This repository Search Pull requests Issues Gist

winterbe / **java8-tutorial**

Watch 483 Unstar 6,048 Fork 945

Code Issues 2 Pull requests 3 Projects 0 Wiki Pulse Graphs

Modern Java - A Guide to Java 8 http://winterbe.com

140 commits **1** branch **0** releases **10** contributors MIT

Branch: master New pull request Create new file Upload files Find file Clone or download

winterbe committed on **GitHub** Update README.md Latest commit 85dae28 16 days ago

res	Inroduce sub packages	2 years ago
src/com/winterbe/java8/samples	BiConsumer Example	11 months ago
.gitignore	Added package custom	a year ago
LICENSE	Happy new year 2016	10 months ago
README.md	Update README.md	16 days ago

README.md

Modern Java - A Guide to Java 8 Java 8指南

If you like this project, please give me a star. ★

本教程引导你一步一步熟悉所有新的语言功能。
通过简短简单的代码示例教你如何使用默认的接口方法、
Lambda表达式、方法引用和可重复的注解

> "Java is still not dead—and people are starting to figure that out."

Welcome to my introduction to Java 8. This tutorial guides you step by step through all new language features. Backed by short and simple code samples you'll learn how to use default interface methods, lambda expressions, method references and repeatable annotations. At the end of the article you'll be familiar with the most recent API changes like streams, functional interfaces, map extensions and the new Date API. **No walls of text, just a bunch of commented code snippets. Enjoy!**

文本没有墙壁，只是一堆注释的代码片段。

This article was originally posted on my blog. You should follow me on Twitter.

Table of Contents

Default Methods for Interfaces 接口的默认方法

Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the `default` keyword. This feature is also known as virtual extension methods. 虚拟扩展方法

Here is our first example:

```
interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

Besides the abstract method `calculate` the interface `Formula` also defines the default method `sqrt`. Concrete classes only have to implement the abstract method `calculate`. The default method `sqrt` can be used out of the box.

```
Formula formula = new Formula() {
    @Override
    public double calculate(int a) {
        return sqrt(a * 100);
    }
};

formula.calculate(100);     // 100.0
formula.sqrt(16);           // 4.0
```

匿名对象

The formula is implemented as an anonymous object. The code is quite verbose: 6 lines of code for such a simple calculation of `sqrt(a * 100)`. As we'll see in the next section, there's a much nicer way of implementing single method objects in Java 8. Java 8中，实现单个方法的对象有更好的方式

Lambda expressions Lambda表达式

Let's start with a simple example of how to sort a list of strings in prior versions of Java:

如何排序字符串列表

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

The static utility method `Collections.sort` accepts a list and a comparator in order to sort the elements of the given list. You often find yourself creating anonymous comparators and pass them to the sort method.

Instead of creating anonymous objects all day long, Java 8 comes with a much shorter syntax, **lambda expressions**:

创建匿名对象，Java 8配备了一个更简短的语法：Lambda表达式

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

代码是更短和更容易阅读

As you can see the code is much shorter and easier to read. But it gets even shorter:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

对于一行方法体，可以忽略大括号和return关键字

For one line method bodies you can skip both the braces `{}` and the `return` keyword. But it gets even shorter:

```
names.sort((a, b) -> b.compareTo(a));
```

Java编译器知道该参数的类型

List now has a `sort` method. Also the java compiler is aware of the parameter types so you can skip them as well. Let's dive deeper into how lambda expressions can be used in the wild.

深入了解如何使用Lambda表达式

Functional Interfaces 函数式接口

函数式接口必须包含一个抽象方法声明

How does lambda expressions fit into Java's type system? Each lambda corresponds to a given type, specified by an interface. A so called *functional interface* must contain **exactly one abstract method** declaration. Each lambda expression of that type will be matched to this abstract method. Since default methods are not abstract you're free to add default methods to your functional interface.

We can use arbitrary interfaces as lambda expressions as long as the interface only contains one abstract method. To ensure that your interface meet the requirements, you should add the `@FunctionalInterface` annotation. The compiler is aware of this annotation and throws a compiler error as soon as you try to add a second abstract method declaration to the interface.

注解标记：编译时就报错

Example:

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
```

```
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
```

```
System.out.println(converted);    // 123
```

Keep in mind that the code is also valid if the `@FunctionalInterface` annotation would be omitted.

Method and Constructor References 方法和构造器引用

The above example code can be further simplified by utilizing static method references:

静态方法引用

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);   // 123
```

通过::关键字传入方法或构造器引用

Java 8 enables you to pass references of methods or constructors via the `::` keyword. The above example shows how to reference a static method. But we can also reference object methods:

对象方法引用

```
class Something {
    String startsWith(String s) {
        return String.valueOf(s.charAt(0));
    }
}



Something something = new Something();
Converter<String, String> converter = something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted);    // "J"
```

Let's see how the `::` keyword works for constructors. First we define an example class with different constructors:

```
class Person {
    String firstName;
    String lastName;

    Person() {}

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Next we specify a person factory interface to be used for creating new persons:

```
interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
}
```

构造方法引用

Instead of implementing the factory manually, we glue everything together via constructor references:

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

通过Person::new创建一个构造方法引用，Java编译器通过匹配调用方法签名来自动选择正确的构造器

We create a reference to the Person constructor via `Person::new`. The Java compiler automatically chooses the right constructor by matching the signature of `PersonFactory.create`.

Lambda Scopes Lambda作用域

Accessing outer scope variables from lambda expressions is very similar to anonymous objects. You can access final variables from the local outer scope as well as instance fields and static variables.

访问本地外部作用域的final变量；实例变量和静态变量

Accessing local variables 局部变量

We can read final local variables from the outer scope of lambda expressions:

从Lambda表达式的外部作用域读取final本地变量

```
final int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

But different to anonymous objects the variable `num` does not have to be declared final. This code is also valid:

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

However `num` must be implicitly final for the code to compile. The following code does **not** compile:

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);
num = 3;
```

Writing to `num` from within the lambda expression is also prohibited.

Accessing fields and static variables 实例字段和静态变量

In contrast to local variables, we have both read and write access to instance fields and static variables from within lambda expressions. This behaviour is well known from anonymous objects.

```
class Lambda4 {
    static int outerStaticNum;
    int outerNum;

    void testScopes() {
        Converter<Integer, String> stringConverter1 = (from) -> {
            outerNum = 23;
            return String.valueOf(from);
        };

        Converter<Integer, String> stringConverter2 = (from) -> {
            outerStaticNum = 72;
            return String.valueOf(from);
        };
    }
}
```

Accessing Default Interface Methods 默认接口方法

Remember the formula example from the first section? Interface `Formula` defines a default method `sqrt` which can be accessed from each formula instance including anonymous objects. This does not work with lambda expressions.

Default methods **cannot** be accessed from within lambda expressions. The following code does not compile:

默认方法不能在Lambda表达式中被访问

```
Formula formula = (a) -> sqrt(a * 100);
```

Built-in Functional Interfaces 内置的函数式接口

The JDK 1.8 API contains many built-in functional interfaces. Some of them are well known from older versions of Java like `Comparator` or `Runnable`. Those existing interfaces are extended to enable Lambda support via the `@FunctionalInterface` annotation.

But the Java 8 API is also full of new functional interfaces to make your life easier. Some of those new interfaces are well known from the Google Guava library. Even if you're familiar with this library you should keep a close eye on how those interfaces are extended by some useful method extensions. Guava提供的接口是如何通过一些有用的方法扩展被扩展的

Predicates 谓词：一个参数的布尔值函数

Predicates are boolean-valued functions of one argument. The interface contains various default methods for composing predicates to complex logical terms (and, or, negate)

```
Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo");              // true
predicate.negate().test("foo");     // false

Predicate<Boolean> nonNull = Objects::nonNull;
Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
```

Functions 函数：接受一个参数并产生一个结果

Functions accept one argument and produce a result. Default methods can be used to chain multiple functions together (compose, andThen).

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123");     // "123"
```

Suppliers 供应者：产生一个给定泛型类型的结果

Suppliers produce a result of a given generic type. Unlike Functions, Suppliers don't accept arguments.

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();   // new Person
```

Consumers 消费者：表示对单个输入参数要执行的操作

Consumers represent operations to be performed on a single input argument.

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
greeter.accept(new Person("Luke", "Skywalker"));
```

Comparators 比较器

Comparators are well known from older versions of Java. Java 8 adds various default methods to the interface.

```
Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);             // > 0
comparator.reversed().compare(p1, p2);  // < 0
```

Optionals 可选项：一个值的简单容器

防止空指针异常的漂亮实用程序

Optionals are not functional interfaces, but nifty utilities to prevent `NullPointerException`. It's an important concept for the next section, so let's have a quick look at how Optionals work.

Optional is a simple container for a value which may be null or non-null. Think of a method which may return a non-null result but sometimes return nothing. Instead of returning `null` you return an `Optional` in Java 8.

```
Optional<String> optional = Optional.of("bam");

optional.isPresent();           // true
optional.get();                 // "bam"
optional.orElse("fallback");    // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0)));     // "b"
```

Streams 流：表示元素序列

在流上面可以执行一个或多个运算

A `java.util.Stream` represents a sequence of elements on which one or more operations can be performed. Stream operations are either *intermediate* or *terminal*. While terminal operations return a result of a certain type, intermediate operations return the stream itself so you can chain multiple method calls in a row. Streams are created on a source, e.g. a `java.util.Collection` like lists or sets (maps are not supported). Stream operations can either be executed sequentially or parallely.

终止运算返回某种类型的结果，中间运算返回流本身

流由一个资源创建，流运算可以被顺序或并发地执行

> Streams are extremely powerful, so I wrote a separate Java 8 Streams Tutorial. You should also check out Stream.js, a JavaScript port of the Java 8 Streams API.

Let's first look how sequential streams work. First we create a sample source in form of a list of strings:

```
List<String> stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

Collections in Java 8 are extended so you can simply create streams either by calling `Collection.stream()` or `Collection.parallelStream()`. The following sections explain the most common stream operations.

Filter 过滤器：接受一个谓词函数来过滤流中的所有元素

Filter accepts a predicate to filter all elements of the stream. This operation is *intermediate* which enables us to call another stream operation (`forEach`) on the result. ForEach accepts a consumer to be executed for each element in the filtered stream. ForEach is a terminal operation. It's `void`, so we cannot call another stream operation.

```
stringCollection
```

```
        .stream()
        .filter((s) -> s.startsWith("a"))
        .forEach(System.out::println);

// "aaa2", "aaa1"
```

Sorted 排序：一个中间运算，它返回流的一个有序视图

Sorted is an *intermediate* operation which returns a sorted view of the stream. The elements are sorted in natural order unless you pass a custom `Comparator`.

```
stringCollection
    .stream()
    .sorted()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa1", "aaa2"
```

不会操作背后集合的顺序

Keep in mind that `sorted` does only create a sorted view of the stream without manipulating the ordering of the backed collection. The ordering of `stringCollection` is untouched: 原集合的顺序是原封不动的

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Map 映射：通过给定函数将每个元素转换为另一个对象

The *intermediate* operation `map` converts each element into another object via the given function. The following example converts each string into an upper-cased string. But you can also use `map` to transform each object into another type. The generic type of the resulting stream depends on the generic type of the function you pass to `map`.

```
stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

终止运算

Match 匹配：用来检查是否某些谓词匹配流

Various matching operations can be used to check whether a certain predicate matches the stream. All of those operations are *terminal* and return a boolean result.

```
boolean anyStartsWithA =
    stringCollection
        .stream()
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);      // true

boolean allStartsWithA =
    stringCollection
        .stream()
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);      // false
```

```
boolean noneStartsWithZ =
    stringCollection
        .stream()
        .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);      // true
```

Count 计数：返回流中元素的数量

Count is a *terminal* operation returning the number of elements in the stream as a `long`.

mapToLong(e -> 1L).sum()

```
long startsWithB =
    stringCollection
        .stream()
        .filter((s) -> s.startsWith("b"))
        .count();

System.out.println(startsWithB);    // 3
```

Reduce 归约：使用给定函数在流中的元素上执行一次归约

This *terminal* operation performs a reduction on the elements of the stream with the given function. The result is an `Optional` holding the reduced value.

```
Optional<String> reduced =
    stringCollection
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

Parallel Streams 并行流

As mentioned above streams can be either sequential or parallel. Operations on sequential streams are performed on a single thread while operations on parallel streams are performed concurrently on multiple threads.

The following example demonstrates how easy it is to increase the performance by using parallel streams.

First we create a large list of unique elements: 如何通过使用并行流来提高性能是多么容易

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
    UUID uuid = UUID.randomUUID();
    values.add(uuid.toString());
}
```

Now we measure the time it takes to sort a stream of this collection.

Sequential Sort 顺序排序

```
long t0 = System.nanoTime();

long count = values.stream().sorted().count();
System.out.println(count);
```

```
long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("sequential sort took: %d ms", millis));

// sequential sort took: 899 ms
```

Parallel Sort 并行排序

```
long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("parallel sort took: %d ms", millis));

// parallel sort took: 472 ms
```

As you can see both code snippets are almost identical but the parallel sort is roughly 50% faster. All you have to do is change `stream()` to `parallelStream()` .

Maps 映射

As already mentioned maps do not directly support streams. There's no `stream()` method available on the `Map` interface itself, however you can create specialized streams upon the keys, values or entries of a map via `map.keySet().stream()` , `map.values().stream()` and `map.entrySet().stream()` .

Furthermore maps support various new and useful methods for doing common tasks.

```
Map<Integer, String> map = new HashMap<>();

for (int i = 0; i < 10; i++) {
    map.putIfAbsent(i, "val" + i);
}

map.forEach((id, val) -> System.out.println(val));
```

The above code should be self-explaining: `putIfAbsent` prevents us from writing additional if null checks; `forEach` accepts a consumer to perform operations for each value of the map.

This example shows how to compute code on the map by utilizing functions:
如何计算映射代码

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3);             // val33

map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9);     // false

map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23);    // true

map.computeIfAbsent(3, num -> "bam");
map.get(3);             // val33
```

Next, we learn how to remove entries for a given key, only if it's currently mapped to a given value:
如何为给定键删除映射条目

```
map.remove(3, "val3");
map.get(3);             // val33

map.remove(3, "val33");
map.get(3);             // null
```

Another helpful method:

```
map.getOrDefault(42, "not found");  // not found
```

Merging entries of a map is quite easy: 合并映射条目

```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9);             // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9);             // val9concat
```

Merge either put the key/value into the map if no entry for the key exists, or the merging function will be called to change the existing value.

Date API 日期API

Java 8 contains a brand new date and time API under the package `java.time`. The new Date API is comparable with the Joda-Time library, however it's not the same. The following examples cover the most important parts of this new API.

Clock 时钟：提供当前日期和时间的访问

Clock provides access to the current date and time. Clocks are aware of a timezone and may be used instead of `System.currentTimeMillis()` to retrieve the current time in milliseconds since Unix EPOCH. Such an instantaneous point on the time-line is also represented by the class `Instant`. Instants can be used to create legacy `java.util.Date` objects.

不可变的和线程安全的

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();

Instant instant = clock.instant();
Date legacyDate = Date.from(instant);   // legacy java.util.Date
```

Timezones 时区：由ZoneId表示

Timezones are represented by a `ZoneId`. They can easily be accessed via static factory methods. Timezones define the offsets which are important to convert between instants and local dates and times.

不可变的和线程安全的

```
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]
```

LocalTime 当地时间：表示一个时间

LocalTime represents a time without a timezone, e.g. 10pm or 17:30:15. The following example creates two local times for the timezones defined above. Then we compare both times and calculate the difference in hours and minutes between both times.

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));  // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween);       // -3
System.out.println(minutesBetween);     // -239
```

LocalTime comes with various factory methods to simplify the creation of new instances, including parsing of time strings.

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);       // 23:59:59

DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedTime(FormatStyle.SHORT)
        .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime);   // 13:37
```

LocalDate 当地日期：表示不同的日期

LocalDate represents a distinct date, e.g. 2014-03-11. It's immutable and works exactly analog to LocalTime. The sample demonstrates how to calculate new dates by adding or subtracting days, months or years. Keep in mind that each manipulation returns a new instance.

不可变的和线程安全的

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println(dayOfWeek);    // FRIDAY
```

Parsing a LocalDate from a string is just as simple as parsing a LocalTime:

```
DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedDate(FormatStyle.MEDIUM)
        .withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);
System.out.println(xmas);   // 2014-12-24
```

LocalDateTime 当地日期时间：表示一个日期-时间

LocalDateTime represents a date-time. It combines date and time as seen in the above sections into one instance. `LocalDateTime` is immutable and works similar to LocalTime and LocalDate. We can utilize methods for retrieving certain fields from a date-time:

```
LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek);      // WEDNESDAY

Month month = sylvester.getMonth();
System.out.println(month);          // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay);    // 1439
```

With the additional information of a timezone it can be converted to an instant. Instants can easily be converted to legacy dates of type `java.util.Date` .

```
Instant instant = sylvester
        .atZone(ZoneId.systemDefault())
        .toInstant();

Date legacyDate = Date.from(instant);
System.out.println(legacyDate);     // Wed Dec 31 23:59:59 CET 2014
```

Formatting date-times works just like formatting dates or times. Instead of using pre-defined formats we can create formatters from custom patterns. 格式化日期-时间，自定义模式

```
DateTimeFormatter formatter =
    DateTimeFormatter
        .ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);
String string = formatter.format(parsed);
System.out.println(string);     // Nov 03, 2014 - 07:13
```

Unlike `java.text.NumberFormat` the new `DateTimeFormatter` is immutable and **thread-safe**.

不可变的和线程安全的

For details on the pattern syntax read here.

Annotations 注解

Annotations in Java 8 are repeatable. Let's dive directly into an example to figure that out.

First, we define a wrapper annotation which holds an array of the actual annotations:

```
@interface Hints {
    Hint[] value();
}

@Repeatable(Hints.class)
@interface Hint {
    String value();
}
```

Java 8 enables us to use multiple annotations of the same type by declaring the annotation `@Repeatable` .

Variant 1: Using the container annotation (old school)

```
@Hints({@Hint("hint1"), @Hint("hint2")})
class Person {}
```

Variant 2: Using repeatable annotations (new school)

```
@Hint("hint1")
@Hint("hint2")
class Person {}
```

Using variant 2 the java compiler implicitly sets up the `@Hints` annotation under the hood. That's important for reading annotation information via reflection.

```
Hint hint = Person.class.getAnnotation(Hint.class);
System.out.println(hint);                   // null

Hints hints1 = Person.class.getAnnotation(Hints.class);
System.out.println(hints1.value().length);  // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);
System.out.println(hints2.length);          // 2
```

Although we never declared the `@Hints` annotation on the `Person` class, it's still readable via `getAnnotation(Hints.class)`. However, the more convenient method is `getAnnotationsByType` which grants direct access to all annotated `@Hint` annotations.

Furthermore the usage of annotations in Java 8 is expanded to two new targets:

```
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```

Where to go from here? 下一步

My programming guide to Java 8 ends here. If you want to learn more about all the new classes and features of the JDK 8 API, check out my JDK8 API Explorer. It helps you figuring out all the new classes and hidden gems of JDK 8, like `Arrays.parallelSort`, `StampedLock` and `CompletableFuture` - just to name a few.

I've also published a bunch of follow-up articles on my blog that might be interesting to you:

- Java 8 Stream Tutorial
- Java 8 Nashorn Tutorial
- Java 8 Concurrency Tutorial: Threads and Executors
- Java 8 Concurrency Tutorial: Synchronization and Locks
- Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap
- Java 8 API by Example: Strings, Numbers, Math and Files
- Avoid Null Checks in Java 8
- Fixing Java 8 Stream Gotchas with IntelliJ IDEA
- Using Backbone.js with Java 8 Nashorn

You should follow me on Twitter. Thanks for reading!