动进行类型提升到 int 类型。如果需要将 int 类型的值赋给 byte 类型的变量，需要进行强制类型转换，确保赋值的值在 byte 的取值范围内。总结来说，Java 的 byte 数据类型可以存储范围为 -128 到 127 的整数值，占用 8 个位（1 字节）。

### boolean

基本类型 boolean 的默认值是 false，包装类型 Boolean 的默认值是 null。

## 引用数据类型

数组、类、接口、集合。

级别从低到高为：byte,char,short(这三个级)-->int-->float-->long-->double

自动类型转换：从低级别到高级别系统会自动转换。（低--》高）

强制类型转换：什么情况下使用?把一个高级别的变量赋给一个比该数级别低的变量。（高--》低）

public,protected,default,private这四种级别的修饰符都可以用来修饰类、方法和字段。

### Long

#### 基本概念

在 Java 中，long 是一种用于存储整数的基本数据类型，它可以表示大范围的整数值。

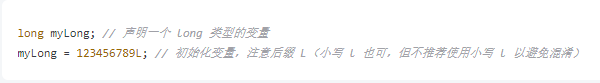
大小：64 位（8 字节）

范围：-2^63 到 2^63 - 1，即 -9,223,372,036,854,775,808 到 9,223,372,036,854,775,807

默认值：0L

表示：long 类型用于表示长整型整数，它比 int 类型能表示更大的数值范围。

#### 声明和初始化



#### 字面量

在 Java 中，long 字面量可以以十进制、十六进制、八进制和二进制表示，并且字面量必须以 L 或 l 结尾：



#### 适用场景

long类型通常用于存储较大的整数，例如时间戳、ID值。

#### 注意事项

1、内存占用：由于 long 占用 8 字节，比 int 占用的 4 字节多，因此在内存使用上可能会有影响。

2、运算精度：long 提供了足够的范围来处理大数，但在运算中仍需注意溢出的问题。

# 运算符

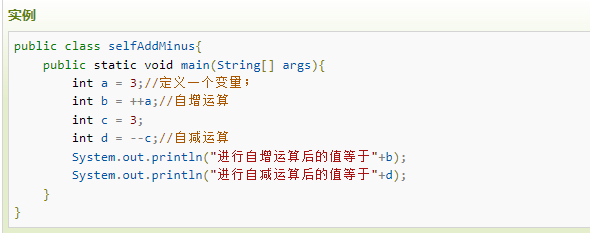
## 算术运算符

算术运算符用在数学表达式中，它们的作用和在数学中的作用一样，下表列出了所有的算术运算符。表格中的实例假设整数变量A的值为10，变量B的值为20：



## 自增自减运算符

**1、自增（++）自减（--）运算符**是一种特殊的算术运算符，在算术运算符中需要两个操作数来进行运算，而自增自减运算符是一个操作数。



运行结果：



解析：

int b = ++a; 拆分运算过程为: a=a+1=4; b=a=4, 最后结果为b=4,a=4

int d = --c; 拆分运算过程为: c=c-1=2; d=c=2, 最后结果为d=2,c=2

2、**前缀自增自减法(++a,--a):**先进行自增或者自减运算，再进行表达式运算。

3、**后缀自增自减法(a++,a--):**先进行表达式运算，再进行自增或者自减运算。

## 关系运算符

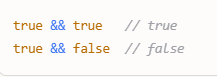
表格中的实例整数变量A的值为10，变量B的值为20：



## 逻辑运算符

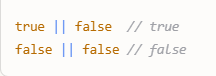
### && (逻辑与)

两个条件都为true时，结果才为true



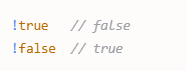
### || (逻辑或)

任意一个条件为true时，结果就为true



### ! (逻辑非)

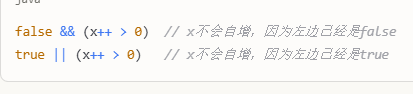
取反，true变false，false变true



### 短路特性

&&：左边为false时，不执行右边

||：左边为true时，不执行右边



### 非短路运算符

& 和 |

功能同&&和||，但不会短路，两边都会执行

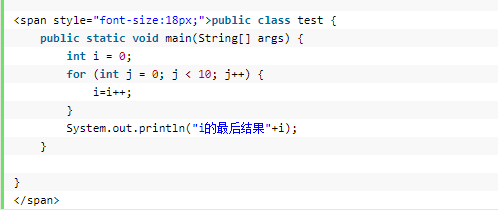


### 实际应用



# i++和++i的区别

先看一个代码：



正确答案是0

区别在于：i=i++是先赋值在自增,所以无论循环多少次，左边的i始终是0，最后结果就是0，改成i=++i,是可以达到效果的，++i是先自增在赋值。

所以要达到自增就可以用i=++i，但是一般是直接用i++，这样更好;这也算是JAVA的一个自增陷阱。

# &和&&区别

& ：无论左边结果是什么，右边都参与运算。

&&：短路与，如果左边为false，那么右边不参与运算。

# | 和|| 区别

|：两边都运算。

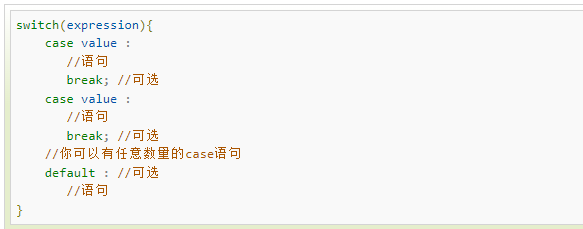
||：短路或，如果左边为true，那么右边不参与运算。

# switch语法

## 基本概念

switch case 语句判断一个变量与一系列值中某个值是否相等，每个值称为一个分支。

## 语法



## 规则

1、switch 语句中的变量类型可以是： byte、short、int 或者 char。从 Java SE 7 开始，switch 支持字符串 String 类型了，同时 case 标签必须为字符串常量或字面量。

2、switch 语句可以拥有多个 case 语句。每个 case 后面跟一个要比较的值和冒号。

3、case 语句中的值的数据类型必须与变量的数据类型相同，而且只能是常量或者字面常量。

4、当变量的值与 case 语句的值相等时，那么 case 语句之后的语句开始执行，直到 break 语句出现才会跳出 switch 语句。

5、当遇到 break 语句时，switch 语句终止。程序跳转到 switch 语句后面的语句执行。case 语句不必须要包含 break 语句。如果没有 break 语句出现，程序会继续执行下一条 case 语句，直到出现 break 语句。

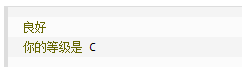
6、switch 语句可以包含一个 default 分支，该分支一般是 switch 语句的最后一个分支（可以在任何位置，但建议在最后一个）。default 在没有 case 语句的值和变量值相等的时候执行。default 分支不需要 break 语句。

switch case 执行时，一定会先进行匹配，匹配成功返回当前 case 的值，再根据是否有 break，判断是否继续输出，或是跳出判断。

## 代码示例



以上代码编译运行结果如下：



# 函数的作用

（1）用于定义功能。

（2）用于封装代码，提高代码的复用性。

注意：函数中只能调用函数，不能定义函数。

# 主函数main的作用

（1）保证该类的独立运行。

（2）它是程序的入口。

（3）它在被jvm调用。

# 面向对象的特点

（1）将复杂的事情简单化。

（2）面向对象将以前过程中的执行者变成了指挥者。

（3）面向对象这种思想符合现在人们思考问题的一种习惯。

# 成员变量和局部变量的区别

（1）成员变量定义在类中。局部变量定义在方法中、参数上、语句中。

（2）成员变量在这个类中有效。局部变量只在自己所属的大括号内有效，大括号结束，局部变量失去作用域。

（3）成员变量存在于堆内存中，随着对象的产生而存在，消失而消失。局部变量存在于栈内存中，随着所属区域的运行而存在，结束而释放。

# 成员变量和静态变量的区别

（1）成员变量所属于对象，所以也称为实例变量。静态变量所属于类，所以也称为类变量。

（2）成员变量存在于堆内存中，静态变量存在于方法区中。

（3）成员变量随着对象创建而存在，随着对象回收而消失。静态变量随着类的加载而存在，随着类的消失而消失。

（4）成员变量只能被对象所调用 ，静态变量可以被对象调用，也可以被类名调用，所以成员变量可以称为对象的特有数据，静态变量称为对象的共享数据。

# 堆内存和栈内存

Java把内存分成两种，一种叫做堆内存，一种叫做栈内存。

在函数中定义的**一些基本数据类型的变量和对象的引用变量都是在栈内存中分配**。当在一段代码块中定义一个变量时，java就在栈中为这个变量分配内存空间，当超过变量的作用域后，java会自动释放掉该变量分配的内存空间，该内存空间可以立刻被另作他用。**堆内存用于存放由new创建的对象和数组**。在堆中分配的内存，由**java虚拟机自动垃圾回收器来管理**。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名或者代号。

引用变量是普通变量，定义时在栈中分配内存，引用变量在程序运行到作用域外释放。而数组和对象在堆内存中分配。

# 构造函数

## 基本概念

用于给对象进行初始化，是给与之对应的对象进行初始化，它具有针对性，是函数中的一种。

## 特点

（1）该函数的名称和所在类的名称相同。

（2）不需要定义返回类型。

（3）函数没有具体的返回值。

记住：所有对象创建时都需要初始化才可以使用。

# 构造函数和一般函数的区别

（1）两个函数定义格式不同。

（2）构造函数是在对象创建时就被调用，用于初始化，而且初始化动作只执行一次。一般函数是对象创建后，需要调用才执行，可以被对象调用多次。

# 构造代码块和构造函数的区别

## 构造代码块

是给所有的对象进行初始化，也就是说所有的对象都会调用一个代码块，只要对象一建立就会调用这个构造代码块。

## 构造函数

是给与之对应的对象进行初始化，它具有针对性。

示例：

{

System.out.println("这是构造代码块！");

}

public Test10(){

System.out.println("这是构造函数！");

}

# static关键字

## 基本概念

static是一个修饰符，用于修饰成员(成员变量和成员函数)，不能修饰类。在java中，我们不能用static修饰顶级类（top level class），只有内部类可以用static修饰。

示例：

public class Child {

public static class Demo{

public static void main(String[] args) {

}

}

}

## static方法

就是没有this的方法，在static方法内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用static方法，这实际上正是static方法的主要用途。

static关键字的主要作用是方便在没有创建对象的情况下调用方法和变量。很显然，被static关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问。另外可以编写static代码块来优化程序的性能。static方法一般称作静态方法，并且由于这个特性，在静态方法中不能访问类的非静态成员变量和非静态成员方法，因为非静态成员方法/变量都是必须依赖具体的对象才能够被调用。但是要注意的是，虽然在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的。因此，如果说想在不创建对象的情况下调用某个方法，就可以将这个方法设置为static。我们最常见的static方法就是main方法，因为程序在执行main方法的时候没有创建任何对象，因此只有通过类名来访问。另外记住，即使没有显示地声明为static，类的构造器实际上也是静态方法。

## static变量

声明为static的变量实质上就是全局变量。当声明一个对象时，并不产生static变量的拷贝，而是该类所有的实例变量共用同一个static变量。静态变量与静态方法类似，所有此类实例共享此静态变量，也就是说在类装载时只分配一块存储空间，所有此类的对象都可以操控此块存储空间，当然对于final则另当别论了。

示例：

**class** Value {

**static** **int** *c* = 0;

**static** **void** inc() {

*c*++;

}

**public** **class** Count2 {

**public** **static** **void** prt(String s) {

       System.*out*.print(s);

    }

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

       Value v1, v2;

       v1 = **new** Value();

       v2 = **new** Value();

*prt*("v1.c=" + v1.*c* + "  v2.c=" + v2.*c*);

       v1.*inc*();

*prt*(" v1.c=" + v1.*c* + "  v2.c=" + v2.*c*);

    }

}

结果为：v1.c=0  v2.c=0 v1.c=1  v2.c=1

由此可以证明它们共享一块存储区，static变量有点类似于C中的全局变量的概念。

## 特点

（1）想要实现对象中共性数据的对象共享，可以将这个数据进行静态修饰。

（2）被静态修饰的成员，可以直接被类名所调用，也就是说静态的成员多了一种调用方式，类名.静态方式。

（3）静态随着类的加载而加载，而且优先于对象存在。静态方法中不能使用this，super关键字。

# 构造代码块

是给所有的对象进行初始化，也就是说所有的对象都会调用一个代码块，只要对象一建立就会调用这个构造代码块。

# 静态代码块

## 基本概念

就是一个有静态关键字static标识的一个代码块区域，定义在类中。

## 作用

可以完成类的初始化。静态代码块随着类的加载而执行，而且只执行一次（new 多个对象就只执行一次）。如果和主函数在同一类中，优先于主函数执行。

静态的注意：静态的生命周期很长。

静态代码块、构造代码块、构造函数同时存在时的执行顺序：静态代码块、构造代码块、构造函数。

# super关键字

## 基本概念

Java中的super关键字用于访问父类的成员

## 作用

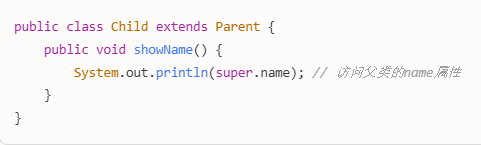
### 1、调用父类构造方法



### 2、访问父类方法



### 3、访问父类属性



## 关键点

1、super()必须是构造方法的第一条语句

2、用于区分父类和子类同名的方法或属性

3、不能在静态方法中使用super

4、super指向的是父类对象，而不是父类本身

## 总结

super就是"向上一级"的意思，让你能够明确调用父类的内容。

# super()和this()是否可以同时出现在构造函数中

　　两个语句只能有一个定义在第一行，所以只能出现其中一个。super()或者this():为什么一定要定义在第一行？因为super()或者this()都是调用构造函数，构造函数用于初始化，所以初始化的动作要先完成。

# final关键字

（1）这个关键字是一个修饰符，可以修饰类，方法，变量。

（2）被final修饰的类是一个最终类，不可以被继承。

（3）被final修饰的方法是一个最终方法，不可以被重写，但可以重载。

（4）被final修饰的变量是一个常量，只能赋值一次。

# transient关键字

transient是短暂的意思。对于transient 修饰的成员变量，在类的实例对象的序列化处理过程中会被忽略。 因此transient变量不会贯穿对象的序列化和反序列化，生命周期仅存于调用者的内存中而不会写到磁盘里进行持久化。

# 接口和抽象类的理解

在面向对象的概念中，所有的对象都是通过类来描绘的，但是并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。抽象类往往用来表征我们在对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象，我们不能把它们实例化，所以称之为抽象。比如：我们要描述“水果”，它就是一个抽象，它有质量、体积等一些共性（水果有质量），但又缺乏特性（苹果、橘子都是水果，它们有自己的特性），我们拿不出唯一一种能代表水果的东西（因为苹果、橘子都不能代表水果），可用抽象类来描述它，所以抽象类是不能够实例化的。当我们用某个类来具体描述“苹果”时，这个类就可以继承描述“水果”的抽象类，我们都知道“苹果”是一种“水果”。在面向对象领域中，抽象类主要用来进行类型隐藏，我们可以构造出一个固定的一组行为的抽象描述，但是这组行为却能够有任意个可能的具体实现方式，这个抽象描述就是抽象类，而这一组任意个可能的具体实现则表现为这个抽象类的所有派生类。接口和抽象类中的所有抽象方法不能有具体的实现，而应在它们的子类中实现所有的抽象方法，java的设计者可能为抽象方法的灵活性考虑，每个子类可根据自己的需要来实现抽象方法，抽象类是为了继承而存在的。

简言之抽象类是一种功能不全的类，接口只是一个抽象方法声明和静态不能被修改的数据的集合，两者都不能被实例化。从某种意义上说，接口是一种特殊形式的抽象类。在许多情况下，接口确实可以代替抽象类，如果你不需要刻意表达属性上的继承的话。进一步理解，关于java引入抽象类、接口的目的，1、从类的层次结构上看，抽象类是在层次的顶端，但在实际的设计当中，一般来说抽象类应当是后面才会出现。为什么？实际上抽象类的获取有点像数学中的提取公因式：ax+bx，x就是抽象类，如果你没有前面的式子，你怎么知道x是不是公因式呢？在这点上，也符合人们认识世界的过程，先具体后抽象。因此在设计过程中如果你得到大量的具体概念并从当中找到其共性时，这个共性的集合就是抽象类应当是没错的。2、interface从表面上看，和抽象类很相似，但用法完全不同。它的基本功能就是把一些毫不相关的类（概念）集合在一起形成一个新的、可集中操作的“新类”。我给学生的一个典型例子就是“司机”。谁可以当司机？谁都可以，只要领取了驾照。所以我不管你是学生，白领、蓝领还是老板，只要有驾照就是司机。

interface DriverLicence {

Licence getLicence(); //获取驾照

}

class StudentDriver extends Student implements DriverLicence {

}

class WhtieCollarEmployeeDriver extends WhtieCollarEmployee implements DriverLicence {

}

class BlueCollarEmployeeDriver extends BlueCollarEmployee implements DriverLicence {

}

class BossDriver extends Boss implements DriverLicence{

}

当我定义了“汽车”类后，我就可以指定“司机”了。

class Car {

setDriver(DriverLicence driver); //指定开车的司机

}

这时候，Car的对象并不关心这个司机到底是干什么的，他们的唯一共同点是领取了驾照（都实现了DriverLicence接口）。这个应当是接口最强大的地方也是抽象类无法比拟的。

总结：抽象类是提取具体类的公因式，而接口是为了将一些不相关的类“杂凑”成一个共同的群体。通常我们平时养成良好的习惯就是多用接口，毕竟java是单继承，不像C++，但是在需要使用抽象类的时候一定还是要用的。

# 抽象类

## 基本概念

抽象类（abstract class）的定义方式如下：

public abstract class AbstractClass //里面至少有一个抽象方法

{

　　public int t; //普通数据成员

　　 public abstract void method1(); //抽象方法，抽象类的子类在类中必须实现抽象类中的抽象方法

　　public abstract void method2(); //抽象方法

　　publi int method3 (){

　　…… //抽象类中可以赋予非抽象方法的默认行为，即方法的具体实现

　　}

}

## 特点

（1）抽象方法只能定义在抽象类中，抽象类和抽象方法必须由abstract关键字修饰（可以描述类和方法，不可以描述变量），抽象类中可以定义非抽象方法。

（2）抽象方法只定义方法声明，并不定义方法实现。

（3）抽象类不可以被实例化。

（4）只有通过子类继承抽象类并重写抽象类中的所有抽象方法后，该子类才可以实例化，否则该子类还是一个抽象类。

（5）抽象关键字abstract和哪些关键字不可以共存？final ,private , static。

# 接口

## 基本概念

接口中有抽象方法，说明接口不可以实例化。接口的子类必须实现接口中所有的抽象方法后，该子类才可以实例化，否则该子类还是一个抽象类。类与类之间存在着继承关系，类与接口之间存在的是实现关系。继承用extends，实现用implements，接口和类不一样的地方就是接口可以被多实现，这就是多继承改良后的结果，Java将多继承机制通过多实现来体现。 抽象类可以不用实现接口中的方法，抽象类不用全部实现接口中的所有方法，其余的方法实现可以交给该抽象类的子类去实现即可。

## 语法

接口（interface）的定义方式如下：

public interface Interface

{

　　static final int i; //接口中不能有普通数据成员，只能够有静态的不能被修改的数据成员,static表示全局，final表示不可修改，可以不用static final 修饰，会隐式的声明为static和final

　　 public void method1(); //接口中的方法一定是抽象方法，所以不用abstract修饰。

　　public void method2(); //接口中不能赋予方法的默认行为，即不能有方法的具体实现。

}

## 特点

接口都用于设计上，设计上的特点：（可以理解为主板上提供的接口）

（1）接口是对外提供的规则。

（2）接口是功能的扩展。

（3）接口的出现降低了耦合性。

耦合性也叫耦合度，是对模块间关联程度的度量。耦合的强弱取决于模块间接口的复杂性、调用模块的方式以及通过界面传送数据的多少。

模块间的耦合度是指模块之间的依赖关系，包括控制关系、调用关系、数据传递关系。模块间联系越多，其耦合性越强，同时表明其独立性越差，联系越少，耦合性越差，表明其独立性越强。软件设计中通常用耦合度和内聚度作为衡量模块独立程度的标准。划分模块的一个准则就是高内聚低耦合。

# 抽象类和接口的区别

（1）抽象类只能被继承，而且只能单继承。接口需要被实现，而且可以多实现。

（2）抽象类中可以定义非抽象方法，子类可以直接继承使用。接口中都是抽象方法，需要子类去实现。

（3）抽象类使用的是is a 关系，接口使用的是 like a 关系。 （is a 表示的是继承关系）

（4）抽象类的成员修饰符可以自定义。接口中的成员修饰符都是固定的，全都是public。

# 内部类

## 基本概念

内部类是指在一个外部类的内部再定义一个类，类名不需要和文件夹相同。内部类可以是静态static的，也可用public，default，protected和private修饰。（而外部顶级类即类名和文件名相同，只能使用public和default关键字修饰）。

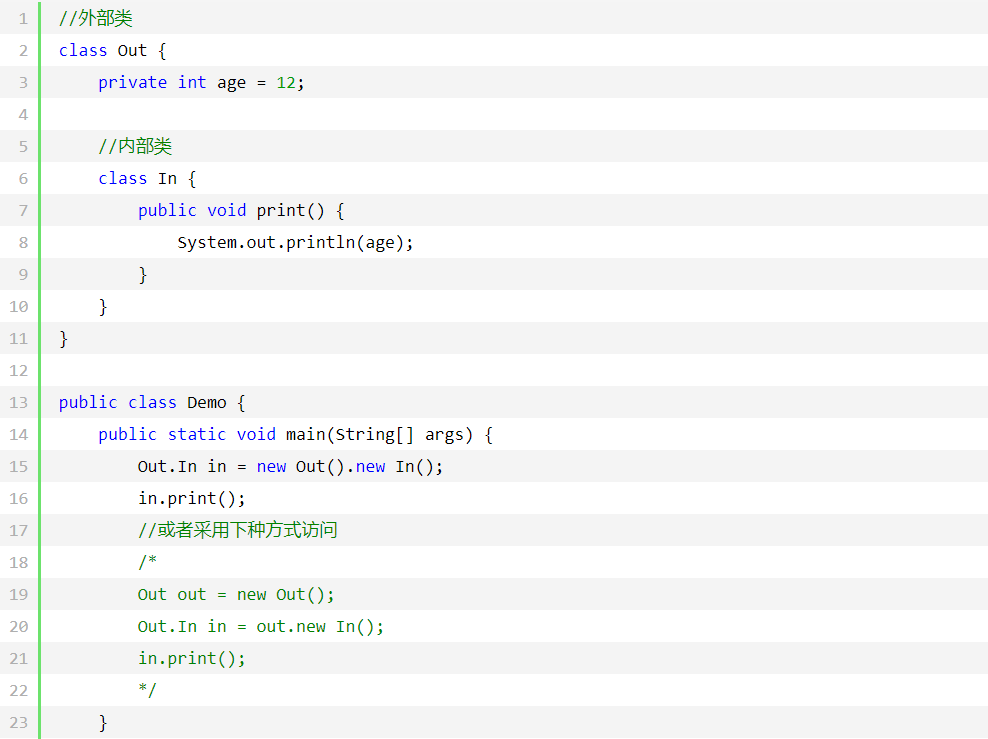
注意：内部类是一个编译时的概念，一旦编译成功，就会成为完全不同的两类，对于一个名为outer的外部类和其内部定义的名为inner的内部类。编译完成后出现outer.class和outer$inner.class两类，所以内部类的成员变量/方法名可以和外部类的相同。

成员内部类就是作为外部类的成员，可以直接使用外部类的所有成员和方法，即使是private的。同时外部类要访问内部类的所有成员变量/方法，则需要通过内部类的对象来获取。要注意的是，成员内部类不能含有static的变量和方法。因为成员内部类需要先创建了外部类，才能创建它自己的。在成员内部类要引用外部类对象时，使用outer.this来表示外部类对象。

示例：



## 内部类的基本结构



运行结果是12，从上面的例子中不难看出，内部类其实严重破坏了良好的代码结构，但为什么还要使用内部类呢？

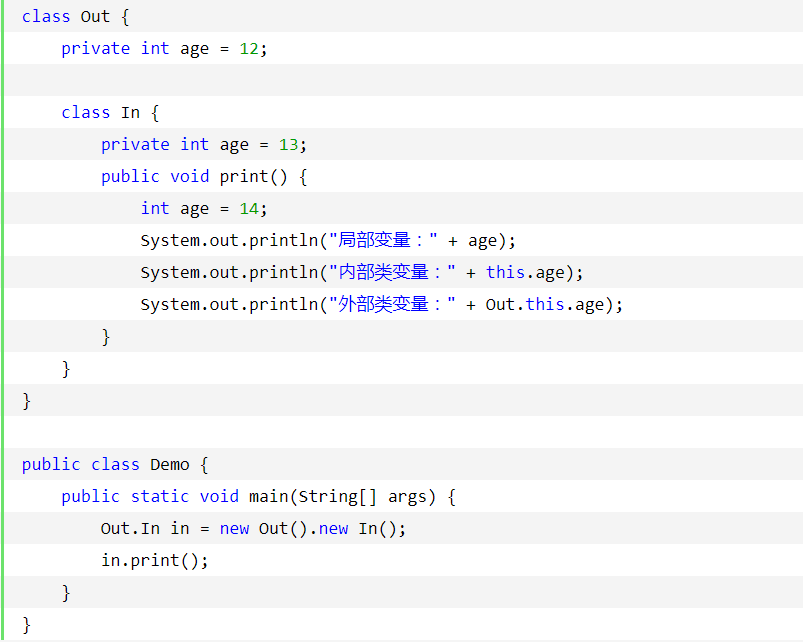
因为内部类可以随意使用外部类的成员变量（包括私有）而不用生成外部类的对象，这也是内部类的唯一优点。程序编译通过后会产生两个.class文件，分别是Out.class和Out$In.class，其中$代表了上面程序中Out.In中的那个 .

Out.In in = new Out().new In()可以用来生成内部类的对象，这种方法存在两个小知识点需要注意：

（1）开头的Out是为了标明需要生成的内部类对象在哪个外部类当中。

（2）必须先有外部类的对象才能生成内部类的对象，因为内部类的作用就是为了访问外部类中的成员变量。

## 内部类中的变量访问形式



运行结果：

局部变量：14

内部类变量：13

外部类变量：12

从实例1中可以发现，内部类在没有同名成员变量和局部变量的情况下，内部类会直接访问外部类的成员变量，而无需指定Out.this.属性名，否则，内部类中的局部变量会覆盖外部类的成员变量，而访问内部类本身的成员变量可用this.属性名，访问外部类的成员变量需要使用Out.this.属性名。

# 匿名内部类

## 基本概念

没有名字的内部类，就是内部类的简化形式，一般只用一次就可以用这种形式。匿名内部类其实就是一个匿名子类对象。想要定义匿名内部类需要前提，内部类必须继承一个类或者实现接口。

## 示例

匿名内部类的基本实现

abstract class Person {

public abstract void eat();

}

public class Demo {

public static void main(String[] args) {

Person p = new Person() {

public void eat() {

System.out.println("eat something");

}

};

p.eat();

}

}

在接口上使用匿名内部类

interface Person {

public void eat();

}

public class Demo {

public static void main(String[] args) {

Person p = new Person() {

public void eat() {

System.out.println("eat something");

}

};

p.eat();

}

}

由上面的例子可以看出，只要一个类是抽象的或是一个接口，那么其子类中的方法都可以使用匿名内部类来实现，最常用的情况就是在多线程的实现上，因为要实现多线程必须继承Thread类或是实现Runnable接口。

## Thread类的匿名内部类实现

public class Demo {

public static void main(String[] args) {

Thread t = new Thread() {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.print(i + " ");

}

}

};

t.start();

}

}

## Runnable接口的匿名内部类实现

public class Demo {

public static void main(String[] args) {

Runnable r = new Runnable() {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.print(i + " ");

}

}

};

Thread t = new Thread(r);

t.start();

}

}

# 枚举

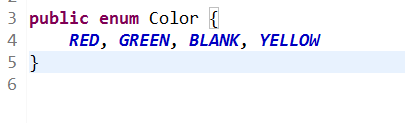
## 基本说明

枚举不能实例化。

## 用法一：常量

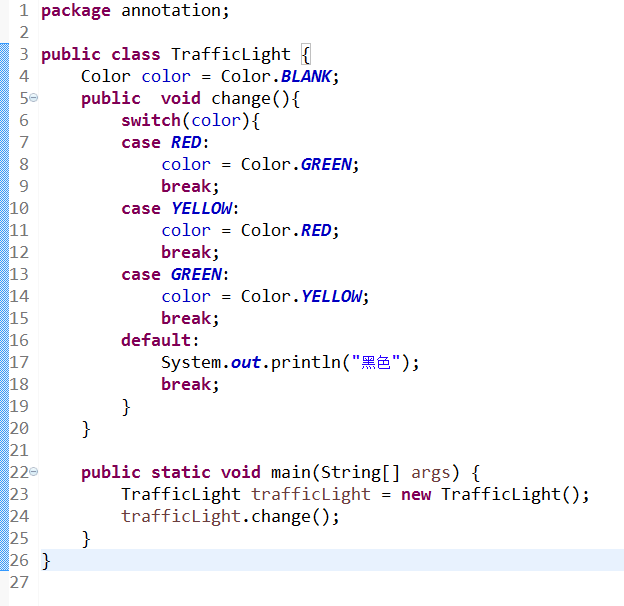
在JDK1.5 之前，我们定义常量都是： public static fianl.... 。现在好了，有了枚举，可以把相关的常量分组到一个枚举类型里，而且枚举提供了比常量更多的方法。

代码示例：



## 用法二：switch

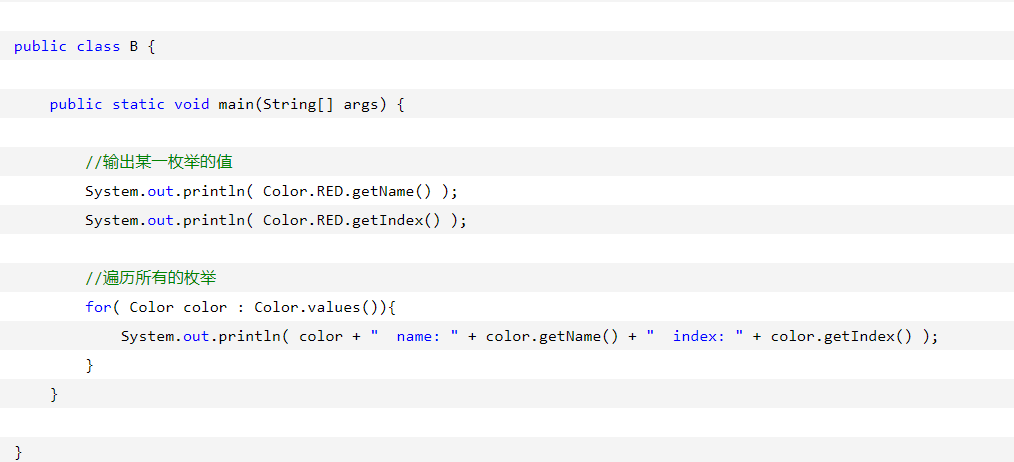
JDK1.6之前的switch语句只支持int,char,enum类型，使用枚举，能让我们的代码可读性更强。



## 用法三：向枚举中添加新方法



使用：



## values()方法

返回一个包含枚举类型所有常量的数组，这个方法在枚举类型中非常有用，可以方便地遍历和访问枚举常量。

1、values() 方法的返回类型是一个数组，该数组包含了枚举类型的所有常量。数组的元素顺序与枚举常量的声明顺序一致。

2、values() 方法是由编译器自动生成的，不需要显式地在枚举类型中定义。

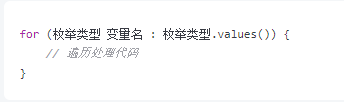
3、values() 方法是静态方法，可以直接通过枚举类型调用，而不需要创建枚举对象。

4、values() 方法返回的数组是一个新数组，对数组的修改不会影响到原来的枚举类型。

## 循环遍历枚举

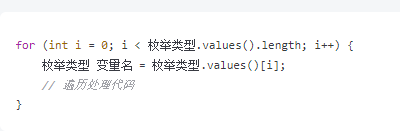
可以使用 Java 中提供的 for-each 循环或者普通的 for 循环。

### 1、使用 for-each 循环进行枚举遍历



在上述代码中，values() 方法是枚举类型自动生成的静态方法，可以返回这个枚举类型的所有常量值。遍历时，使用 for-each 循环，在每次迭代中将当前的常量值保存到变量中，并执行对应的处理代码即可。

### 2、使用普通的 for 循环进行枚举遍历



在上述代码中，使用 values() 方法获取枚举类型的所有常量值，并通过 length 属性获取这些常量值的数量。然后，使用普通的 for 循环对这些常量值进行遍历，在每次迭代中将当前的常量值保存到变量中，并执行对应的处理代码即可。

## 总结

1、枚举的本质是类，在没有枚举之前，仍然可以按照java最基本的编程手段来解决需要用到枚举的地方。枚举屏蔽了枚举值的类型信息，不像在用public static final定义变量必须指定类型。枚举是用来构建常量数据结构的模板，这个模板可扩展。枚举的使用增强了程序的健壮性，比如在引用一个不存在的枚举值的时候，编译器会报错。

2、枚举在Java家族中只占了很小的一块比重，所以我在项目中用枚举的地方不是很多，所以常量大都是用public static final来定义的。

# Error

错误，一般情况下不编写针对性的代码进行处理，通常是jvm发生的，需要对程序进行修正。Exception：异常，可以有针对性的处理方式。无论是错误还是异常，它们都有具体的子类体现每一个问题，它们的子类都有一个共性，就是都以父类名作为子类的后缀名。

处理异常的方式：（1）抛出（2）捕获。

# throw和throws关键字的区别

throw用于抛出异常对象，后面跟的是异常对象，throw用在函数内。throws用于抛出异常类，后面跟的是异常类名，可以跟多个，用逗号隔开，throws用在函数上。通常情况：函数内容如果有throw抛出异常对象，并没有进行处理，那么函数上一定要声明，否则编译失败，但是也有特殊情况。

# 异常

## 基本概念

在Java中，异常（Exception）是指程序在运行过程中发生的非正常情况，它们干扰了正常的执行流程。Java提供了一套强大的异常处理机制，用于捕获和处理这些异常，从而提高程序的健壮性和可维护性。

## 分类

Java中的异常分为两大类：**已检查异常（Checked Exception）** 和 **未检查异常（Unchecked Exception）。**

### 已检查异常

已检查异常是编译时异常，继承自 java.lang.Exception 类，但不包括 RuntimeException 子类。它们必须在代码中显式地进行处理，否则编译器会报错。这类异常通常代表可以恢复的条件或预计的异常情况。

常见的已检查异常包括：

IOException

FileNotFoundException

SQLException

### 未检查异常

未检查异常是程序运行时异常，继承自 java.lang.RuntimeException 类。对于这类异常，编译器不会强制要求进行处理，可以选择捕获处理，但不强制要求使用 try-catch 块来捕获，也不要求在方法签名中声明 throws 关键字来声明抛出。这类异常通常代表程序逻辑错误或不合理的操作。

常见的未检查异常包括：

NullPointerException

ArrayIndexOutOfBoundsException

ClassCastException

ArithmeticException

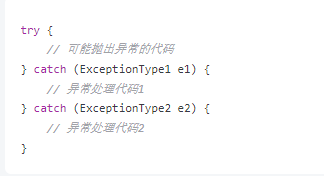
此外还有 Error 类及其子类，如 OutOfMemoryError 和 StackOverflowError，用于表示更严重的问题，这些问题一般不由应用程序捕获和处理。

## 异常的处理

Java 提供了基于 try-catch 块的异常处理机制，以及 finally 块用于在异常处理结束后执行清理工作。

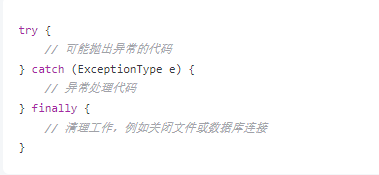
### try-catch 块

try 块包含可能抛出异常的代码，catch 块用于捕获和处理异常。



finally 块

finally 块中的代码无论是否发生异常都会执行，通常用于释放资源。



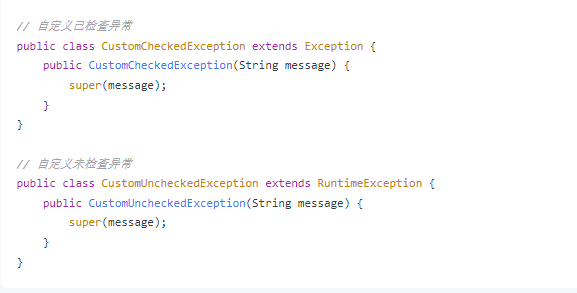
### throws关键字

如果一个方法可能抛出已检查异常，但并不在方法内部捕获处理，则必须在方法签名中使用 throws 关键字声明该异常。



## 自定义异常

在 Java 中可以自定义异常，通过继承 Exception 或 RuntimeException 类来实现。



## 示例代码

以下是一个综合示例，演示如何使用异常处理机制：



在这个示例中，readFile 方法尝试读取一个文件，并可能抛出 IOException 或自定义的 CustomCheckedException。在 main 方法中，我们使用 try-catch-finally 块来捕获和处理这些异常，并在 finally 块中执行清理工作。

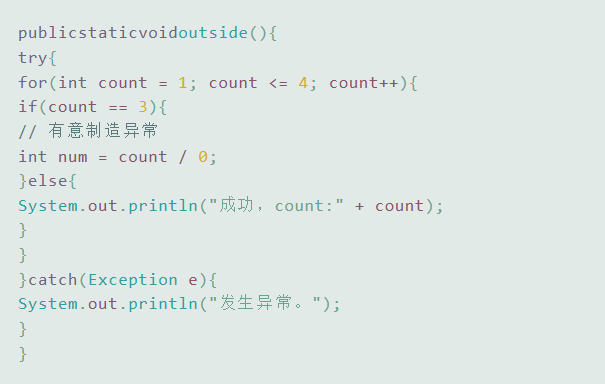
## 总结

Java 的异常处理机制通过 try-catch-finally 块和 throws 关键字提供了强大的工具，帮助开发者捕获和处理异常情况，提高程序的健壮性和可维护性。同时，通过自定义异常类型，开发者可以创建更加适合具体应用需求的异常处理逻辑。

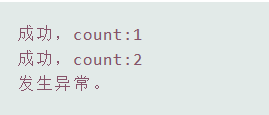
## try-catch 应该写在 for 循环的内部还是外部

这个问题的答案并不是简单地选择总是放在里面或外面就好，而是要结合具体使用场景来回答。因为当发生异常时，将 try-catch 放在循环外部或内部会产生不同的效果。所以要结合具体的业务场景来考虑。

### try-catch 放在for循环外部

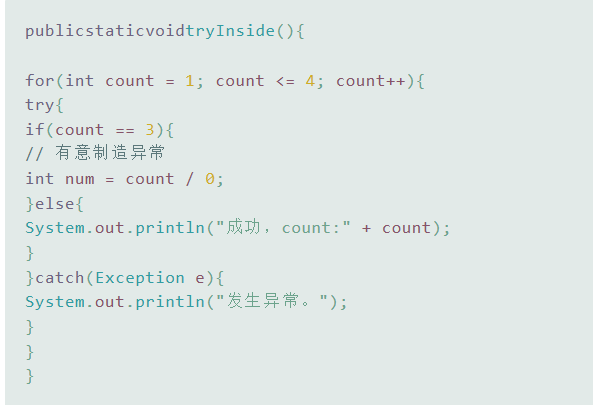


输出结果

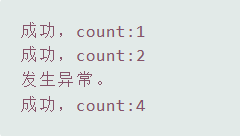


显然，当 try-catch 放在 for 循环外部时，如果循环中发生异常，for 循环将被终止。

### try-catch 放在 for 循环内部



输出结果：



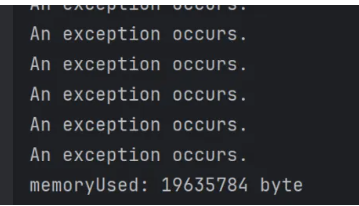
当 try-catch 在 for 循环内部时，循环中发生异常后会被 catch 捕获，不影响 for 循环的继续。

### 性能

在时间上，其实并没有差异。在内存使用方面，如果没有发生异常，同样也没有差异。但如果发生异常，那就需要注意了。



输出结果：



可以看到，当 try-catch 放在 for 循环内部时，由于异常不会终止 for 循环，如果确实存在大量业务处理过程中都会出现异常的场景，那么短时间内会消耗大量内存。如果代码错误情况不多，则 try-catch 放在循环内部和外部的差异几乎不大。

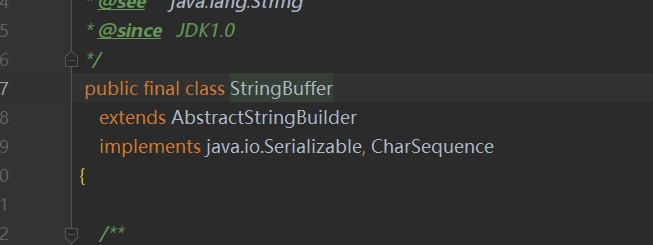
# String

Java中用String类进行描述字符串，对字符串进行了对象的封装。这样的好处是可以对字符串这种常见数据进行方便的操作。对象封装后，可以定义N多属性和行为。

特点：字符串一旦被初始化，就不可以被改变，存放在方法区中的常量池中。

# StringBuffer

## 源码



## 特点

（1）可以对字符串内容进行修改。

（2）是一个容器。

（3）是可变长度的。

（4）缓冲区中可以存储任意类型的数据。

（5）最终需要变成字符串。

# StringBuilder

## 基本概念

如果要修改字符串而不是创建新的对象，可以使用StringBuilder对象。当在一个循环中将许多字符串连接在一起时，使用 StringBuilder类可以提升性能。

## 源码

图形用户界面, 文本, 应用程序

描述已自动生成

# StringBuffer和StringBuilder的区别

（1）StringBuffer是线程安全，StringBuilder是线程不安全。

（2）单线程操作，使用StringBuilder 效率高，多线程操作，使用StringBuffer安全。

# 泛型

## 基本概念

（1）jdk1.5版本以后出现的一个安全机制，表现格式：< >。

（2）所谓泛型就是允许在定义类、接口、方法时使用类型形参，这个类型形参将在声明变量、创建对象、调用方法时动态的指定（即传入实际的类型参数，也可称为类型实参）。Java5改写了集合框架中的全部接口和类，为这些接口、类增加了泛型支持，从而可以在声明集合变量、创建集合对象时传入类型实参，比如看到的List<String>和ArrayList<String>两种类型。

（3）包含泛型声明的类型可以在定义变量、创建对象时传入一个类型实参，从而可以动态的生成无数多个逻辑上的子类，但这种子类在物理上并不存在。

（4）我们可以为任何类、接口增加泛型声明（并不是只有集合类才可以使用泛型声明，虽然集合类是泛型的重要使用场所）。

## 定义泛型类

//定义泛型类

public class Apple<T> {

private T info;

public Apple(){}//构造函数

public Apple(T info){

this.info = info;

}

public T getInfo() {

return info;

}

public void setInfo(T info) {

this.info = info;

}

public static void main(String[] args) {

Apple<String> a = new Apple<String>("苹果");

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

Apple<Double> a1 = new Apple<Double>(3.14);

System.out.println(a1.getInfo());

}

}

当创建带泛型声明的自定义类，为该类定义构造器，构造器名还是原来的类名，不要增加泛型声明。例如：为Apple<T>类定义构造器，其构造器名依然是Apple,而不是Apple<T>，调用该构造器时却可以使用Apple<T>的形式，当然应该为T形参传入实际的类型参数，Java7提供了菱形语法，允许省略<>中的类型实参。Apple<T>意思是Apple这个类里面需要用到另外一个类，但是又不能确定要用到的那个类的具体类型，所以暂时用T来代替，当具体的程序知道要用到的那个类型是什么的时候就用那个类型来代替T就ok了。例如我需要实例化一个Apple类，需要用到的那个类是String，那么我就可以这么来实例化Apple<String>();

## 好处

（1）将运行时期的ClassCastException问题转换成了编译失败，体现在编译时期，程序员就可以解决问题。

（2）避免了强制转换的麻烦。

只要带有<>的类或者接口，都属于带有类型参数的类或者接口，在使用这些类或者接口时，必须给<>中传递一个具体的引用数据类型。

（3）泛型技术：其实应用在编译时期，是给编译器使用的技术，到了运行时期，泛型就不存在了。因为泛型的擦除：也就是说编辑器检查了泛型的类型正确后，在生成的类文件中是没有泛型的。

（4）在运行时如何知道获取的元素类型而不用强转呢？

泛型的补偿：因为存储的时候，类型已经确定了是同一个类型的元素，所以在运行时，只要获取到该元素的类型，在内部进行一次转换即可，所以使用者不用再做转换动作了。

## 什么时候使用泛型类

　　当类中操作的引用数据类型不确定的时候，以前用的Object来进行扩展的，现在可以用泛型来表示。这样可以避免强转的麻烦，而且将运行问题转移到编译时期的问题。

## 底层实现原理

使用的是”类型擦除“，也就是说java里面的泛型只存在于源代码里面，一旦经过编译之后，所有的泛型都会被擦除掉，全部被替换为原来的裸类型，并在对元素进行访问和修改的时候，才会加上强制类型转换。

# Java中两层for循环问题

Java中如果有两个for循环，当执行里层的循环结束时，如果想要继续执行外层的循环，那么就需要在里层循环代码的下面添加一个标识，判断里层循环的结果，然后继续执行外层循环，使用break关键字跳出当前循环，执行外层循环。

# Instanceof

## 基本概念

instanceof是一个运算符，运算符是双目运算符，左面的操作元是一个对象，右面是一个类或接口，当左面的对象是右面类创建的对象时，该运算符运算的结果是true，否则是false。

说明：一个类的实例包括本身的实例以及所有直接或间接子类的实例，instanceof左边操作元显示声明的类型与右边操作元必须是同种类或右边是左边父类的继承关系，不同的继承关系下，编译会出错。

## 示例

**package** mavenproject;

**class** Person{

}

**class** Student **extends** Person{

}

**class** Postgraduate **extends** Student{

}

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

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

*instanceofTest*(**new** Student());

}

**public** **static** **void** instanceofTest(Person p){

**if**(p **instanceof** Person){

System.***out***.println("p是Person类的实例");

}

**if**(p **instanceof** Student){

System.***out***.println("p是Student类的实例");

}

**if**(p **instanceof** Postgraduate){

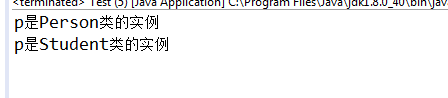
System.***out***.println("p是Postgraduate类的实例");

}

}

}

输出的结果：



# 注解

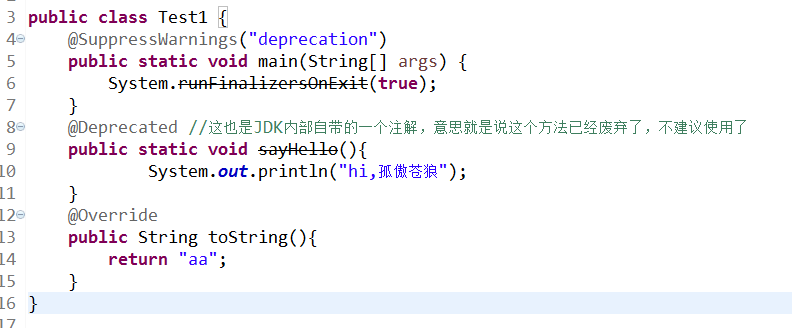
## 概念

注解是JDK1.5之后才有的新特性，JDK1.5之后内部提供的三个注解：

（1）@Deprecated：废弃的、过时的。

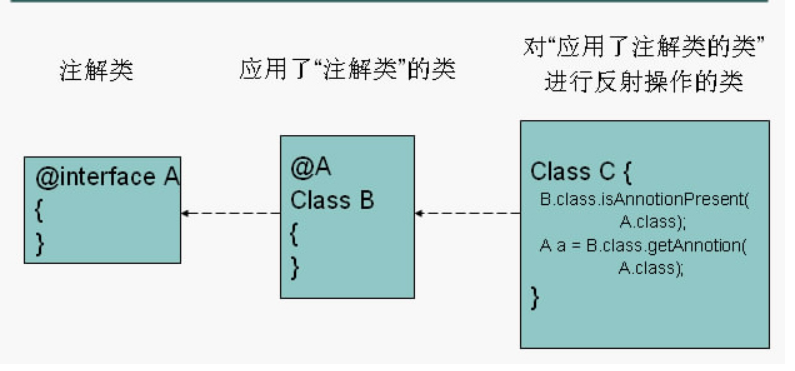
（2）@Override：重写、覆盖

（3）@SuppressWarnings：压缩、警告



总结：**注解(Annotation)相当于一种标记，在程序中加入注解就等于为程序打上某种标记**，没有加则等于没有任何标记，以后javac编译器、开发工具和其他程序可以通过反射来了解你的类及各种元素上有无何种标记，看你的程序有什么标记，就去干相应的事，**标记可以加在包、类，属性、方法，方法的参数以及局部变量上。**

注解的应用结构图：



注解就相当于一个你的源程序要调用一个类，在源程序中应用某个注解，得事先准备好这个注解类，就像你要调用某个类，得事先开发好这个类。

## 作用

1、生成文档。这是最常见的，也是java 最早提供的注解。常用的有@param @return 等。

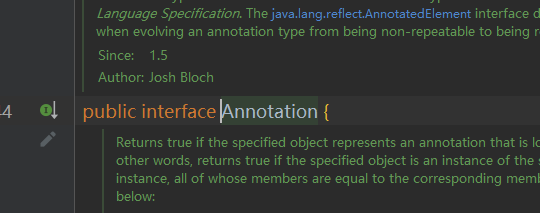
2、跟踪代码依赖性，实现替代配置文件的功能。

3、在编译时进行格式检查。如@override 放在方法前，如果你这个方法并不是覆盖了超类方法，则编译时就能检查出。

## Annotation接口

### 基本概念

### 源码



### 方法

#### int hashCode()

#### String toString();

#### Class<? extends Annotation> annotationType();

#### boolean equals(Object obj);

## 底层原理

注解的本质就是一个继承了 Annotation 接口的接口，底层实现就是一个 JDK 的动态代理类。

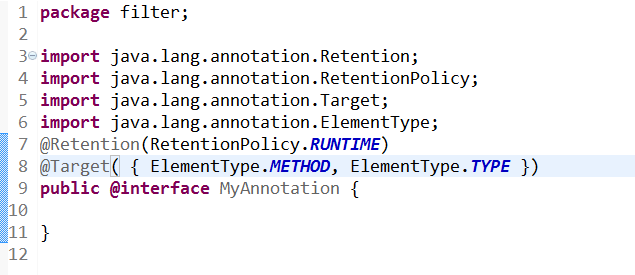
## 自定义注解

### 语法

public @interface MyAnnotation{}

### 代码示例

#### 1、定义注解

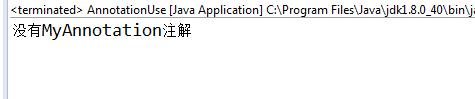


#### 2、把自定义的注解加到某个类上



用反射进行测试AnnotationUse类的定义上是否有@MyAnnotation注解。一旦在某个类上使用了@MyAnnotation注解，那么这个MyAnnotation类的实例对象annotation就会被创建出来了。

结果如图：



## 元注解

元注解的作用就是负责注解其他的注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。这些类型和它们所支持的类在java.lang.annotation包中可以找到。

### @Retention

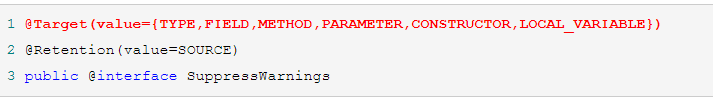
设置注解的生命周期，其三种取值：RetentionPolicy.SOURCE、RetentionPolicy.CLASS、RetentionPolicy.RUNTIME分别对应：Java源文件(.java文件)---->.class文件---->内存中的字节码。

说明：当在Java源程序上加了一个注解，这个Java源程序要由javac去编译，javac把java源文件编译成.class文件，在编译成class时可能会把Java源程序上的一些注解给去掉，java编译器(javac)在处理java源程序时，可能会认为这个注解没有用了，于是就把这个注解去掉了，那么此时在编译好的class中就找不到注解了， 这是编译器编译java源程序时对注解进行处理的第一种可能情况。假设java编译器在把java源程序编译成class时，没有把java源程序中的注解去掉，那么此时在编译好的class中就可以找到注解，当程序使用编译好的class文件时，需要用类加载器把class文件加载到内存中，class文件中的东西不是字节码，class文件里面的东西由类加载器加载到内存中去，类加载器在加载class文件时，会对class文件里面的东西进行处理，如安全检查，处理完以后得到的最终在内存中的二进制的东西才是字节码，类加载器在把class文件加载到内存中时也有转换，转换时是否把class文件中的注解保留下来，这也有说法，所以说一个注解的生命周期有三个阶段：java源文件是一个阶段，class文件是一个阶段，内存中的字节码是一个阶段**，**javac把java源文件编译成.class文件时，有可能去掉里面的注解，类加载器把.class文件加载到内存时也有可能去掉里面的注解，因此在自定义注解时就可以使用Retention注解指明自定义注解的生命周期，自定义注解的生命周期是在RetentionPolicy.SOURCE阶段(java源文件阶段)，还是在RetentionPolicy.CLASS阶段(class文件阶段)，或者是在RetentionPolicy.RUNTIME阶段(内存中的字节码运行时阶段)，根据JDK提供的API可以知道默认是在RetentionPolicy.CLASS阶段 (JDK的API写到：the retention policy defaults to RetentionPolicy.CLASS.)。

### @Target

@Target元注解决定了一个注解可以标识到哪些成分上，如标识在类身上，或者属性身上或者方法身上等成分，@Target默认值为任何元素(成分)。

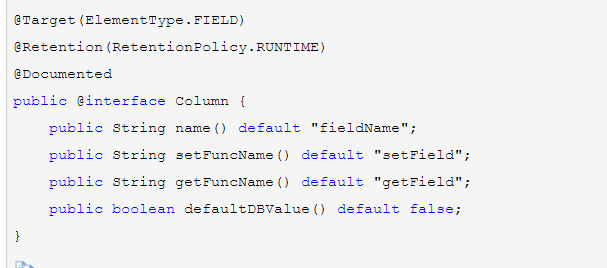
示例：



### @Documented

@Documented用于描述其它类型的annotation，应该被作为被标注的程序成员的公共API，因此可以被例如javadoc此类的工具文档化。Documented是一个标记注解，没有成员。说明：该注解将被包含在javadoc中。

示例代码：



### @Inherited

@Inherited 元注解是一个标记注解，@Inherited阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。说明子类可以继承父类中的该注解。

## 为注解增加属性

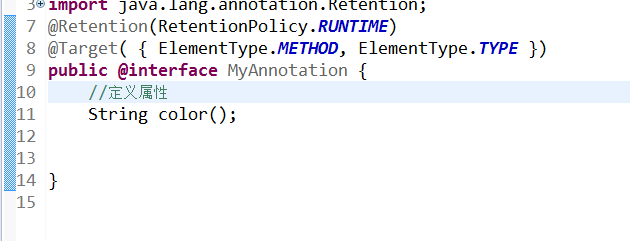
注解可以看成是一种特殊的类，既然是类，那自然可以为类添加属性。

### 添加属性

#### 语法

类型 属性名();

#### 示例



其实从代码的写法上来看，注解更像是一种特殊的接口，注解的属性定义方式就和接口中定义方法的方式一样，而应用了注解的类可以认为是实现了这个特殊的接口。

### 应用属性

示例代码：

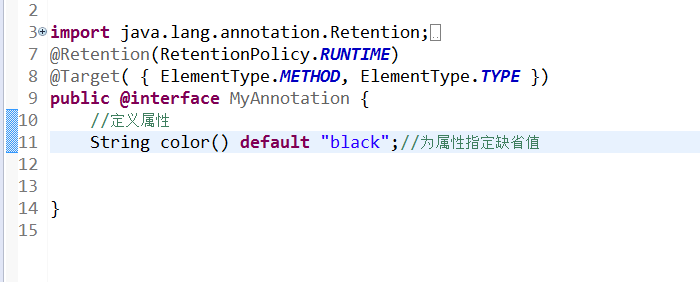


### 为属性指定缺省值（默认值）

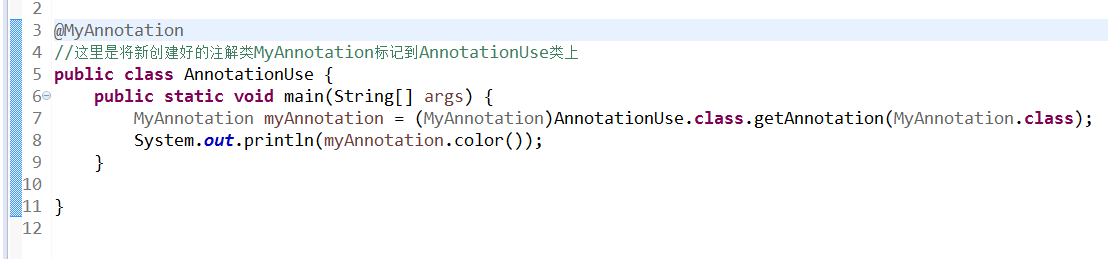
#### 语法

类型 属性名() default 默认值；

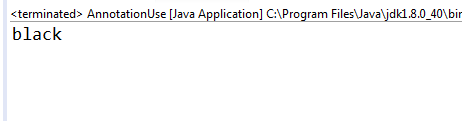
#### 示例代码



编写测试类：



输出结果：

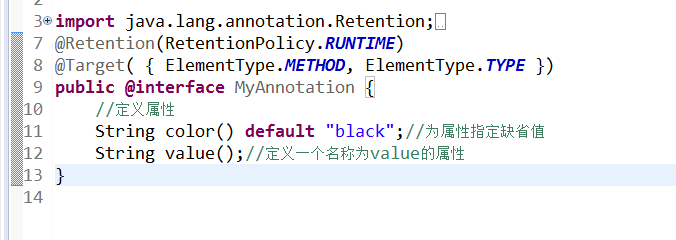


### value属性

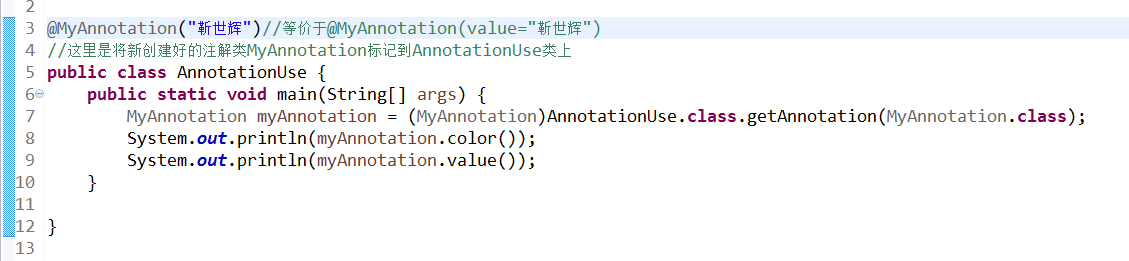
如果一个注解中有一个名称为value的属性，且你只想设置value属性(即其他属性都采用默认值或者你只有一个value属性)，那么可以省略掉“value=”部分。

例如：@SuppressWarnings("deprecation")

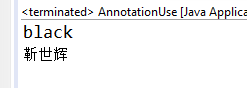
#### 代码示例



编写测试类：



运行结果：



### 为注解增加高级属性

#### 数组类型的属性

增加数组类型的属性：int[] arrayAttr() default {1,2,4};

应用数组类型的属性：@MyAnnotation(arrayAttr={2,4,5})

如果数组属性只有一个值，这时候属性值部分可以省略大括号，如：@MyAnnotation(arrayAttr=2)，这就表示数组属性只有一个值，值为2。

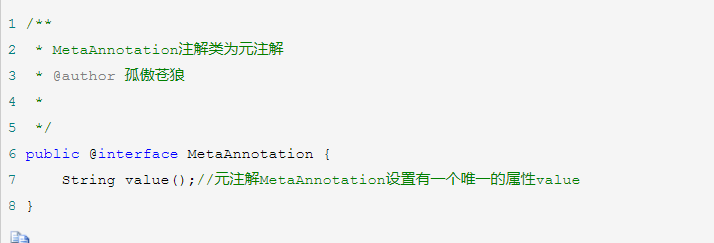
#### 枚举类型的属性

增加枚举类型的属性：EumTrafficLamp lamp() default EumTrafficLamp.RED;

应用枚举类型的属性：@MyAnnotation(lamp=EumTrafficLamp.GREEN)

#### 注解类型的属性

示例：

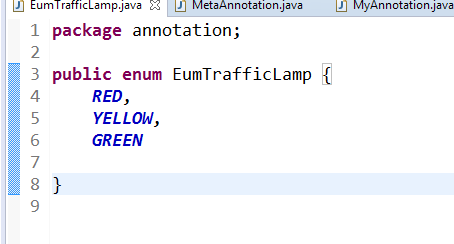


为注解添加一个注解类型的属性,并指定注解属性的缺省值：MetaAnnotation annotationAttr() default @MetaAnnotation("xdp");

### 注解综合测试

代码示例：

编写枚举类



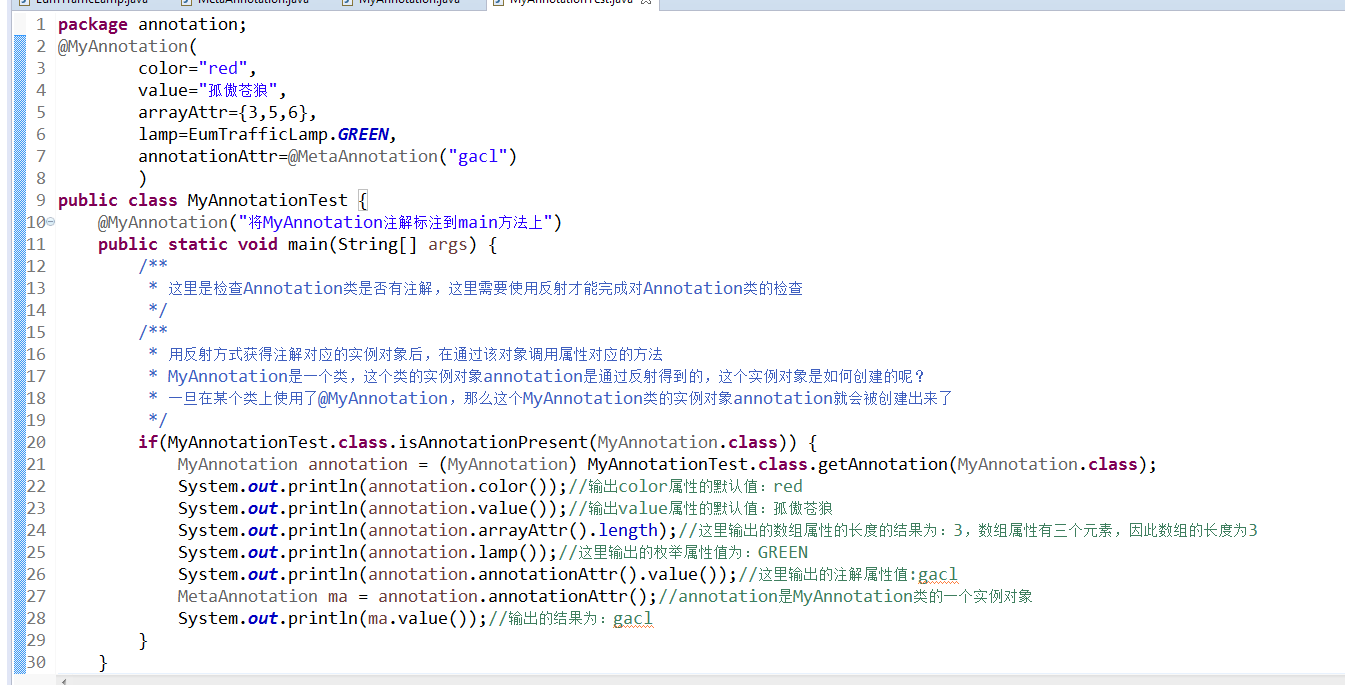
编写元注解类



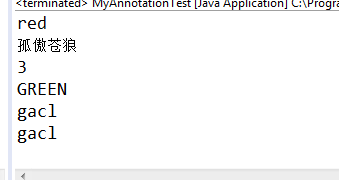
编写注解类



编写测试类：



运行结果：



# 正则表达式

## 语法

\ 转义字符

^ 匹配输入字符串的开始位置

$ 匹配输入字符串的结束位置

\* 匹配前面的子表达式任意次，0次或者多次。

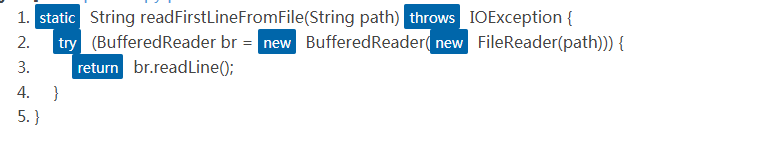
+ 匹配前面的子表达式一次或多次(大于等于1次）

？ 匹配前面的子表达式零次或一次

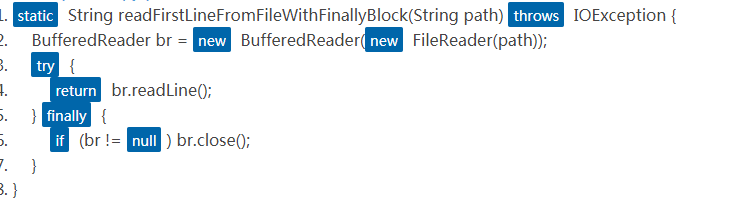
# Java7里try-with-resources

这个所谓的try-with-resources，是个语法糖。实际上就是自动调用资源的close()函数。

示例：



可以看到try语句多了个括号，而在括号里初始化了一个BufferedReader。这种在try后面加个括号，再初始化对象的语法就叫try-with-resources。实际上，相当于下面的代码（其实略有不同，下面会说明）：



很容易可以猜想到，这是编绎器自动在try-with-resources后面增加了判断对象是否为null，如果不为null，则调用close()函数的字节码。

说明：只有实现了java.lang.AutoCloseable接口，或者java.io.Closable（实际上继承自java.lang.AutoCloseable）接口的对象，才会自动调用其close()函数。有点不同的是java.io.Closable要求实现者保证close函数可以被重复调用。而AutoCloseable的close()函数则不要求是幂等的。使用try-with-resources的语法可以实现资源的自动回收处理，大大提高了代码的便利性，和mutil catch一样，是个好东西，用编绎器生成的字节码的角度来看，try-with-resources语法更加高效点。

# Java开发技巧

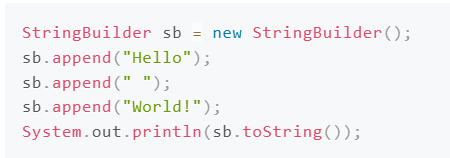
## 1. 熟练掌握你的 IDE 快捷键

熟悉你所使用的 IDE 的快捷键可以为你节省大量时间。节省在重复任务上的时间，就意味着有更多时间真正编写代码。虽然看起来很小，但这些快捷键加起来效果显著！

## 2、使用 StringBuilder 进行字符串操作

我们都知道在 Java 中使用 + 操作符拼接字符串非常简单，但你知道使用 StringBuilder 更高效吗？使用 + 拼接字符串时，Java 每次都会创建一个新的 String 对象，这可能会降低应用程序的速度。相反，使用 StringBuilder 可以提升性能。

示例：



当你在循环或处理大量数据时，StringBuilder 是你的好帮手。它可以减少内存使用并提高性能。

## 3、利用增强的 For 循环

增强的 for 循环（也叫作 "for-each" 循环）是遍历集合或数组的更简洁、更具可读性的方法。

示例：



它简洁、易读，并且避免了与索引操作相关的 bug。

## 4、利用 Java Streams 进行数据处理

Java Streams 是处理集合数据的强大工具。它们允许你以声明式方式处理数据，并帮助你编写更简洁、可读性更强的代码。

示例：



Streams 是进行数据过滤、映射和归约的理想工具。它们让你在 Java 中能够编写更干净的函数式风格代码。

## 5、使用 Optional 避免 NullPointerException

没有人喜欢 NullPointerException。它是 Java 中最常见且最令人沮丧的错误之一。Optional 是一个很棒的功能，可以帮助你避免这些意外。

示例：

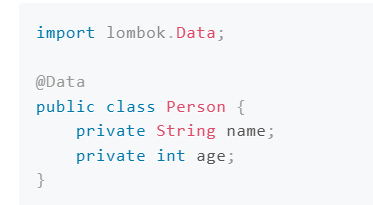


使用 Optional 能够鼓励更好的编码习惯，通过考虑 null 值的可能性，使代码更安全可靠。

## 6、使用 Lombok 减少样板代码

Lombok 是一个可以在编译时自动生成这些代码的库，节省了编写重复样板代码的时间。

示例：

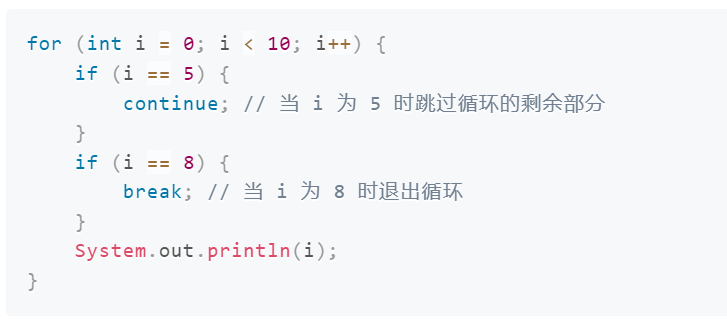


更少的样板代码意味着你可以更专注于实际的业务逻辑。Lombok 使你的代码库更加简洁且易于维护。

## 7、使用 Break 和 Continue 优化循环

循环是编程的基础，但如果使用不当，也会导致效率低下。break 和 continue 语句可以帮助你通过控制循环的流来优化代码。

示例：



有效的循环在处理大量数据或复杂逻辑时可以显著提升性能。

## 8、实现单例模式

单例模式确保一个类只有一个实例，并提供全局访问点。它在管理共享资源（如数据库连接）时特别有用。

示例：

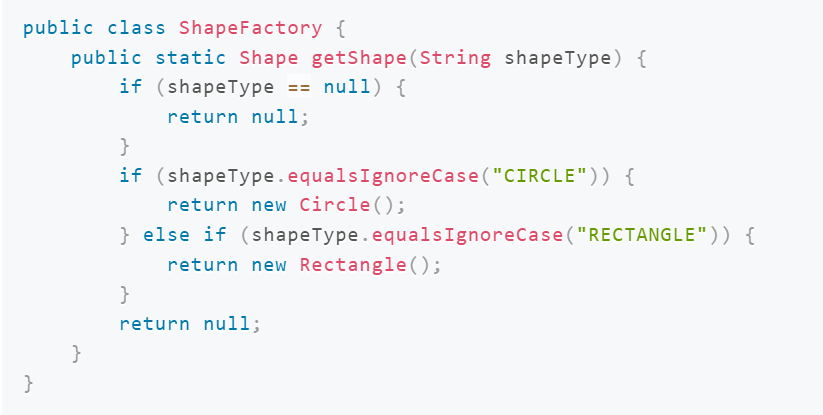


单例模式可以确保资源的高效使用和一致性，防止创建不必要的多个实例。

## 9、使用工厂模式创建对象

工厂模式是一种很好的方法，可以将对象的创建逻辑封装起来，使你的代码更加模块化和灵活。

示例：



它将对象的创建与客户端代码解耦，使得添加新类型更加容易，而无需修改现有代码。

## 10、使用 Collections.unmodifiableList 控制集合的不可修改性

当你需要从方法返回一个集合但希望确保它不能被修改时，可以使用 Collections.unmodifiableList。

示例：



这是强制实现不可变性的简单方法，能够防止集合被意外修改，从而避免潜在的 bug。

## 11、使用 Comparator.comparing 进行高效比较

需要排序对象列表？Comparator.comparing 可以让你轻松、简洁地实现。

示例：

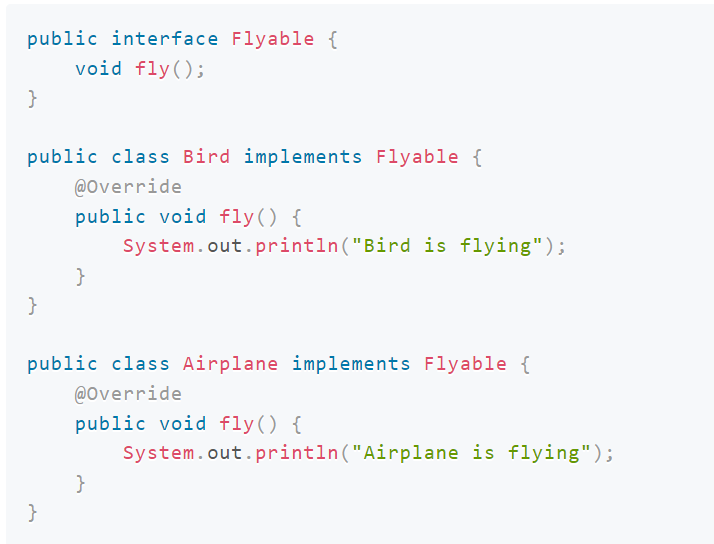


编写干净、高效的比较逻辑对于排序和排列集合至关重要，而 Comparator.comparing 简化了这个过程。

## 12、优先使用接口而不是抽象类

在 Java 中，接口通常比抽象类更加灵活。它们允许你定义多个类可以实现的约定，而不规定如何实现。

示例：

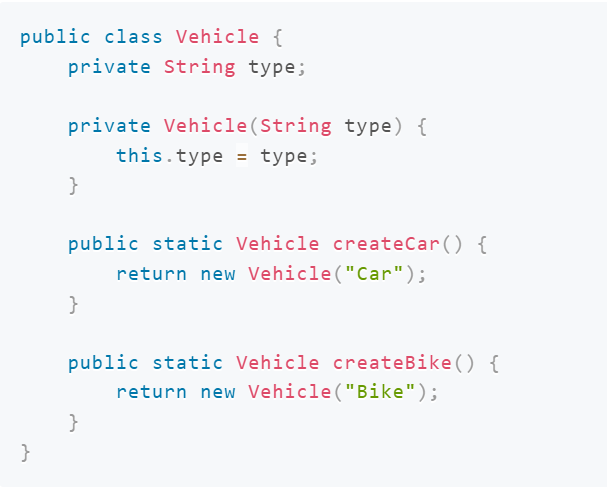


接口促进了松耦合和灵活性，使你的代码更易于维护和扩展。

## 13、使用静态工厂方法创建对象

与构造函数相比，考虑使用静态工厂方法来创建对象。它们可以有意义的名字，并返回子类型。

示例：



静态工厂方法提高了可读性，并使你的代码更直观，通过清晰地表明正在创建的对象类型。

## 14、使用依赖注入提升测试性

依赖注入（DI）允许你将依赖项（如服务或仓库）传递给类，而不是在类中直接实例化它们。这使代码更加模块化，也更容易测试。

示例：

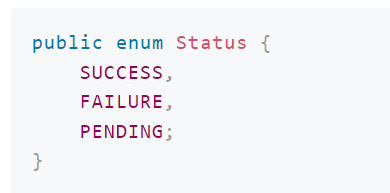


依赖注入促进了松耦合，使代码更容易测试和维护。这是现代软件开发实践的基石之一。

## 15、使用枚举代替常量

当你有一组相关的常量时，使用 enum 而不是静态的 final 常量。enum 类型更强大，并且提供类型安全。

示例：



enum 是表示固定常量集的类型安全方式，且它可以包含方法和字段，功能比简单常量更强大。

## 16、使用try-with-resources管理资源

手动管理文件流或数据库连接等资源可能会出错。Java 的 try-with-resources 能确保操作完成后自动关闭资源。

示例：



自动资源管理防止内存泄漏，使代码更干净、更安全。

## 17、利用方法引用简化代码

方法引用是 lambda 表达式调用方法的简写，它让代码更简洁易读。

示例：



方法引用提高了代码可读性，减少了冗余，特别是在函数式编程中。

## 18、明智地使用final关键字

final 关键字可以用来让变量、方法和类不可变，从而防止意外更改。

示例：

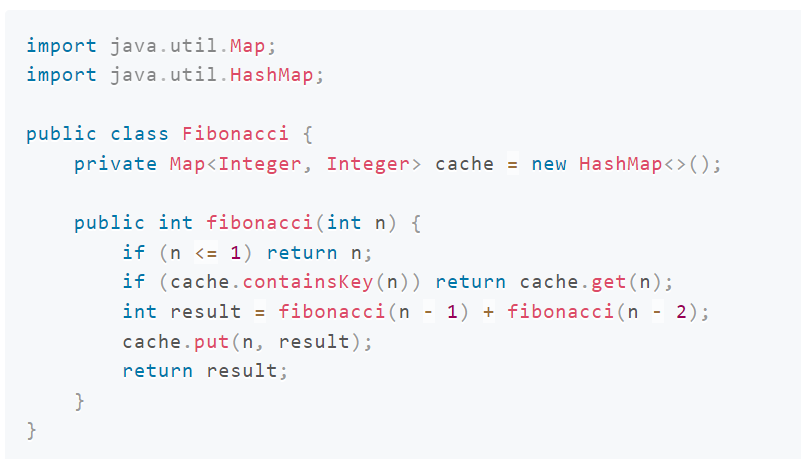


使用 final 帮助强制实现不可变性，使代码更具可预测性。

## 19、实现缓存以提高性能

缓存是一种存储耗时计算或频繁访问数据的技术，可以加速应用程序的运行。

示例：



缓存可以显著提高应用程序的性能，减少重复计算。

## 20、使用 @Override 注解

在重写方法时，总是使用 @Override 注解。它确保你确实在重写父类的方法。

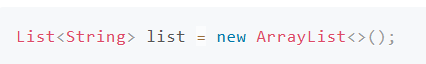


重要性： @Override 注解提供编译时检查，防止方法签名不匹配引发的错误。

## 21、利用泛型操作符

Java 7 引入的泛型操作符 <>，允许在编译器能够推断类型信息时省略泛型类型。

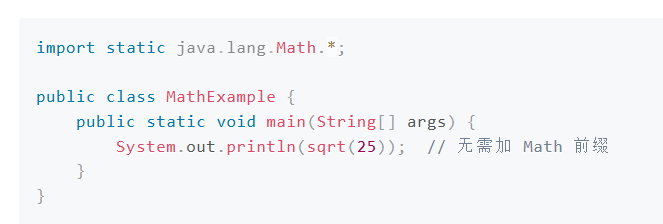
示例：



重要性： 钻石操作符减少了冗余，使代码更简洁可读。

## 22、使用静态导入提高可读性

静态导入允许你导入类的静态成员（字段和方法），使你可以在使用时省略类名。



重要性： 静态导入使代码更简洁，尤其是在频繁使用静态成员时。

## 23. 使用 Map.of 和 List.of 创建不可变集合

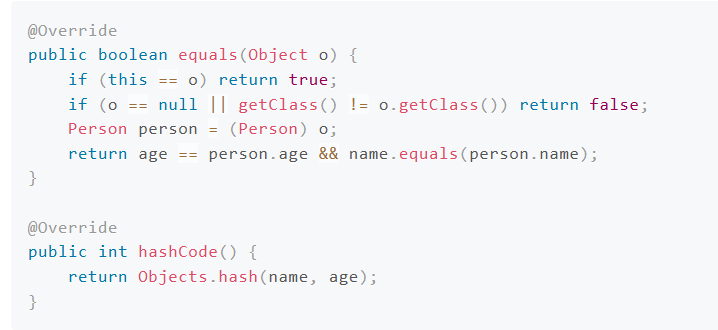
Java 9 引入了 Map.of() 和 List.of() 方法，可以轻松创建不可变的 Map 和 List。这些集合一旦创建就不能修改。



重要性： 不可变集合确保数据不能在创建后修改，有助于防止 bug，使代码更具可预测性和线程安全性。

## 24、理解 equals 和 hashCode 合约

如果重写了 equals()，你也必须重写 hashCode() 以维护它们之间的一致性，确保在基于哈希的集合中正常工作。



**重要性：** 正确实现 equals 和 hashCode 对于保持哈希集合（如 HashMap 和 HashSet）中的一致性至关重要。

## 25、利用 Java 内置的函数式接口

Java 提供了多个内置的函数式接口，如 Function、Predicate 和 Supplier，你可以使用它们编写更简洁和函数式风格的代码。



重要性： 使用内置函数式接口让代码更具表现力，充分利用了 Java 的函数式编程特性。

## 26、使用 CompletableFuture 进行异步编程

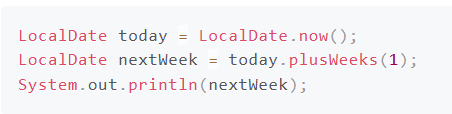
CompletableFuture 是一个处理异步编程的多功能类，它可以帮助你构建非阻塞和异步的应用程序。



重要性： CompletableFuture 帮助你编写高效的非阻塞代码，这对于构建可扩展的应用程序至关重要。

## 27、使用 java.time 包处理日期和时间

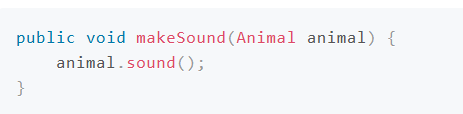
在 Java 8 中引入的 java.time 包提供了一个全面且现代化的日期和时间处理 API，取代了过时的 Date 和 Calendar 类。



重要性: java.time 提供了一种更直观且可靠的方式来处理日期和时间，降低了错误的风险。

## 28、利用多态性编写灵活代码

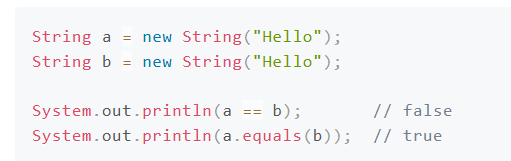
多态性允许你通过同一接口处理不同子类的对象，使代码更加灵活和可扩展。



重要性: 多态性使你能够编写更加通用和可重用的代码，便于扩展和维护。

## 29、理解 == 和 equals() 的区别

在 Java 中，== 检查的是引用的相等性，而 equals() 检查的是值的相等性。比较值时，尤其是对象，始终应使用 equals()。



重要性: 使用 == 而非 equals() 可能导致微妙的错误，特别是在比较字符串或对象时。

## 30、应用 DRY（不要重复自己）原则

DRY 原则鼓励你避免代码重复。如果你发现自己多次编写相同的代码，考虑将其重构为一个方法或类。

