# 默认方法

## 说明：

Java 8 新增了接口的默认方法。

简单说，默认方法就是接口可以有实现方法，而且不需要实现类去实现其方法。

我们只需在方法名前面加个default关键字即可实现默认方法。

## 为什么要有这个特性？

首先，之前的接口是个双刃剑，好处是面向抽象而不是面向具体编程，缺陷是，当需要修改接口时候，需要修改全部实现该接口的类，目前的java 8之前的集合框架没有foreach方法，通常能想到的解决办法是在JDK里给相关的接口添加新的方法及实现。然而，对于已经发布的版本，是没法在给接口添加新方法的同时不影响已有的实现。所以引进的默认方法。他们的目的是为了解决接口的修改与现有的实现不兼容的问题。

## 1.1语法

默认方法语法格式如下：

**interface** Vehicle{

**default** **void** print(){

System.***out***.println("我是一辆车！");

}

}

## 1.2多个默认方法

一个接口有默认方法，考虑这样的情况，一个类实现了多个接口，且这些接口有相同的默认方法，以下实例说明了这种情况的解决方法

**public** **interface** Vehicle {

**default** **void** print() {

System.out.println("我是一辆车!");

}

}

**public** **interface** FourWheeler {

**default** **void** print() {

System.out.println("我是一辆四轮车!");

}

}

第一个解决方案是创建自己的默认方法，来覆盖重写接口的默认方法：

**public** **class** Car **implements** Vehicle, FourWheeler {

@Override

**public** **void** print() {

System.***out***.println("我是一辆四轮汽车!");

}

}

第二种解决方案可以使用 super 来调用指定接口的默认方法：

**public** **class** Car **implements** Vehicle, FourWheeler {

@Override

**public** **void** print() {

vehicle.**super**.print();

}

}

## 1.3 静态默认方法

Java 8 的另一个特性是接口可以声明（并且可以提供实现）静态方法。例如：

**public** **interface** Vehicle {

**default** **void** print() {

System.***out***.println("我是一辆车!");

}

// 静态方法

**static** **void** blowHorn() {

System.***out***.println("按喇叭!!!");

}

}

//静态方法调用

**public** **class** DefaultMethod {

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

Vehicle.*blowHorn*();

}

**interface** Vehicle{

**static** **void** blowHorn(){

System.***out***.println("按喇叭！");

}

}

}

## 1.4 默认方法实例

我们可以通过以下代码来了解关于默认方法的使用，可以将代码放入 DefaultMethod.java 文件中：

DefaultMethod：

**public** **class** DefaultMethod {

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

Vehicle vehicle = **new** Car();

vehicle.print();

}

**interface** Vehicle{

**default** **void** print(){

System.***out***.println("我是一辆车！");

}

**static** **void** blowHorn(){

System.***out***.println("按喇叭！");

}

}

**interface** FourWheeler{

**default** **void** print(){

System.***out***.println("我是一辆四轮车！");

}

}

}

Car：

**import** defaultmethod.DefaultMethod.FourWheeler;

**import** defaultmethod.DefaultMethod.Vehicle;

**public** **class** Car **implements** Vehicle, FourWheeler{

@Override

**public** **void** print(){

Vehicle.**super**.print();

FourWheeler.**super**.print();

Vehicle.*blowHorn*();

System.***out***.println("我是一辆汽车！");

}

}

执行以上脚本，输出结果为：

我是一辆车!

我是一辆四轮车!

按喇叭!!!

我是一辆汽车!

## 示例：



# Optional 类

## 说明：

Optional 类是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。

Optional 是个容器：它可以保存类型T的值，或者仅仅保存null。Optional提供很多有用的方法，这样我们就不用显式进行空值检测。

Optional 类的引入很好的解决空指针异常。

## 2.1类声明

以下是一个 java.util.Optional<T> 类的声明：

**public** **final** **class** Optional<T> **extends** Object

## 2.2 类方法

1、**static** <T> Optional<T> empty()

返回空的 Optional 实例。

2、**boolean** equals(Object obj)

判断其他对象是否等于 Optional。

3、Optional<T> filter(Predicate<? **super** <T> predicate)

如果值存在，并且这个值匹配给定的 predicate，返回一个Optional用以描述这个值，否则返回一个空的Option Optional。

4、<U> Optional<U> flatMap(Function<? **super** T,Optional<U>> mapper)

如果值存在，返回基于Optional包含的映射方法的值，否则返回一个空的Optional

5、T get()

如果在这个Optional中包含这个值，返回值，否则抛出异常：NoSuchElementException

6、**int** hashCode()

返回存在值的哈希码，如果值不存在返回 0。

7、**void** ifPresent(Consumer<? **super** T> consumer)

如果值存在则使用该值调用 consumer , 否则不做任何事情。

8、**boolean** isPresent()

如果值存在则方法会返回true，否则返回 **false**。

9、<U>Optional<U> map(Function<? **super** T,? **extends** U> mapper)

如果存在该值，提供的映射方法，如果返回非null，返回一个Optional描述结果。

10、**static** <T> Optional<T> of(T value)

返回一个指定非null值的Optional。

11、**static** <T> Optional<T> ofNullable(T value)

如果为非空，返回 Optional 描述的指定值，否则返回空的 Optional。

12、T orElse(T other)

如果存在该值，返回值，否则返回 other。

13、T orElseGet(Supplier<? **extends** T> other)

如果存在该值，返回值，否则触发 other，并返回 other 调用的结果。

14、<X **extends** Throwable> T orElseThrow(Supplier<? **extends** X> exceptionSupplier)

如果存在该值，返回包含的值，否则抛出由 Supplier 继承的异常

15、String toString()

返回一个Optional的非空字符串，用来调试

注意： 这些方法是从 java.lang.Object 类继承来的。

## 2.3 Optional 实例

**import** java.util.Optional;

**public** **class** OptionalMain {

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

Integer value1 = **null**;

Integer value2 = **new** Integer(10);

// Optional.ofNullable - 允许传递为 null 参数

Optional<Integer> a = Optional.*ofNullable*(value1);

// Optional.of - 如果传递的参数是 null，抛出异常 NullPointerException

Optional<Integer> b = Optional.*of*(value2);

System.***out***.println(*sum*(a, b));

}

**private** **static** Integer sum(Optional<Integer> a, Optional<Integer> b) {

// Optional.isPresent - 判断值是否存在

System.***out***.println("第一个参数值存在:" + a.isPresent());

System.***out***.println("第二个参数值存在:" + b.isPresent());

// Optional.orElse - 如果值存在，返回它，否则返回默认值

Integer value1 = a.orElse(**new** Integer(0));

//System.out.println(a.get());//java.util.NoSuchElementException: No value present

//Optional.get - 获取值，值需要存在

Integer value2 = b.get();

**return** value1 + value2;

}

}

执行以上脚本，输出结果为：

第一个参数值存在:false

第二个参数值存在:true

10

## 示例：



# 日期时间 API

Java 8通过发布新的Date-Time API (JSR 310)来进一步加强对日期与时间的处理。

在旧版的Java 中，日期时间API 存在诸多问题，其中有：  非线程安全 − java.util.Date 是非线程安全的，所有的日期类都是可变的，这是Java日期类最大的问题之一。

设计很差 − Java的日期/时间类的定义并不一致，在java.util和java.sql的包中都有日期类，此外用于格式化和解析的类在java.text包中定义。java.util.Date同时包含日期和时间，而java.sql.Date仅包含日期，将其纳入java.sql包并不合理。另外这两个类都有相同的名字，这本身就是一个非常糟糕的设计。

时区处理麻烦 − 日期类并不提供国际化，没有时区支持，因此Java引入了java.util.Calendar和java.util.TimeZone类，但他们同样存在上述所有的问题。

Java 8 在 java.time 包下提供了很多新的 API。以下为两个比较重要的 API：

1. Local(本地) − 简化了日期时间的处理，没有时区的问题。

2. Zoned(时区) − 通过制定的时区处理日期时间。

新的java.time包涵盖了所有处理日期，时间，日期/时间，时区，时刻（instants），过程（during）与时钟（clock）的操作。

## 3.1本地化日期时间 API

LocalDate/LocalTime 和 LocalDateTime 类可以在处理时区不是必须的情况。代码如下：

//本地化日期时间 API

**public** **static** **void** testLocalDateTime() {

// 获取当前的日期时间

LocalDateTime currentTime = LocalDateTime.now();

System.out.println("当前时间: " + currentTime);

LocalDate date1 = currentTime.toLocalDate();

System.out.println("date1: " + date1);

Month month = currentTime.getMonth();

**int** day = currentTime.getDayOfMonth();

**int** seconds = currentTime.getSecond();

System.out.println("月: " + month + ", 日: " + day + ", 秒: " + seconds);

LocalDateTime date2 = currentTime.withDayOfMonth(10).withYear(2012);

System.out.println("date2: " + date2);

// 12 december 2014

LocalDate date3 = LocalDate.of(2014, Month.DECEMBER, 12);

System.out.println("date3: " + date3);

// 22 小时 15 分钟

LocalTime date4 = LocalTime.of(22, 15);

System.out.println("date4: " + date4);

// 解析字符串

LocalTime date5 = LocalTime.parse("20:15:30");

System.out.println("date5: " + date5);

}

## 3.2 使用时区的日期时间API

如果我们需要考虑到时区，就可以使用时区的日期时间API：

//使用时区的日期时间API

**public** **static** **void** testZonedDateTime() {

// 获取当前时间日期

ZonedDateTime date1 = ZonedDateTime.parse("2015-12-03T10:15:30+05:30[Asia/Shanghai]");

System.out.println("date1: " + date1);

ZoneId id = ZoneId.of("Europe/Paris");

System.out.println("ZoneId: " + id);

ZoneId currentZone = ZoneId.systemDefault();

System.out.println("当期时区: " + currentZone);

}

**示例：**

****

# Base64

## 说明：

在Java8中，Base64编码已经成为Java类库的标准。

Java 8 内置了 Base64 编码的编码器和解码器。

Base64工具类提供了一套静态方法获取下面三种BASE64编解码器：

1、基本：输出被映射到一组字符A-Za-z0-9+/，编码不添加任何行标，输出的解码仅支持A-Za-z0-9+/。

2、URL：输出映射到一组字符A-Za-z0-9+\_，输出是URL和文件。

3、MIME：输出隐射到MIME友好格式。输出每行不超过76字符，并且使用'\r'并跟随'\n'作为分割。编码输出最后没有行分割。

## 4.1内嵌类



## 4. 2 方法



## Base64 实例

以下实例演示了Base64的使用:

**try** {

// 使用基本编码

String base64encodedString = Base64.*getEncoder*().encodeToString("runoob?java8".getBytes("utf-8"));

System.out.println("Base64 编码字符串 (基本) :" + base64encodedString);

// 解码

**byte**[] base64decodedBytes = Base64.*getDecoder*().decode(base64encodedString);

System.out.println("原始字符串: " + **new** String(base64decodedBytes, "utf-8"));

base64encodedString = Base64.getUrlEncoder().encodeToString("TutorialsPoint?java8".getBytes("utf-8"));

System.out.println("Base64 编码字符串 (URL) :" + base64encodedString);

base64decodedBytes = Base64.getUrlDecoder().decode(base64encodedString);

System.out.println("原始字符串(URL): " + **new** String(base64decodedBytes, "utf-8"));

StringBuilder stringBuilder = **new** StringBuilder();

**for** (**int** i = 0; i < 10; ++i) {

stringBuilder.append(UUID.*randomUUID*().toString());

}

**byte**[] mimeBytes = stringBuilder.toString().getBytes("utf-8");

String mimeEncodedString = Base64.*getMimeEncoder*().encodeToString(mimeBytes);

System.out.println("Base64 编码字符串 (MIME) :" + mimeEncodedString);

} **catch** (UnsupportedEncodingException e) {

System.out.println("Error :" + e.getMessage());

}

## 示例：

