Java 8 Guide Chinese Translation

Guide About 6612 words About 22 minutes

With the growing popularity of Java 8, many people have mentioned that Java 8 knowledge is a frequently asked topic in interviews. In response to your requests and needs, I'm planning to summarize this knowledge. I originally planned to do it myself, but then I came across a related repository on GitHub: https://github.com/winterbe/java8-tutorial . This repository is in English, so I translated it and added and modified some content. Below is the full text.

Welcome to my introduction to Java 8. This tutorial will guide you step-by-step through all the new language features. Building on short code examples, you'll learn how to use default interface methods, lambda expressions, method references, and repeatable annotations. By the end of this article, you'll be familiar with the latest API changes, such as streams, functional interfaces, extensions to the Map class, and the new Date API. No long, boring paragraphs, just a bunch of commented code snippets.

Default Methods for Interfaces

Java 8 enables us to `default` add non-abstract method implementations to interfaces by using the keyword. This feature is also known as virtual extension methods .

First example:

```java
interface Formula{

    double calculate(int a);

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

}
```

The Formula interface defines not only abstract methods for calculating interface forr but also default methods `sqrt` . Classes implementing this interface only need to implement the abstract methods `calculate` . The default methods `sqrt` can be used directly. Alternatively, you can create objects directly from the interface and then implement the default methods. Let's demonstrate this approach with code.

```java
public class Main {

  public static void main(String[] args) {
    // 通过匿名内部类方式访问接口
    Formula formula = new Formula() {
        @Override
        public double calculate(int a) {
            return sqrt(a * 100);
        }
    };

    System.out.println(formula.calculate(100));     // 100.0
    System.out.println(formula.sqrt(16));           // 4.0

  }

}
```

formula is implemented as an anonymous object. The code is very easy to understand, and the calculation is implemented in 6 lines of code `sqrt(a * 100)`. In the next section, we will see that there is a better and more convenient way to implement single method objects in Java 8.

Translator's note: Both abstract classes and interfaces can be accessed through anonymous inner classes. Objects cannot be created directly from abstract classes or interfaces. Regarding accessing interfaces through anonymous inner classes, we can understand this as follows: an inner class implements the abstract method in the interface and returns an inner class object, which we then reference through the interface.

Lambda expressions

First, let's look at how strings are sorted in older versions of Java:

```java
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);
    }
});
```

Simply `Collections.sort` pass a List object and a comparator to the static method to sort the items in the specified order. A common approach is to create an anonymous comparator object and pass it to `sort` the method.

In Java 8, you no longer need to use this traditional anonymous object method. Java 8 provides a more concise syntax, lambda expressions:

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

As you can see, the code has become shorter and more readable, but it can actually be written even shorter:

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

For a function body that has only one line of code, you can remove the curly braces {} and the return keyword, but you can also write it even shorter:

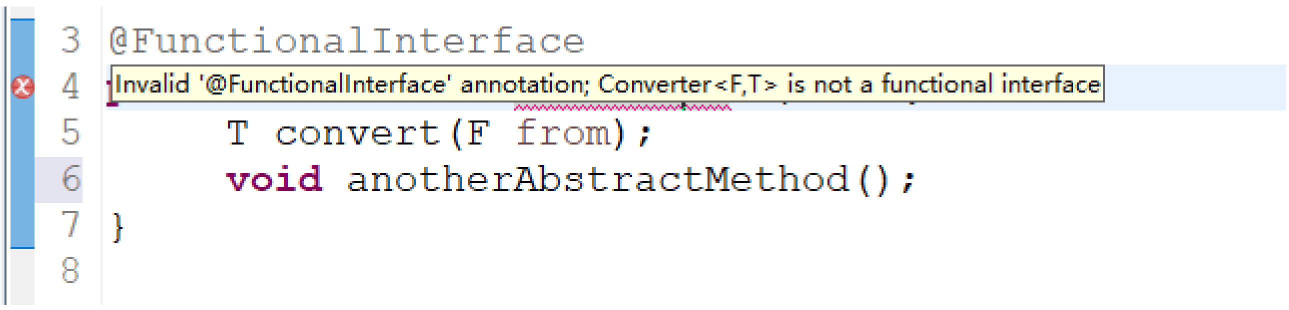
```java
names.sort((a, b) -> b.compareTo(a));
```

The List class itself has a `sort` method. The Java compiler can automatically infer the parameter types, so you don't have to write the types again. Next, let's look at other uses for lambda expressions.

Functional Interfaces

Translator's note: The original text was not very clear about this part, so I made some changes!

The Java language designers have invested a lot of energy in thinking about how to make existing functions friendly to Lambda. The final approach is to add the concept of functional interface. **"Functional interface" refers to an interface that contai only one abstract method, but can have multiple non-abstract methods (that is, the default method mentioned above).** Interfaces like this can be implicitly converted to lambda expressions. `java.lang.Runnable` and `java.util.concurrent.Callable` are the two most typical examples of functional interfaces. Java 8 adds a special annotation `@FunctionalInterface`, but this annotation is usually not required (it is recommended in some cases). As long as the interface contains only one abstract method, the virtual machine will automatically determine that the interface is a functional interface. It is generally recommended to use `@FunctionalInterface` annotations on the interface for declaration. In this way, if the compiler finds that the interface marked with this annotation has more than one abstract method, it will report an error, as shown in the figure below.



```
@FunctionalInterface
Invalid '@FunctionalInterface' annotation; Converter<F,T> is not a functional interface
     T convert(F from);
     void anotherAbstractMethod();
}
```

Example:

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

```java
    // TODO 将数字字符串转换为整数类型
    Converter<String, Integer> converter = (from) ->
Integer.valueOf(from);
    Integer converted = converter.convert("123");
    System.out.println(converted.getClass()); //class
java.lang.Integer
```

Translator's note: We don't need to write most functional interfaces ourselves, Java8 has implemented them for us. These interfaces are in the java.util.function package.

Method and Constructor References

The code in the previous section can also be expressed using a static method referenc

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

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

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

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

Next, let's look at how constructors are `::` referenced using keywords. First, we define a simple class that contains multiple constructors:

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

    Person() {}

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

Next we specify an object factory interface for creating Person objects:

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

Here we use constructor references to connect them together, rather than manually implementing a complete factory:

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

We only need to use `Person::new` to obtain a reference to the Person class constructor, and the Java compiler will automatically `PersonFactory.create` select the appropriate constructor based on the method parameter types.

Lambda Scopes

Accessing local variables

We can access external local variables directly in lambda expressions:

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

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

However, unlike anonymous objects, the variable num does not need to be declared final, and the code is still correct:

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

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

However, num here must not be modified by subsequent code (that is, it implicitly has final semantics). For example, the following will not compile:

```java
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);
num = 3;//在lambda表达式中试图修改num同样是不允许的。
```

Accessing fields and static variables

Compared to local variables, we have read and write access to instance fields and static variables in lambda expressions. This behavior is consistent with anonymous objects.

```java
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

Remember the formula example from the first section? `Formula` The interface defines a default method `sqrt` that can be accessed from every instance of formula containing the anonymous object. This does not apply to lambda expressions.

Default methods cannot be accessed from lambda expressions, so the following code will not compile:

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

Built-in Functional Interfaces

The JDK 1.8 API includes many built-in functional interfaces. Some of these interfaces are common in older versions of Java, such as `Comparator` or `Runnable` , and these interfaces have been `@FunctionalInterface` annotated so that they can be used in lambda expressions.

But the Java 8 API also provides many new functional interfaces to make your programming work more convenient. Some of these interfaces come from the Google Guava library. Even if you are familiar with them, it is still necessary to see how they are extended to use on lambda.

Predicate

The Predicate interface is a predicate interface that takes a single parameter and returns a Boolean value . This interface contains several default methods for combining Predicates into more complex logic (e.g., AND, OR, NOT):

Translator's note: The source code of the Predicate interface is as follows

```java
package java.util.function;
import java.util.Objects;

@FunctionalInterface
public interface Predicate<T> {

    // 该方法是接受一个传入类型,返回一个布尔值.此方法应用于判断.
    boolean test(T t);

    //and方法与关系型运算符"&&"相似，两边都成立才返回true
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }
    // 与关系运算符"!"相似，对判断进行取反
    default Predicate<T> negate() {
        return (t) -> !test(t);
    }
    //or方法与关系型运算符"||"相似，两边只要有一个成立就返回true
    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }
   // 该方法接收一个Object对象,返回一个Predicate类型.此方法用于判断第一个
test的方法与第二个test方法相同(equal).
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
                ? Objects::isNull
                : object -> targetRef.equals(object);
    }
```

Example:

```java
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();
```

Function

The Function interface accepts a single argument and produces a result. Default methods can be used to chain multiple functions together (compose, andThen):

Translator's note: The Function interface source code is as follows

```java

package java.util.function;

import java.util.Objects;

@FunctionalInterface
public interface Function<T, R> {

    //将Function对象应用到输入的参数上，然后返回计算结果。
    R apply(T t);
    //将两个Function整合，并返回一个能够执行两个Function对象功能的Function对象。
    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }
    //
    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

```java
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);
backToString.apply("123");     // "123"
```

Supplier

The Supplier interface produces a result of a given generic type. Unlike the Function interface, the Supplier interface does not accept parameters.

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

Consumer

The Consumer interface represents an operation to be performed on a single input argument.

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

Comparator

Comparator is a classic interface in old Java. Java 8 adds a variety of default methods on top of it:

```java
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
```

Optional

Optional is not a functional interface, but a beautiful tool for preventing NullPointerException. This is an important concept in the next section, so let's take a quick look at how Optional works.

Optional is a simple container whose value may be null or not null. Before Java 8, a function should generally return a non-null object but sometimes it returns nothing. In Java 8, you should return Optional instead of null.

Translator's note: The function of each method in the example has been added.

```java
//of()：为非null的值创建一个Optional
Optional<String> optional = Optional.of("bam");
// isPresent()：如果值存在返回true，否则返回false
optional.isPresent();           // true
//get()：如果Optional有值则将其返回，否则抛出NoSuchElementException
optional.get();                 // "bam"
```

```java
//orElse()：如果有值则将其返回，否则返回指定的其它值
optional.orElse("fallback");    // "bam"
//ifPresent()：如果Optional实例有值则为其调用consumer，否则不做处理
optional.ifPresent((s) -> System.out.println(s.charAt(0)));     //
"b"
```

Recommended reading: [Java8] How to use Optional correctly

Streams

`java.util.Stream` Represents a sequence of operations that can be applied to a set of elements at once. Stream operations are categorized as either intermediate or terminal. Terminal operations return a computational result of a specific type, while intermediate operations return the Stream itself, allowing you to chain multiple operations together. Creating a Stream requires specifying a data source, such as `java.util.Collection` a subclass of List or Set; Map does not support this. Stream operations can be executed serially or in parallel.

First, let's see how to use Stream. First, create the data List needed for the example code:

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

Java 8 extends the collection class, allowing you to create a Stream using Collection.stream() or Collection.parallelStream(). The following sections explain common Stream operations in detail:

Filter

Filtering uses a predicate interface to filter and retain only elements that meet the predicate. This operation is **an intermediate operation** , allowing you to apply othe stream operations (such as forEach) to the filtered results. forEach requires a function to be executed sequentially on the filtered elements. forEach is a terminal operation, so you cannot perform other stream operations after forEach.

```java
        // 测试 Filter(过滤)
        stringList
                .stream()
                .filter((s) -> s.startsWith("a"))
                .forEach(System.out::println);//aaa2 aaa1
```

forEach is designed for Lambda, maintaining the most compact style. Lambda expressions themselves are reusable, making them very convenient.

Sorted

Sorting is an **intermediate operation** that returns a sorted Stream. **If you don't specify a custom Comparator, the default sorting will be used.**

```java
        // 测试 Sort (排序)
        stringList
                .stream()
                .sorted()
                .filter((s) -> s.startsWith("a"))
                .forEach(System.out::println);// aaa1 aaa2
```

It should be noted that sorting only creates a sorted Stream and does not affect the original data source. The original stringList data will not be modified after sorting:

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

Map

The intermediate operation map will convert the elements into other objects in turn according to the specified Function interface.

The following example converts a string to uppercase. You can also use map to convert objects to other types. The type of the Stream returned by map is determined by the r value of the function you pass to map.

```java
        // 测试 Map 操作
        stringList
                .stream()
                .map(String::toUpperCase)
                .sorted((a, b) -> b.compareTo(a))
                .forEach(System.out::println);// "DDD2", "DDD1",
"CCC", "BBB3", "BBB2", "BBB1", "AAA2", "AAA1"
```

Match

Stream provides a variety of matching operations, allowing you to check whether the specified Predicate matches the entire stream. All matching operations are **terminal operations** and return a boolean value.

```java
        // 测试 Match (匹配)操作
        boolean anyStartsWithA =
                stringList
                        .stream()
                        .anyMatch((s) -> s.startsWith("a"));
        System.out.println(anyStartsWithA);      // true

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

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

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

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

Count

Count is a **terminal operation** that returns the number of elements in the Stream. **return value type is long** .

```java
        //测试 Count (计数)操作
          long startsWithB =
                  stringList
                          .stream()
                          .filter((s) -> s.startsWith("b"))
                          .count();
          System.out.println(startsWithB);    // 3
```

Reduce

This is a **final operation** that allows multiple elements in the stream to be reduced to one element through a specified function. The result of the reduction is represented by the Optional interface:

```java
          //测试 Reduce (规约)操作
          Optional<String> reduced =
                  stringList
                          .stream()
                          .sorted()
                          .reduce((s1, s2) -> s1 + "#" + s2);


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

Translator's Note: This method primarily combines elements of a Stream. It provides a starting value (seed) and then, following an operation (BinaryOperator), combines it with the first, second, and nth elements of the preceding Stream. In this sense, string concatenation and the sum, min, max, and average of numbers are special cases of reduce. For example, summing a Stream is equivalent to `Integer sum = integers.reduce(0, (a, b) -> a+b);` the case where there is no starting value; in this case, the first two elements of the Stream are combined, returning an Optional.

```java
// 字符串连接, concat = "ABCD"
String concat = Stream.of("A", "B", "C", "D").reduce("",
String::concat);
// 求最小值, minValue = -3.0
double minValue = Stream.of(-1.5, 1.0, -3.0,
-2.0).reduce(Double.MAX_VALUE, Double::min);
// 求和, sumValue = 10, 有起始值
int sumValue = Stream.of(1, 2, 3, 4).reduce(0, Integer::sum);
```

```java
// 求和，sumValue = 10, 无起始值
sumValue = Stream.of(1, 2, 3, 4).reduce(Integer::sum).get();
// 过滤，字符串连接，concat = "ace"
concat = Stream.of("a", "B", "c", "D", "e", "F").
 filter(x -> x.compareTo("Z") > 0).
 reduce("", String::concat);
```

In the code above, for example, in the first reduce() example, the first parameter (a whitespace character) is the starting value, and the second parameter (String::concat) is a BinaryOperator. These reduce() methods with starting values all return specific objects. In contrast, in the fourth reduce() method, which doesn't have a starting value, an Optional is returned because there may not be enough elements. Please note this difference. For more information, see: IBM: Detailed Explanation of the Streams API in Java 8

Parallel Streams

As mentioned earlier, there are two types of streams: serial and parallel. Operations on a serial stream are completed sequentially in a single thread, while operations on a parallel stream are executed simultaneously on multiple threads.

The following example shows how to improve performance by parallelizing Streams:

First we create a large table with no duplicate elements:

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

We sort them in serial and parallel ways respectively, and finally compare the time taken.

Sequential Sort

```java
//串行排序
long t0 = System.nanoTime();
long count = values.stream().sorted().count();
System.out.println(count);

```

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

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

```plain
1000000
sequential sort took: 709 ms//串行排序所用的时间
```

Parallel Sort

```java
//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));
```

```java
1000000
parallel sort took: 475 ms//并行排序所用的时间
```

The two codes above are almost identical, but the parallel version is about 50% faster. The only change required is to `stream()` change `parallelStream()` .

Maps

As mentioned earlier, the Map type does not support streams, but Map provides some new useful methods to handle some common tasks. The Map interface itself has no available `stream()` methods, but you can create specialized streams on keys, values, or through `map.keySet().stream()` , `map.values().stream()` and `map.entrySet().stream()` .

Additionally, Maps supports a variety of new and useful methods to perform commo tasks.

```java
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));//val0 val1 val2 val3 val4 val5 val6 val7 val8 val9
```

`putIfAbsent` Prevents us from writing extra code for null checks; `forEach` accepts a consumer to operate on each element in the map.

This example shows how to use a function to evaluate code on a map:

```java
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 show how to delete an item in a Map whose key and value all match:

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

Another useful method:

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

Merging Map elements is also easy:

```java
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
```

What Merge does is to insert the key if it does not exist, otherwise merge the value corresponding to the original key and reinsert it into the map.

Date API (date-related API)

Java 8 `java.time` includes a completely new date and time API in the Java 8 package. The new Date API is similar to the Joda-Time library, but it's not the same. The following examples cover the most important parts of this new API. The translator made most of the revisions to this section based on references to the relevant books.

Translator's Note (Summary):

- The Clock class provides methods for accessing the current date and time. Clock is time-zone-sensitive and can be used instead of `System.currentTimeMillis()` to retrieve the current number of microseconds. A specific point in time can also be `Instant` represented using the class, `Instant` which can also be used to create the older version of `java.util.Date` the object.
- In the new API, time zones are represented by ZoneId. Time zones can be easily retrieved using the static method of . The abstract class `ZoneId` (in `java.time` the package) represents a zone identifier. It has a `getAvailableZoneIds` static method called that returns all zone identifiers.
- JDK 1.8 adds classes such as LocalDate and LocalDateTime to handle date processing, and also introduces a new class, DateTimeFormatter, to handle date formatting. You can use Instant instead of Date, LocalDateTime instead of Calendar, and DateTimeFormatter instead of SimpleDateFormat.

Clock

The Clock class provides methods for accessing the current date and time. Clock is tir zone-sensitive and can be used instead of `System.currentTimeMillis()` to retrieve current number of microseconds. A specific point in time can also be `Instant` represented using the class, `Instant` which can also be used to create the older version of `java.util.Date` the object.

```java
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();
System.out.println(millis);//1552379579043
Instant instant = clock.instant();
System.out.println(instant);
Date legacyDate = Date.from(instant); //2019-03-12T08:46:42.588Z
System.out.println(legacyDate);//Tue Mar 12 16:32:59 CST 2019
```

Timezones

In the new API, time zones are represented by ZoneId. Time zones can be easily retrieved using the static method of . The abstract class `ZoneId` (in `java.time` the package) represents a zone identifier. It has a `getAvailableZoneIds` static method called that returns all zone identifiers.

```java
//输出所有区域标识符
System.out.println(ZoneId.getAvailableZoneIds());

ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());// ZoneRules[currentStandardOffset=+01:00]
System.out.println(zone2.getRules());// ZoneRules[currentStandardOffset=-03:00]
```

LocalTime

LocalTime defines a time without a time zone, such as 10:00 PM or 5:30:15 PM. The following example creates two local times using the time zone created in the previous code. It then compares the times and calculates the difference in hours and minutes:

```java
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 provides various factory methods to simplify the creation of objects, including parsing time strings.

```java
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

A LocalDate represents an exact date, such as 2014-03-11. This object is immutable and works similarly to a LocalTime object. The following example shows how to add or subtract days, months, and years to a Date object. Also note that these objects are immutable; operations always return a new instance.

```java
LocalDate today = LocalDate.now();//获取现在的日期
System.out.println("今天的日期: "+today);//2019-03-12
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
System.out.println("明天的日期: "+tomorrow);//2019-03-13
LocalDate yesterday = tomorrow.minusDays(2);
System.out.println("昨天的日期: "+yesterday);//2019-03-11
LocalDate independenceDay = LocalDate.of(2019, Month.MARCH, 12);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println("今天是周几:"+dayOfWeek);//TUESDAY
```

Parsing a LocalDate type from a string is as simple as parsing a LocalTime. Here is `DateTimeFormatter` an example of using parseString:

```java
    String str1 = "2014==04==12 01时06分09秒";
        // 根据需要解析的日期、时间字符串定义解析所用的格式器
        DateTimeFormatter fomatter1 = DateTimeFormatter
                .ofPattern("yyyy==MM==dd HH时mm分ss秒");

        LocalDateTime dt1 = LocalDateTime.parse(str1, fomatter1);
        System.out.println(dt1); // 输出 2014-04-12T01:06:09

        String str2 = "2014$$$四月$$$13 20小时";
        DateTimeFormatter fomatter2 = DateTimeFormatter
                .ofPattern("yyy$$$MMM$$$dd HH小时");
        LocalDateTime dt2 = LocalDateTime.parse(str2, fomatter2);
        System.out.println(dt2); // 输出 2014-04-13T20:00
```

Let's look at an example of using `DateTimeFormatter` formatted dates.

```java
LocalDateTime rightNow=LocalDateTime.now();
String date=DateTimeFormatter.ISO_LOCAL_DATE_TIME.format(rightNow);
System.out.println(date);//2019-03-12T16:26:48.29
DateTimeFormatter formatter=DateTimeFormatter.ofPattern("YYYY-MM-dd HH:mm:ss");
System.out.println(formatter.format(rightNow));//2019-03-12 16:26:48
```

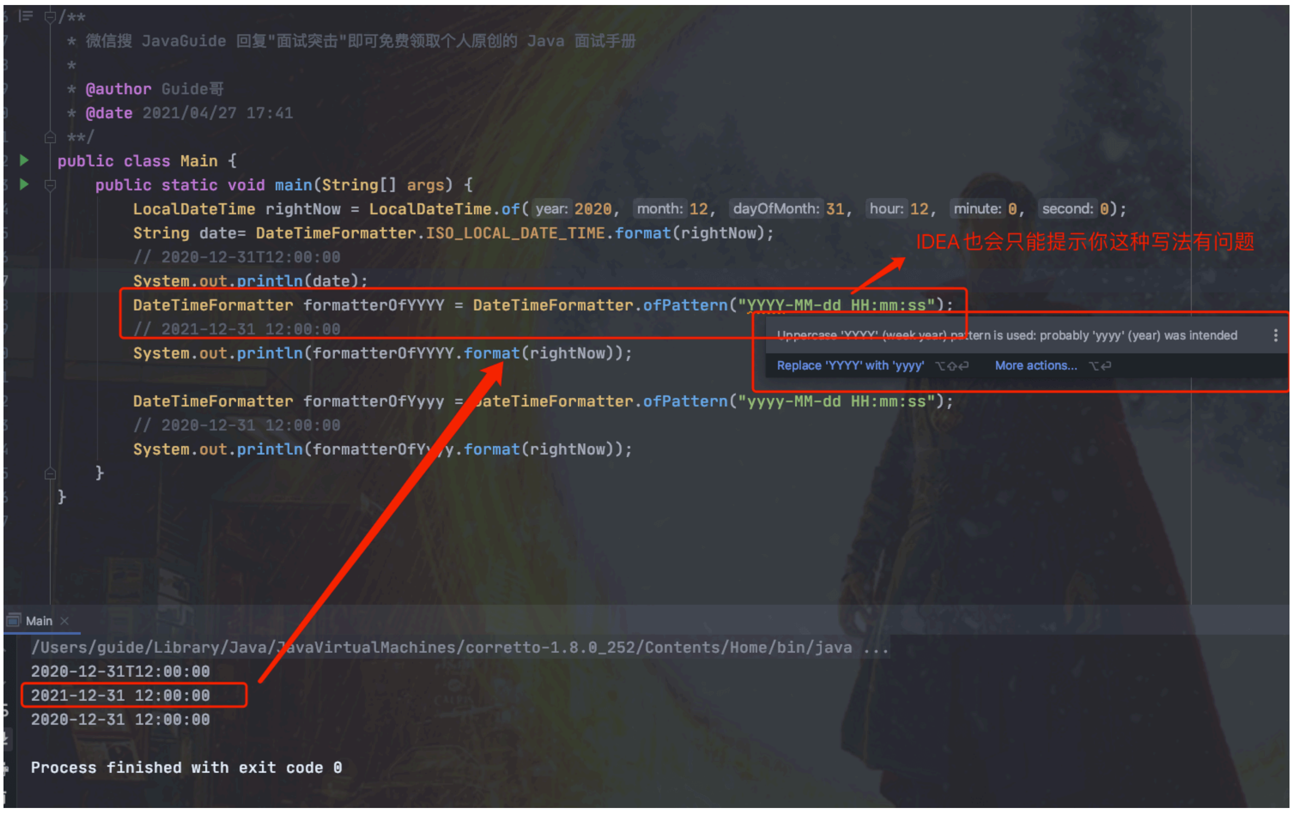
🐛 **Fixed (see: issue#1157):** `YYYY` When using to display the year, the year of the current week will be displayed, which will cause problems during the year-end week. In general `yyyy` , use to display the correct year.

Example of incorrect date display due to New Year's Eve:

```java
LocalDateTime rightNow = LocalDateTime.of(2020, 12, 31, 12, 0, 0);
String date= DateTimeFormatter.ISO_LOCAL_DATE_TIME.format(rightNow);
// 2020-12-31T12:00:00
System.out.println(date);
DateTimeFormatter formatterOfYYYY = DateTimeFormatter.ofPattern("YYYY-MM-dd HH:mm:ss");
// 2021-12-31 12:00:00
System.out.println(formatterOfYYYY.format(rightNow));
```

```java
DateTimeFormatter formatterOfYyyy =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
// 2020-12-31 12:00:00
System.out.println(formatterOfYyyy.format(rightNow));
```

The specific error can be seen more clearly in the figure below, and IDEA has intelligently suggested that it is more appropriate to use `yyyy` instead of `YYYY` .



```java
/**
 * 微信搜 JavaGuide 回复"面试突击"即可免费领取个人原创的 Java 面试手册
 *
 * @author Guide哥
 * @date 2021/04/27 17:41
 **/
public class Main {
    public static void main(String[] args) {
        LocalDateTime rightNow = LocalDateTime.of( year: 2020, month: 12, dayOfMonth: 31, hour: 12, minute: 0, second: 0);
        String date= DateTimeFormatter.ISO_LOCAL_DATE_TIME.format(rightNow);
        // 2020-12-31T12:00:00
        System.out.println(date);
        DateTimeFormatter formatterOfYYYY = DateTimeFormatter.ofPattern("YYYY-MM-dd HH:mm:ss");
        // 2021-12-31 12:00:00
        System.out.println(formatterOfYYYY.format(rightNow));

        DateTimeFormatter formatterOfYyyy = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        // 2020-12-31 12:00:00
        System.out.println(formatterOfYyyy.format(rightNow));
    }
}
```

IDEA 也会只能提示你这种写法有问题

Uppercase 'YYYY' (week year) pattern is used: probably 'yyyy' (year) was intended

```
/Users/guide/Library/Java/JavaVirtualMachines/corretto-1.8.0_252/Contents/Home/bin/java ...
2020-12-31T12:00:00
2021-12-31 12:00:00
2020-12-31 12:00:00

Process finished with exit code 0
```

LocalDateTime (local date and time)

LocalDateTime represents both time and date, effectively combining the concepts from the previous two sections into a single object. Like LocalTime and LocalDate, LocalDateTime is immutable. LocalDateTime provides methods for accessing specific fields.

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

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

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

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

As long as the time zone information is attached, it can be converted into an Instant object, and the Instant object can be easily converted into the old one `java.util.Date` .

```java
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 LocalDateTime is the same as formatting time and date. In addition to using predefined formats, we can also define our own formats:

```java
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, so it is thread-safe.
Detailed information about date and time formats is available here .

Annotations

Java 8 supports multiple annotations. Let's take a look at an example to understand what it means.
First, define a wrapper class Hints annotation to place a group of specific Hint annotations:

```java
@Retention(RetentionPolicy.RUNTIME)
@interface Hints {
    Hint[] value();
}
@Repeatable(Hints.class)
@interface Hint {
    String value();
}
```

Java 8 allows us to use the same type of annotation multiple times by simply marking the annotation `@Repeatable` .

Example 1: Using a wrapper class as a container to store multiple annotations (old method)

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

Example 2: Using multiple annotations (new method)

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

In the second example, the Java compiler will implicitly define the @Hints annotation for you. Knowing this will help you get this information using reflection:

```java
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
```

Even if we don't define annotations on `Person` a class , we can still get annotations through . A more convenient way is to use to get all annotations directly. In addition, Java 8 annotations have been added to two new targets: `@Hints` `getAnnotation(Hints.class)` `@Hints` `getAnnotationsByType` `@`
`t`

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

Where to go from here?

That's all I have to say about the new features in Java 8. There are certainly more features waiting to be discovered. JDK 1.8 also has many useful features, such as `Arrays.parallelSort` , `StampedLock` and `CompletableFuture` .

JavaGuide官方公众号
（微信搜索JavaGuide）
1、公众号后台回复“PDF”获取原创PDF面试手册
2、公众号后台回复“学习路线”获取Java学习路线最新版
3、公众号后台回复“开源”获取优质Java开源项目合集
4、公众号后台回复“八股文”获取Java面试真题+面经

Recently Updated2024/7/11 16:55
Contributors: Snailclimb , SnailClimb , rawlinxx , Liu Guangming , Jin Yang , Ryze-Zhao , shihui , shuang.kou , qinkai , ygf , guide , ZhiHong Li , Mike Liu , anaer , Guide , Mr.Hope , paigeman , SteamBoatv