

Java SE 8 New Features

Study Notes



December 1, 2015

Vladimir Virlan

Contents

[Introduction 2](#_Toc437009540)

[Java Programming Language 2](#_Toc437009541)

[Lambda Expressions 2](#_Toc437009542)

[Functional Interfaces 2](#_Toc437009543)

[Predicate<T > predicate 5](#_Toc437009544)

[Consumer<T> block 6](#_Toc437009545)

[Function<T,R> 6](#_Toc437009546)

[Supplier<T> 7](#_Toc437009547)

[Custom Functional Interfaces 8](#_Toc437009548)

[Method references 8](#_Toc437009549)

[Reference to a Static Method 9](#_Toc437009550)

[Reference to an Instance Method of a Particular Object 9](#_Toc437009551)

[Reference to an Instance Method of an Arbitrary Object of a Particular Type 9](#_Toc437009552)

[Reference to a Constructor 9](#_Toc437009553)

[Default methods and statics in interfaces 10](#_Toc437009554)

[Repeating Annotations 12](#_Toc437009555)

[Type Annotations 13](#_Toc437009556)

[Method parameter reflection 13](#_Toc437009557)

[java.lang and java.util Packages 13](#_Toc437009558)

[Standard Encoding and Decoding Base64 13](#_Toc437009559)

[Unsigned Arithmetic Support 18](#_Toc437009560)

[Stream API 19](#_Toc437009561)

[Aggregate Operations 19](#_Toc437009562)

[Reduction 19](#_Toc437009563)

[Parallelism 19](#_Toc437009564)

[Date Time API 19](#_Toc437009565)

[LocalDate and LocalTime 21](#_Toc437009566)

[Parsing and Formatting 25](#_Toc437009567)

[Temporal Adjuster 26](#_Toc437009568)

[Optional 28](#_Toc437009569)

[Concurrency 30](#_Toc437009570)

[Tools 31](#_Toc437009571)

[jdeps 31](#_Toc437009572)

[Nashorn, JavaScript Engine 32](#_Toc437009573)

# Introduction

This document contains my study notes for the new features in Java SE 8.

# Java Programming Language

## Lambda Expressions

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

Here is an example of some lambda expressions:

ArithmeticOperation addition = (**int** a, **int** b) -> a + b;

ArithmeticOperation subtraction = (**int** a, **int** b) -> a - b;

ArithmeticOperation multiplication = (**int** a, **int** b) -> a \* b;

ArithmeticOperation division = (**int** a, **int** b) -> a / b;

System.***out***.println("10 + 10 = "+operate(10, 10, addition));

System.***out***.println("10 - 10 = "+operate(10, 10, subtraction));

System.***out***.println("10 \* 10 = "+operate(10, 10, multiplication));

System.***out***.println("10 / 10 = "+operate(10, 10, division));

This is the operate method:

**private** **int** operate(**int** a, **int** b, ArithmeticOperation operation) {

**return** operation.operation(a, b);

}

And here is the ArithmeticOperation:

**public** **interface** ArithmeticOperation {

**int** operation(**int** a, **int** b);

}

Following are the important characteristics of a lambda expression −

* **Optional type declaration** − No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
* **Optional parenthesis around parameter** − No need to declare a single parameter in parenthesis. For multiple parameters, parentheses are required.
* **Optional curly braces** − No need to use curly braces in expression body if the body contains a single statement.
* **Optional return keyword** − The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

### Functional Interfaces

#### Package java.util.function

*Functional interfaces* provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called the *functional method* for that functional interface, to which the lambda expression's parameter and return types are matched or adapted. Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, or cast context:

// Assignment context

Predicate<String> p = String::isEmpty;

// Method invocation context

stream.filter(e -> e.getSize() > 10)...

// Cast context

stream.map((ToIntFunction) e -> e.getSize())...

The interfaces in this package are general purpose functional interfaces used by the JDK, and are available to be used by user code as well. While they do not identify a complete set of function shapes to which lambda expressions might be adapted, they provide enough to cover common requirements. Other functional interfaces provided for specific purposes, such as [FileFilter](https://docs.oracle.com/javase/8/docs/api/java/io/FileFilter.html), are defined in the packages where they are used.

The interfaces in this package are annotated with [FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html). This annotation is not a requirement for the compiler to recognize an interface as a functional interface, but merely an aid to capture design intent and enlist the help of the compiler in identifying accidental violations of design intent.

Functional interfaces often represent abstract concepts like functions, actions, or predicates. In documenting functional interfaces, or referring to variables typed as functional interfaces, it is common to refer directly to those abstract concepts, for example using "this function" instead of "the function represented by this object". When an API method is said to accept or return a functional interface in this manner, such as "applies the provided function to...", this is understood to mean a *non-null* reference to an object implementing the appropriate functional interface, unless potential nullity is explicitly specified.

The functional interfaces in this package follow an extensible naming convention, as follows:

* There are several basic function shapes, including [Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) (unary function from T to R), [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) (unary function from T to void), [Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) (unary function from T to boolean), and [Supplier](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) (nilary function to R).
* Function shapes have a natural arity based on how they are most commonly used. The basic shapes can be modified by an arity prefix to indicate a different arity, such as [BiFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html) (binary function from T and U to R).
* There are additional derived function shapes which extend the basic function shapes, including [UnaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html) (extends Function) and [BinaryOperator](https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html) (extends BiFunction).
* Type parameters of functional interfaces can be specialized to primitives with additional type prefixes. To specialize the return type for a type that has both generic return type and generic arguments, we prefix ToXxx, as in [ToIntFunction](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntFunction.html). Otherwise, type arguments are specialized left-to-right, as in [DoubleConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleConsumer.html) or [ObjIntConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html). (The type prefix Obj is used to indicate that we don't want to specialize this parameter, but want to move on to the next parameter, as in [ObjIntConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html).) These schemes can be combined, as in IntToDoubleFunction.
* If there are specialization prefixes for all arguments, the arity prefix may be left out (as in [ObjIntConsumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html)).

|  |  |
| --- | --- |
| **Interface Summary** | |
| **Interface** | **Description** |
| [**BiConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiConsumer.html)<T,U> | Represents an operation that accepts two input arguments and returns no result. |
| [**BiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html)<T,U,R> | Represents a function that accepts two arguments and produces a result. |
| [**BinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html)<T> | Represents an operation upon two operands of the same type, producing a result of the same type as the operands. |
| [**BiPredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BiPredicate.html)<T,U> | Represents a predicate (boolean-valued function) of two arguments. |
| [**BooleanSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BooleanSupplier.html) | Represents a supplier of boolean-valued results. |
| [**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<T> | Represents an operation that accepts a single input argument and returns no result. |
| [**DoubleBinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleBinaryOperator.html) | Represents an operation upon two double-valued operands and producing a double-valued result. |
| [**DoubleConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleConsumer.html) | Represents an operation that accepts a single double-valued argument and returns no result. |
| [**DoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleFunction.html)<R> | Represents a function that accepts a double-valued argument and produces a result. |
| [**DoublePredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoublePredicate.html) | Represents a predicate (boolean-valued function) of one double-valued argument. |
| [**DoubleSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleSupplier.html) | Represents a supplier of double-valued results. |
| [**DoubleToIntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleToIntFunction.html) | Represents a function that accepts a double-valued argument and produces an int-valued result. |
| [**DoubleToLongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleToLongFunction.html) | Represents a function that accepts a double-valued argument and produces a long-valued result. |
| [**DoubleUnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/DoubleUnaryOperator.html) | Represents an operation on a single double-valued operand that produces a double-valued result. |
| [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<T,R> | Represents a function that accepts one argument and produces a result. |
| [**IntBinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntBinaryOperator.html) | Represents an operation upon two int-valued operands and producing an int-valued result. |
| [**IntConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntConsumer.html) | Represents an operation that accepts a single int-valued argument and returns no result. |
| [**IntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntFunction.html)<R> | Represents a function that accepts an int-valued argument and produces a result. |
| [**IntPredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntPredicate.html) | Represents a predicate (boolean-valued function) of one int-valued argument. |
| [**IntSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntSupplier.html) | Represents a supplier of int-valued results. |
| [**IntToDoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntToDoubleFunction.html) | Represents a function that accepts an int-valued argument and produces a double-valued result. |
| [**IntToLongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntToLongFunction.html) | Represents a function that accepts an int-valued argument and produces a long-valued result. |
| [**IntUnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/IntUnaryOperator.html) | Represents an operation on a single int-valued operand that produces an int-valued result. |
| [**LongBinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongBinaryOperator.html) | Represents an operation upon two long-valued operands and producing a long-valued result. |
| [**LongConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongConsumer.html) | Represents an operation that accepts a single long-valued argument and returns no result. |
| [**LongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongFunction.html)<R> | Represents a function that accepts a long-valued argument and produces a result. |
| [**LongPredicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongPredicate.html) | Represents a predicate (boolean-valued function) of one long-valued argument. |
| [**LongSupplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongSupplier.html) | Represents a supplier of long-valued results. |
| [**LongToDoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongToDoubleFunction.html) | Represents a function that accepts a long-valued argument and produces a double-valued result. |
| [**LongToIntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongToIntFunction.html) | Represents a function that accepts a long-valued argument and produces an int-valued result. |
| [**LongUnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/LongUnaryOperator.html) | Represents an operation on a single long-valued operand that produces a long-valued result. |
| [**ObjDoubleConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjDoubleConsumer.html)<T> | Represents an operation that accepts an object-valued and a double-valued argument, and returns no result. |
| [**ObjIntConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjIntConsumer.html)<T> | Represents an operation that accepts an object-valued and a int-valued argument, and returns no result. |
| [**ObjLongConsumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ObjLongConsumer.html)<T> | Represents an operation that accepts an object-valued and a long-valued argument, and returns no result. |
| [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<T> | Represents a predicate (boolean-valued function) of one argument. |
| [**Supplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html)<T> | Represents a supplier of results. |
| [**ToDoubleBiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToDoubleBiFunction.html)<T,U> | Represents a function that accepts two arguments and produces a double-valued result. |
| [**ToDoubleFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToDoubleFunction.html)<T> | Represents a function that produces a double-valued result. |
| [**ToIntBiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntBiFunction.html)<T,U> | Represents a function that accepts two arguments and produces an int-valued result. |
| [**ToIntFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToIntFunction.html)<T> | Represents a function that produces an int-valued result. |
| [**ToLongBiFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToLongBiFunction.html)<T,U> | Represents a function that accepts two arguments and produces a long-valued result. |
| [**ToLongFunction**](https://docs.oracle.com/javase/8/docs/api/java/util/function/ToLongFunction.html)<T> | Represents a function that produces a long-valued result. |
| [**UnaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/UnaryOperator.html)<T> | Represents an operation on a single operand that produces a result of the same type as its operand. |

### Predicate<T > predicate

Represents a predicate (boolean-valued function) of one argument.This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [test(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#test-T-).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| default [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> | [**and**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#and-java.util.function.Predicate-)([**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> other)  Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. |
| static <T> [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<T> | [**isEqual**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#isEqual-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) targetRef)  Returns a predicate that tests if two arguments are equal according to [**Objects.equals(Object, Object)**](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#equals-java.lang.Object-java.lang.Object-). |
| default [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> | [**negate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#negate--)()  Returns a predicate that represents the logical negation of this predicate. |
| default [**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> | [**or**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#or-java.util.function.Predicate-)([**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)> other)  Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. |
| boolean | [**test**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#test-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) t)  Evaluates this predicate on the given argument. |

Example:

Predicate<String> p = String::isEmpty;

System.***out***.println(p.test("aha"));

Will print: false

### Consumer<T> block

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects. This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [accept(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| void | [**accept**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t)  Performs this operation on the given argument. |
| default [**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> | [**andThen**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#andThen-java.util.function.Consumer-)([**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)> after)  Returns a composed Consumer that performs, in sequence, this operation followed by the after operation. |

Example:

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

consumer.accept("Hello World!");

Will print: HELLO WORLD!

### Function<T,R>

Represents a function that accepts one argument and produces a result. This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [apply(Object)](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#apply-T-).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| default <V> [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html),V> | [**andThen**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#andThen-java.util.function.Function-)([**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html),? extends V> after)  Returns a composed function that first applies this function to its input, and then applies the after function to the result. |
| [**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) | [**apply**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#apply-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) t)  Applies this function to the given argument. |
| default <V> [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<V,[**R**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)> | [**compose**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#compose-java.util.function.Function-)([**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super V,? extends [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)> before)  Returns a composed function that first applies the before function to its input, and then applies this function to the result. |
| static <T> [**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<T,T> | [**identity**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#identity--)()  Returns a function that always returns its input argument. |

Example:

Function<String,Integer> function = (f) -> f.length();

System.***out***.println(function.apply("Hello There!"));

Will print: 12

### Supplier<T>

Represents a supplier of results. There is no requirement that a new or distinct result be returned each time the supplier is invoked. This is a [functional interface](https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html) whose functional method is [get()](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html#get--).

|  |  |
| --- | --- |
| **Modifier and Type** | **Method and Description** |
| [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) | [**get**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html#get--)()  Gets a result. |

Example 1:

Random r = **new** Random();

Supplier<Integer> intSupplier = () -> r.nextInt();

System.***out***.println(intSupplier.get());

Result: 1643357758 or -17532822 or any other random int

Example 2:

**private** **void** supplier() {

Random r = **new** Random();

Supplier<Integer> intSupplier = () -> r.nextInt(25);

**int** len = 10;

Supplier<String> randomStrSupplier = () -> {

String alpha = "abcdefghijklmnopqrstuvwxyz";

StringBuilder sb = **new** StringBuilder();

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

sb.append(alpha.toCharArray()[intSupplier.get()]);

}

**return** sb.toString();

};

System.***out***.println(randomStrSupplier.get());

}

Result: nljwotquta or uhypnprjxv or any other 10 sized string

### Custom Functional Interfaces

It is possible to create your own functional interfaces. You just have to annotate them with:

Annotation Type FunctionalInterface

@Documented

@Retention(value=RUNTIME)

@Target(value=TYPE)

**public** **@interface** FunctionalInterface

An informative annotation type used to indicate that an interface type declaration is intended to be a *functional interface* as defined by the Java Language Specification. Conceptually, a functional interface has exactly one abstract method. Since [default methods](https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Method.html#isDefault--) have an implementation, they are not abstract. If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does *not* count toward the interface's abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere.

Note that instances of functional interfaces can be created with lambda expressions, method references, or constructor references.

If a type is annotated with this annotation type, compilers are required to generate an error message unless:

* The type is an interface type and not an annotation type, enum, or class.
* The annotated type satisfies the requirements of a functional interface.

However, the compiler will treat any interface meeting the definition of a functional interface as a functional interface regardless of whether or not a FunctionalInterface annotation is present on the interface declaration.

Example:

MyPersonalFunctionalInterface<String> mine = (x) -> System.***out***.println("Hello "+x);

mine.say("Vlad");

where MyPersonalFunctionalInterface is:

@FunctionalInterface

**public** **interface** MyPersonalFunctionalInterface<T> {

**public** **void** say(T arg);

}

## Method references

<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>

There are four kinds of method references:

|  |  |
| --- | --- |
| **Kind** | **Example** |
| Reference to a static method | ContainingClass::staticMethodName |
| Reference to an instance method of a particular object | containingObject::instanceMethodName |
| Reference to an instance method of an arbitrary object of a particular type | ContainingType::methodName |
| Reference to a constructor | ClassName::new |

### Reference to a Static Method

Here you can reference a static method from your class like this: YourClass::theStaticMethod. In our case it is: Car::compareByMileage. But this makes sense in a context because you are not providing the method arguments: Arrays.sort(carsArray, Car::compareByMileage); The method reference Car::compareByMileage is equivalent to the lambda expression:(a,b) -> Car.compareByMileage(a,b)

### Reference to an Instance Method of a Particular Object

Here is an example of accessing a non-static method on an instance. Again the context is important. You can do this only where the method is expecting a functional interface.

ComparisonProvider comparisonProvider = **new** ComparisonProvider();

Arrays.sort(carsArray, comparisonProvider::compareByBrand);

Here, comparisonProvider.compareByBrand would be:

**public** **class** ComparisonProvider {

**public** **int** compareByBrand(Car a, Car b) {

**return** a.getBrand().compareTo(b.getBrand());

}

}

In this case, comparisonProvider::compareByBrand invokes the compareByName method and JRE infers the method arguments (Car, Car).

Another interesting example here. Instead of this:

File[] hiddenFiles = **new** File(".").listFiles(**new** FileFilter() {

**public** **boolean** accept(File file) {

**return** file.isHidden();

}

});

Do this:

File[] hiddenFiles = **new** File(".").listFiles(File::isHidden);

### Reference to an Instance Method of an Arbitrary Object of a Particular Type

Here we show an example of an instance method of an arbitrary type:

String[] array = { "Zoi", "Wow", "Aoc", "Dao", "Fao", "Aao" };

Arrays.*sort*(array, String::compareToIgnoreCase);

### Reference to a Constructor

You can refer to constructors just like you would refer to a static method:

Set<Car> carsSet = transferElements(cars, HashSet::**new**);

The Java compiler infers that you want to create a HashSet collection that contains elements of type Person. Alternatively, you can specify this as follows:

Set<Car> carsSetNew = *transferElements*(cars, HashSet<Car>::**new**);

This is equivalent to:

Set<Car> carsSetLambda = *transferElements*(cars, ()->{**return** **new** HashSet<>();});

Where transferElements is:

**public** **static** <T, SOURCE **extends** Collection<T>, DEST **extends** Collection<T>>

DEST transferElements(SOURCE sourceCollection, Supplier<DEST> collectionFactory) {

DEST result = collectionFactory.get();

**for** (T t : sourceCollection) {

result.add(t);

}

**return** result;

}

The functional interface Supplier contains one method get that takes no arguments and returns an object.

## Default methods and statics in interfaces

<https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html>

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

You specify that a method definition in an interface is a default method with the default keyword at the beginning of the method signature.

Here is an example of an interface called MonitorControls:

**public** **interface** MonitorControls {

**void** showMenu();

**void** adjustBrightness(Integer newValue);

**void** adjustContrast(Integer newValue);

**boolean** getStatus();

**void** setStatus(**boolean** on);

**default** **void** power() {

**if** (getStatus()) {

System.***out***.println("Turning off");

setStatus(**false**);

} **else** {

System.***out***.println("Turning on");

setStatus(**true**);

}

}

In this case, the method power is a default method. It provides the implementation. One interesting thing is that you can call other methods from default methods and they will get correctly resolved during runtime. Here is the class that implements this interface:

**public** **class** DesktopMonitorControls **implements** MonitorControls, Connectors {

**private** **boolean** on;

@Override

**public** **void** showMenu() {

System.***out***.println("Showing Menu");

}

@Override

**public** **void** adjustBrightness(Integer newValue) {

System.***out***.println("Adjusting Brightness");

}

@Override

**public** **void** adjustContrast(Integer newValue) {

System.***out***.println("Adjusting Contrast");

}

@Override

**public** **boolean** getStatus() {

**return** **this**.on;

}

@Override

**public** **void** setStatus(**boolean** on) {

**this**.on = on;

}

}

To be noted here that there is no implementation of the power method provided. If we use this class as it is:

MonitorControls controls = **new** DesktopMonitorControls();

controls.power();

controls.adjustBrightness(10);

controls.power();

we get the result:

Turning on

Adjusting Brightness

Turning off

As by magic, the power() method becomes available and more interestingly, it can correctly resolve the current status of the monitor by calling the getStatus() implementation.

What if you provide an implementation of power()?

@Override

**public** **void** power() {

System.***out***.println("NEW POWER");

}

In this case you get as expected:

NEW POWER

Adjusting Brightness

NEW POWER

Another interesting question: what happens if 2 interfaces have the same default method? Say you have this interface:

**public** **interface** Connectors {

**void** hdmi();

**void** rgb();

**default** **void** power(){

System.***out***.println("Power cable is present");

}

}

And your class implements this interface as well:

**public** **class** DesktopMonitorControls **implements** MonitorControls, Connectors {

In this case you’ll get the compilation error:

Duplicate default methods named power with the parameters () and () are inherited from the types Connectors and MonitorControls

So the compiler doesn’t know which default version to invoke during runtime, the one from Connectors or from MonitorControls. An IDE would propose to implement the method either from MonitorControls or from Connectors and it will generate either:

@Override

**public** **void** power() {

// **TODO** Auto-generated method stub

MonitorControls.**super**.power();

}

Or

@Override

**public** **void** power() {

// **TODO** Auto-generated method stub

Connectors.**super**.power();

}

Interestingly enough, it uses the keyword **super** which usually refers to classes. Care should be taken when using these default methods as they might get easily misused.

In addition to default methods, you can define static methods in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class.

You can add this into your Connectors interface:

**static** **void** mystat(){

System.***out***.println("Nothing Special");

}

And then use it:

Connectors.*mystat*();

## Repeating Annotations

There are some situations where you want to apply the same annotation to a declaration or type use. As of the Java SE 8 release,*repeating annotations* enable you to do this. Normally you cannot apply the same annotation twice, you’ll get a compilation exception. With this new mechanism it is now possible to do so, but there are 2 steps you need to follow. First you must declar the annotation and apply the @Repeatable annotation to it. In paranthesis you will indicate the container class (usually the plural of your annotation):

@Repeatable(MyAnnotations.**class**)

**public** **@interface** MyRepeatableAnnotation {

String name() **default** "noname";

**int** age() **default** 0;

}

Step 2 is to create your MyAnnotation class:

**public** **@interface** MyAnnotations {

MyRepeatableAnnotation[] value();

}

And here is the usage:

@MyRepeatableAnnotation(name = "Bob", age = 10)

@MyRepeatableAnnotation(name = "Alice", age = 12)

**private** **void** testAnnotation() {

System.***out***.println("Annotated method ");

}

That way you will be able to apply the same annotation with differen values.

However this is identical to (pre java 8):

@MyAnnotations({

@MyRepeatableAnnotation(name = "Bob", age = 10),

@MyRepeatableAnnotation(name = "Alice", age = 12) })

**private** **void** testMultiple() {

}

## Type Annotations

Before the Java SE 8 release, annotations could only be applied to declarations. As of the Java SE 8 release, annotations can also be applied to any type use. This means that annotations can be used anywhere you use a type. A few examples of where types are used are class instance creation expressions (new), casts, implements clauses, and throws clauses.

Type annotations were created to support improved analysis of Java programs way of ensuring stronger type checking. The Java SE 8 release does not provide a type checking framework, but it allows you to write (or download) a type checking framework that is implemented as one or more pluggable modules that are used in conjunction with the Java compiler.

For example, you want to ensure that a particular variable in your program is never assigned to null; you want to avoid triggering aNullPointerException. You can write a custom plug-in to check for this. You would then modify your code to annotate that particular variable, indicating that it is never assigned to null. The variable declaration might look like this:

**@NonNull** String str;

## Method parameter reflection

Method Parameter Reflection (RFE: [**JDK-8004841**](http://bugs.java.com/bugdatabase/view_bug.do?bug_id=8004841)): You can obtain the names of the formal parameters of any method or constructor with the method[**java.lang.reflect.Executable.getParameters**](https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Executable.html#getParameters--). However, .class files do not store formal parameter names by default. To store formal parameter names in a particular .classfile, and thus enable the Reflection API to retrieve formal parameter names, compile the source file with the -parameters option of the javac compiler.

# java.lang and java.util Packages

## Standard Encoding and Decoding Base64

<https://en.wikipedia.org/wiki/Base64>

A quote from Thomas Hobbes' *Leviathan* (be aware of spaces between lines):

Man is distinguished, not only by his reason, but by this singular passion from

other animals, which is a lust of the mind, that by a perseverance of delight

in the continued and indefatigable generation of knowledge, exceeds the short

vehemence of any carnal pleasure.

is represented as a byte sequence of 8-bit-padded [ASCII](https://en.wikipedia.org/wiki/ASCII) characters encoded in [MIME](https://en.wikipedia.org/wiki/MIME)'s Base64 scheme as follows:

TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJ5IGhpcyByZWFzb24sIGJ1dCBieSB0aGlz

IHNpbmd1bGFyIHBhc3Npb24gZnJvbSBvdGhlciBhbmltYWxzLCB3aGljaCBpcyBhIGx1c3Qgb2Yg

dGhlIG1pbmQsIHRoYXQgYnkgYSBwZXJzZXZlcmFuY2Ugb2YgZGVsaWdodCBpbiB0aGUgY29udGlu

dWVkIGFuZCBpbmRlZmF0aWdhYmxlIGdlbmVyYXRpb24gb2Yga25vd2xlZGdlLCBleGNlZWRzIHRo

ZSBzaG9ydCB2ZWhlbWVuY2Ugb2YgYW55IGNhcm5hbCBwbGVhc3VyZS4=

In the above quote, the encoded value of *Man* is *TWFu*. Encoded in ASCII, the characters *M*, *a*, and *n* are stored as the bytes 77, 97, and 110, which are the 8-bit binary values 01001101, 01100001, and 01101110. These three values are joined together into a 24-bit string, producing010011010110000101101110. Groups of 6 bits (6 bits have a maximum of 26 = 64 different binary values) are [converted into individual numbers](https://en.wikipedia.org/wiki/Binary_number#Counting_in_binary) from left to right (in this case, there are four numbers in a 24-bit string), which are then converted into their corresponding Base64 character values.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Text content** | **M** | | | | | | | | **a** | | | | | | | | **n** | | | | | | | |
| **ASCII** | 77 (0x4d) | | | | | | | | 97 (0x61) | | | | | | | | 110 (0x6e) | | | | | | | |
| **Bit pattern** | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| **Index** | 19 | | | | | | 22 | | | | | | 5 | | | | | | 46 | | | | | |
| **Base64-encoded** | **T** | | | | | | **W** | | | | | | **F** | | | | | | **u** | | | | | |

As this example illustrates, Base64 encoding converts three [octets](https://en.wikipedia.org/wiki/Octet_(computing)) into four encoded characters.

The Base64 index table:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Value** | **Char** |  | **Value** | **Char** |  | **Value** | **Char** |  | **Value** | **Char** |
| 0 | A | 16 | Q | 32 | g | 48 | w |
| 1 | B | 17 | R | 33 | h | 49 | x |
| 2 | C | 18 | S | 34 | i | 50 | y |
| 3 | D | 19 | T | 35 | j | 51 | z |
| 4 | E | 20 | U | 36 | k | 52 | 0 |
| 5 | F | 21 | V | 37 | l | 53 | 1 |
| 6 | G | 22 | W | 38 | m | 54 | 2 |
| 7 | H | 23 | X | 39 | n | 55 | 3 |
| 8 | I | 24 | Y | 40 | o | 56 | 4 |
| 9 | J | 25 | Z | 41 | p | 57 | 5 |
| 10 | K | 26 | a | 42 | q | 58 | 6 |
| 11 | L | 27 | b | 43 | r | 59 | 7 |
| 12 | M | 28 | c | 44 | s | 60 | 8 |
| 13 | N | 29 | d | 45 | t | 61 | 9 |
| 14 | O | 30 | e | 46 | u | 62 | + |
| 15 | P | 31 | f | 47 | v | 63 | / |

When the number of bytes to encode is not divisible by three (that is, if there are only one or two bytes of input for the last 24-bit block), then the following action is performed:

Add extra bytes with value zero so there are three bytes, and perform the conversion to base64. If there was only one significant input byte, only the first two base64 digits are picked (12 bits), and if there were two significant input bytes, the first three base64 digits are picked (18 bits). '=' characters might be added to make the last block contain four base64 characters.

As a result, when the last group contains one octet, the four [least significant bits](https://en.wikipedia.org/wiki/Least_significant_bit) of the final 6-bit block are set to zero; and when the last group contains two octets, the two least significant bits of the final 6-bit block are set to zero.

Here is an example of how to use Encoder and Decoder in Java 8:

**package** com.vvirlan;

**import** java.net.URL;

**import** java.**util**.Base64;

**class** Base64Sample {

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

**try** {

// Encode URL

URL originalUrl = **new** URL("http://example.com");

Base64.Encoder urlEncoder = Base64.*getUrlEncoder*();

System.***out***.println("URL: " + originalUrl);

**byte**[] encodedUrl = urlEncoder.encode(originalUrl.toString().getBytes("UTF8"));

System.***out***.println("Base64 Encoded URL : " + **new** String(encodedUrl, "UTF-8"));

// Encode text

String originalText = "thisIsAnExampleString";

Base64.Encoder textEncoder = Base64.*getEncoder*();

**byte**[] encodedText = textEncoder.encode(originalText.getBytes("UTF-8"));

System.***out***.println("Base64 Encoded String : " + **new** String(encodedText, "UTF-8"));

// Decode URL and text

Base64.Decoder urlDecoder = Base64.*getUrlDecoder*();

Base64.Decoder textDecoder = Base64.*getDecoder*();

**byte**[] urlDecoded = urlDecoder.decode(encodedUrl);

**byte**[] textDecoded = textDecoder.decode(encodedText);

System.***out***.println("Base64 Decoded URL : " + **new** String(urlDecoded, "UTF-8"));

System.***out***.println("Base64 Decoded String : " + **new** String(textDecoded, "UTF-8"));

}

**catch** (Exception e) {

System.***out***.println("Invalid URL Exception");

}

}

}

Here are the APIs for **Base64**:

public class **Base64** extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

This class consists exclusively of static methods for obtaining encoders and decoders for the Base64 encoding scheme. The implementation of this class supports the following types of Base64 as specified in [RFC 4648](http://www.ietf.org/rfc/rfc4648.txt) and [RFC 2045](http://www.ietf.org/rfc/rfc2045.txt).

**Basic**

Uses "The Base64 Alphabet" as specified in Table 1 of RFC 4648 and RFC 2045 for encoding and decoding operation. The encoder does not add any line feed (line separator) character. The decoder rejects data that contains characters outside the base64 alphabet.

**URL and Filename safe**

Uses the "URL and Filename safe Base64 Alphabet" as specified in Table 2 of RFC 4648 for encoding and decoding. The encoder does not add any line feed (line separator) character. The decoder rejects data that contains characters outside the base64 alphabet.

**MIME**

Uses the "The Base64 Alphabet" as specified in Table 1 of RFC 2045 for encoding and decoding operation. The encoded output must be represented in lines of no more than 76 characters each and uses a carriage return '\r' followed immediately by a linefeed '\n' as the line separator. No line separator is added to the end of the encoded output. All line separators or other characters not found in the base64 alphabet table are ignored in decoding operation.

Unless otherwise noted, passing a null argument to a method of this class will cause a [NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) to be thrown.

|  |  |
| --- | --- |
| static [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) | [**getDecoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getDecoder--)()  Returns a [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) that decodes using the [**Basic**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#basic) type base64 encoding scheme. |
| static [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**getEncoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getEncoder--)()  Returns a [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) that encodes using the [**Basic**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#basic) type base64 encoding scheme. |
| static [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) | [**getMimeDecoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getMimeDecoder--)()  Returns a [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) that decodes using the [**MIME**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#mime) type base64 decoding scheme. |
| static [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**getMimeEncoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getMimeEncoder--)()  Returns a [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) that encodes using the [**MIME**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#mime) type base64 encoding scheme. |
| static [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**getMimeEncoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getMimeEncoder-int-byte:A-)(int lineLength, byte[] lineSeparator)  Returns a [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) that encodes using the [**MIME**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#mime) type base64 encoding scheme with specified line length and line separators. |
| static [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) | [**getUrlDecoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getUrlDecoder--)()  Returns a [**Base64.Decoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) that decodes using the [**URL and Filename safe**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#url) type base64 encoding scheme. |
| static [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**getUrlEncoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#getUrlEncoder--)()  Returns a [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) that encodes using the [**URL and Filename safe**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html#url) type base64 encoding scheme. |

public static class **Base64.Decoder** extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. The Base64 padding character '=' is accepted and interpreted as the end of the encoded byte data, but is not required. So if the final unit of the encoded byte data only has two or three Base64 characters (without the corresponding padding character(s) padded), they are decoded as if followed by padding character(s). If there is a padding character present in the final unit, the correct number of padding character(s) must be present, otherwise IllegalArgumentException (IOException when reading from a Base64 stream) is thrown during decoding.

Instances of [Base64.Decoder](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html) class are safe for use by multiple concurrent threads.

Unless otherwise noted, passing a null argument to a method of this class will cause a [NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) to be thrown.

|  |  |
| --- | --- |
| byte[] | [**decode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#decode-byte:A-)(byte[] src)  Decodes all bytes from the input byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme, writing the results into a newly-allocated output byte array. |
| int | [**decode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#decode-byte:A-byte:A-)(byte[] src, byte[] dst)  Decodes all bytes from the input byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme, writing the results into the given output byte array, starting at offset 0. |
| [**ByteBuffer**](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) | [**decode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#decode-java.nio.ByteBuffer-)([**ByteBuffer**](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) buffer)  Decodes all bytes from the input byte buffer using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme, writing the results into a newly-allocated ByteBuffer. |
| byte[] | [**decode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#decode-java.lang.String-)([**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) src)  Decodes a Base64 encoded String into a newly-allocated byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme. |
| [**InputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html) | [**wrap**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Decoder.html#wrap-java.io.InputStream-)([**InputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html) is)  Returns an input stream for decoding [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoded byte stream. |

public static class **Base64.Encoder** extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. Instances of [Base64.Encoder](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) class are safe for use by multiple concurrent threads. Unless otherwise noted, passing a null argument to a method of this class will cause a [NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html) to be thrown.

|  |  |
| --- | --- |
| byte[] | [**encode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#encode-byte:A-)(byte[] src)  Encodes all bytes from the specified byte array into a newly-allocated byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme. |
| int | [**encode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#encode-byte:A-byte:A-)(byte[] src, byte[] dst)  Encodes all bytes from the specified byte array using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme, writing the resulting bytes to the given output byte array, starting at offset 0. |
| [**ByteBuffer**](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) | [**encode**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#encode-java.nio.ByteBuffer-)([**ByteBuffer**](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) buffer)  Encodes all remaining bytes from the specified byte buffer into a newly-allocated ByteBuffer using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html)encoding scheme. |
| [**String**](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) | [**encodeToString**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#encodeToString-byte:A-)(byte[] src)  Encodes the specified byte array into a String using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme. |
| [**Base64.Encoder**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html) | [**withoutPadding**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#withoutPadding--)()  Returns an encoder instance that encodes equivalently to this one, but without adding any padding character at the end of the encoded byte data. |
| [**OutputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html) | [**wrap**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.Encoder.html#wrap-java.io.OutputStream-)([**OutputStream**](https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html) os)  Wraps an output stream for encoding byte data using the [**Base64**](https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html) encoding scheme. |

## Unsigned Arithmetic Support

The support is implemented via static methods, primarily on java.lang.Integer and java.lang.Long, that:

* Provide bidirectional conversion between strings and unsigned integers
* Compare values as unsigned
* Compute unsigned divide and remainder

Colloquially, "unsigned integer" means a 32-bit int or 64-bit long value where all the bits are interpreted as contributing to the magnitude. In the unsigned realm, the values of an integer type of a given bit-width range from 0 to 2width-1 rather than from -(2width-1) to 2width-1-1. To avoid dealing with the overhead of boxed values and to allow reuse of the built-in arithmetic operators, the unsigned API support does not introduce new types like UnsignedInt with instance methods to perform addition, subtraction, etc. However, that lack of separate Java-level unsigned types does mean a programmer can accidentally improperly mix signed and unsigned values.

# Stream API

## Aggregate Operations

<https://docs.oracle.com/javase/tutorial/collections/streams/index.html>

**Differences Between Aggregate Operations and Iterators**

Aggregate operations, like forEach, appear to be like iterators. However, they have several fundamental differences:

* **They use internal iteration**: Aggregate operations do not contain a method like next to instruct them to process the next element of the collection. With internal delegation, your application determines what collection it iterates, but the JDK determines how to iterate the collection. With external iteration, your application determines both what collection it iterates and how it iterates it. However, external iteration can only iterate over the elements of a collection sequentially. Internal iteration does not have this limitation. It can more easily take advantage of parallel computing, which involves dividing a problem into sub-problems, solving those problems simultaneously, and then combining the results of the solutions to the sub-problems. See the section [Parallelism](https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html) for more information.
* **They process elements from a stream**: Aggregate operations process elements from a stream, not directly from a collection. Consequently, they are also called stream operations.
* **They support behavior as parameters**: You can specify [lambda expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html) as parameters for most aggregate operations. This enables you to customize the behavior of a particular aggregate operation.

## Reduction

<https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html>

Map<Person.Sex, List<Person>> byGender =

persons

.stream()

.collect(Collectors.*groupingBy*(Person::getSex));

System.***out***.println(byGender);

## Parallelism

<https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html>

An example of parallel stream processing:

Date start = **new** Date();

**double** average = persons

.parallelStream()

.filter(p->p.getSex().equals(Person.Sex.***MALE***))

.mapToInt(Person::getAge)

.average()

.getAsDouble();

Date end = **new** Date();

**long** diff = end.getTime() - start.getTime();

System.***out***.println("Average age of Males: "+average+" Time: "+diff+" ms.");

## Collectors Examples

### AveragingDouble

**double** avgLength = users.stream().collect(Collectors.*averagingDouble*(String::length));

System.***out***.println("Average length: "+avgLength);

Will produce:

Average length: 3.6666666666666665

### Joining

String joined = users.stream().collect(Collectors.*joining*("|"));

System.***out***.println(joined);

Will produce:

Alice|Bob|Bob|Josh|Zair|Mat

## Other Streaming API Examples

### Filter

Here is an example of filtering:

List<String> users = Arrays.*asList*("Alice", "Bob", "Josh", "Zair", "Mat");

users

.stream()

.filter(x -> x.length() == 4)

.forEach(x -> System.***out***.println(x));

}

### Map

List<String> uppercased = users.stream().map(String::toUpperCase).collect(Collectors.*toList*());

System.***out***.println(uppercased);

//OR

uppercased = users.stream().map(x -> x.toUpperCase()).collect(Collectors.*toList*());

System.***out***.println(uppercased);

### MapToInt

List<String> users = Arrays.*asList*("Alice", "Bob", "Josh", "Zair", "Mat");

users

.stream()

.mapToInt(x -> x.length())

.forEach(x -> System.***out***.println(x));

//OR

users

.stream()

.mapToInt(String::length)

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

### Count

System.***out***.println(users.stream().count());

### Distinct

List<String> distinctUsers = users.stream().distinct().collect(Collectors.*toList*());

usersunique.stream().forEach(System.***out***::println);

### Limit

List<String> limited = users.stream().limit(4).collect(Collectors.*toList*());

System.***out***.println(limited);

# Date Time API

A long-standing bugbear of Java developers has been the inadequate support for the date and time use cases of ordinary developers.

For example, the existing classes (such as java.util.Date and SimpleDateFormatter) aren’t thread-safe, leading to potential concurrency issues for users—not something the average developer would expect to deal with when writing date-handling code.

Some of the date and time classes also exhibit quite poor API design. For example, years in java.util.Date start at 1900, months start at 1, and days start at 0—not very intuitive.

These issues, and several others, have led to the popularity of third-party date and time libraries, such as Joda-Time.

In order to address these problems and provide better support in the JDK core, a new date and time API, which is free of these problems, has been designed for Java SE 8. It is under JSR 310, and will appear in the new Java SE 8 package java.time.

The new API is driven by three core ideas:

* **Immutable-value classes.** One of the serious weaknesses of the existing formatters in Java is that they aren’t thread-safe. This puts the burden on developers to use them in a thread-safe manner and to think about concurrency problems in their day-to-day development of date-handling code. The new API avoids this issue by ensuring that all its core classes are immutable and represent well-defined values.
* **Domain-driven design.** The new API models its domain very precisely with classes that represent different use cases for Date andTime closely. This differs from previous Java libraries that were quite poor in that regard. For example, java.util.Date represents an instant on the timeline—a wrapper around the number of milli-seconds since the UNIX epoch—but if you call toString(), the result suggests that it has a time zone, causing confusion among developers.

This emphasis on domain-driven design offers long-term benefits around clarity and understandability, but you might need to think through your application’s domain model of dates when porting from previous APIs to Java SE 8.

* **Separation of chronologies.** The new API allows people to work with different calendaring systems in order to support the needs of users in some areas of the world, such as Japan or Thailand, that don’t necessarily follow ISO-8601. It does so without imposing additional burden on the majority of developers, who need to work only with the standard chronology.

The Date-Time API consists of the primary package, java.time, and four subpackages:

java.time

The core of the API for representing date and time. It includes classes for date, time, date and time combined, time zones, instants, duration, and clocks. These classes are based on the calendar system defined in ISO-8601, and are immutable and thread-safe.

java.time.chrono

The API for representing calendar systems other than the default ISO-8601. You can also define your own calendar system. This tutorial does not cover this package in any detail.

java.time.format

Classes for formatting and parsing dates and times.

java.time.temporal

Extended API, primarily for framework and library writers, allowing interoperations between the date and time classes, querying, and adjustment. Fields (TemporalField and ChronoField) and units (TemporalUnit and ChronoUnit) are defined in this package.

java.time.zone

Classes that support time zones, offsets from time zones, and time zone rules. If working with time zones, most developers will need to use only ZonedDateTime, and ZoneId or ZoneOffset.

The Date-Time API offers a rich set of methods within a rich set of classes. The method names are made consistent between classes wherever possible. For example, many of the classes offer a now method that captures the date or time values of the current moment that are relevant to that class. There are from methods that allow conversion from one class to another.

There is also standardization regarding the method name prefixes. Because most of the classes in the Date-Time API are immutable, the API does not include set methods. (After its creation, the value of an immutable object cannot be changed. The immutable equivalent of aset method is with.) The following table lists the commonly used prefixes:

|  |  |  |
| --- | --- | --- |
| **Prefix** | **Method Type** | **Use** |
| of | static factory | Creates an instance where the factory is primarily validating the input parameters, not converting them. |
| from | static factory | Converts the input parameters to an instance of the target class, which may involve losing information from the input. |
| parse | static factory | Parses the input string to produce an instance of the target class. |
| format | instance | Uses the specified formatter to format the values in the temporal object to produce a string. |
| get | instance | Returns a part of the state of the target object. |
| is | instance | Queries the state of the target object. |
| with | instance | Returns a copy of the target object with one element changed; this is the immutable equivalent to a set method on a JavaBean. |
| plus | instance | Returns a copy of the target object with an amount of time added. |
| minus | instance | Returns a copy of the target object with an amount of time subtracted. |
| to | instance | Converts this object to another type. |
| at | instance | Combines this object with another. |

### LocalDate and LocalTime

The first classes you will probably encounter when using the new API are LocalDate and LocalTime. They are local in the sense that they represent date and time from the context of the observer, such as a calendar on a desk or a clock on your wall. There is also a composite class called LocalDateTime, which is a pairing of LocalDate and LocalTime.

LocalDateTime localDateTime = LocalDateTime.*now*();

System.***out***.println("LocalDateTime.now = " + localDateTime);

LocalDate localDate = localDateTime.toLocalDate();

System.***out***.println("ldt.toLocalDate() = " + localDate);

Month mont = localDateTime.getMonth();

System.***out***.println("Month is : "+mont);

System.***out***.println("Day of monts: "+localDateTime.getDayOfMonth());

System.***out***.println("LocalDate.of(2015, 12, 04) = " + LocalDate.*of*(2015, 12, 04));

System.***out***.println("LocalTime.of(20,59) = " + LocalTime.*of*(20, 59));

System.***out***.println("LocalTime.parse(12:10:55) = " + LocalTime.*parse*("12:10:55"));

LocalDateTime thefuture = localDateTime.withDayOfMonth(22).withDayOfYear(200).plus(10, ChronoUnit.***CENTURIES***);

System.***out***.println(thefuture);

The new API supports different precision time points by offering types to represent a date, a time, and date with time, but obviously there are notions of precision that are more fine-grained than this.

The truncatedTo method exists to support such use cases, and it allows you to truncate a value to a field, as shown in Listing 5. LocalTime truncatedTime = time.truncatedTo(ChronoUnit.SECONDS);

**Time Zones**

The local classes that we looked at previously abstract away the complexity introduced by time zones. A time zone is a set of rules, corresponding to a region in which the standard time is the same. There are about 40 of them. Time zones are defined by their offset from Coordinated Universal Time (UTC). They move roughly in sync, but by a specified difference.

Time zones can be referred to by two identifiers: abbreviated, for example, “PLT,” and longer, for example, “Asia/Karachi.” When designing your application, you should consider what scenarios are appropriate for using time zones and when offsets are appropriate.

* ZoneId is an identifier for a region (see Listing 6). Each ZoneId corresponds to some rules that define the time zone for that location. When designing your software, if you consider throwing around a string such as “PLT” or “Asia/Karachi,” you should use this domain class instead. An example use case would be storing users’ preferences for their time zone.

// You can specify the zone id when creating a zoned date time

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

ZonedDateTime zoned = ZonedDateTime.of(dateTime, id);

assertEquals(id, ZoneId.from(zoned));

**Listing 6**

* ZoneOffset is the period of time representing a difference between Greenwich/UTC and a time zone. This can be resolved for a specific ZoneId at a specific moment in time, as shown in Listing 7.

ZoneOffset offset = ZoneOffset.of("+2:00");

**Listing 7**

**Time Zone Classes**

* ZonedDateTime is a date and time with a fully qualified time zone (see Listing 8). This can resolve an offset at any point in time. The rule of thumb is that if you want to represent a date and time without relying on the context of a specific server, you should useZonedDateTime.
* ZonedDateTime.parse("2007-12-03T10:15:30+01:00[Europe/Paris]");

**Listing 8**

* OffsetDateTime is a date and time with a resolved offset. This is useful for serializing data into a database and also should be used as the serialization format for logging time stamps if you have servers in different time zones.
* OffsetTime is a time with a resolved offset, as shown in Listing 9.

OffsetTime time = OffsetTime.now();

// changes offset, while keeping the same point on the timeline

OffsetTime sameTimeDifferentOffset = time.withOffsetSameInstant(offset);

// changes the offset, and updates the point on the timeline

OffsetTime changeTimeWithNewOffset = time.withOffsetSameLocal(offset);

// Can also create new object with altered fields as before

changeTimeWithNewOffset.withHour(3).plusSeconds(2); **Listing 9**

There is an existing time zone class in Java—java.util.TimeZone—but it isn’t used by Java SE 8 be-cause all JSR 310 classes are immutable and time zone is mutable.

**Periods**

A Period represents a value such as “3 months and 1 day,” which is a distance on the timeline. This is in contrast to the other classes we’ve looked at so far, which have been points on the timeline. See Listing 10.

// 3 years, 2 months, 1 day

Period period = Period.of(3, 2, 1);

// You can modify the values of dates using periods

LocalDate newDate = oldDate.plus(period);

ZonedDateTime newDateTime = oldDateTime.minus(period);

// Components of a Period are represented by ChronoUnit values

assertEquals(1, period.get(ChronoUnit.DAYS));

**Listing 10**

**Java SE 8 will ship with a new date and time API in java.time** that offers greatly improved safety and functionality for developers. The new API models the domain well, with a good selection of classes for modeling a wide variety of developer use cases.

**Durations**

A Duration is a distance on the timeline measured in terms of time, and it fulfills a similar purpose to Period, but with different precision, as shown in Listing 11.

// A duration of 3 seconds and 5 nanoseconds

Duration duration = Duration.ofSeconds(3, 5);

Duration oneDay = Duration.between(today, yesterday); **Listing 11**

It’s possible to perform normal plus, minus, and “with” operations on a Durationinstance and also to modify the value of a date or time using the Duration.

**Chronologies**

In order to support the needs of developers using non-ISO calendaring systems, Java SE 8 introduces the concept of a Chronology, which represents a calendaring system and acts as a factory for time points within the calendaring system. There are also interfaces that correspond to core time point classes, but are parameterized by

Chronology:

ChronoLocalDate

ChronoLocalDateTime

ChronoZonedDateTime

These classes are there purely for developers who are working on highly internationalized applications that need to take into account local calendaring systems, and they shouldn’t be used by developers without these requirements. Some calendaring systems don’t even have a concept of a month or a week and calculations would need to be performed via the very generic field API.

|  |
| --- |
| **Learn More**  [**JSR 310**](http://jcp.org/en/jsr/detail?id=310) |

**The Rest of the API**

Java SE 8 also has classes for some other common use cases. There is the MonthDay class, which contains a pair ofMonth and Day and is useful for representing birthdays. The YearMonth class covers the credit card start date and expiration date use cases and scenarios in which people have a date with no specified day.

JDBC in Java SE 8 will support these new types, but there will be no public JDBC API changes. The existing generic setObject andgetObject methods will be sufficient.

These types can be mapped to vendor-specific database types or ANSI SQL types; for example, the ANSI mapping looks like Table 1.

|  |  |
| --- | --- |
| ANSI SQL | Java SE 8 |
| DATE | LocalDate |
| TIME | LocalTime |
| TIMESTAMP | LocalDateTime |
| TIME WITH TIMEZONE | OffsetTime |
| TIMESTAMP WITH TIMEZONE | OffsetDateTime |

**Table 1**

One of the core classes of the Date-Time API is the [Instant](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html) class, which represents the start of a nanosecond on the timeline. This class is useful for generating a time stamp to represent machine time.

**import** java.time.Instant;

Instant timestamp = Instant.now();

A value returned from the Instant class counts time beginning from the first second of January 1, 1970 (1970-01-01T00:00:00Z) also called the[EPOCH](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html#EPOCH). An instant that occurs before the epoch has a negative value, and an instant that occurs after the epoch has a positive value.

The other constants provided by the Instant class are [MIN](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html#MIN), representing the smallest possible (far past) instant, and [MAX](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html#MAX), representing the largest (far future) instant.

Invoking toString on an Instant produces output like the following:

2013-05-30T23:38:23.085Z

This format follows the [ISO-8601](http://www.iso.org/iso/home/standards/iso8601.htm) standard for representing date and time.

The Instant class provides a variety of methods for manipulating an Instant. There are plus and minus methods for adding or subtracting time. The following code adds 1 hour to the current time:

Instant oneHourLater = Instant.now().plusHours(1);

There are methods for comparing instants, such as [isAfter](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html#isAfter-java.time.Instant-) and [isBefore](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html#isBefore-java.time.Instant-). The [until](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html#until-java.time.temporal.Temporal-java.time.temporal.TemporalUnit-) method returns how much time exists between twoInstant objects. The following line of code reports how many seconds have occurred since the beginning of the Java epoch.

**long** secondsFromEpoch = Instant.ofEpochSecond(0L).until(Instant.now(),

ChronoUnit.SECONDS);

The Instant class does not work with human units of time, such as years, months, or days. If you want to perform calculations in those units, you can convert an Instant to another class, such as LocalDateTime or ZonedDateTime, by binding the Instant with a time zone. You can then access the value in the desired units. The following code converts an Instant to a LocalDateTime object using the [ofInstant](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html#ofInstant-java.time.Instant-java.time.ZoneId-) method and the default time zone, and then prints out the date and time in a more readable form:

Instant timestamp;

...

LocalDateTime ldt = LocalDateTime.ofInstant(timestamp, ZoneId.systemDefault());

System.***out***.printf("%s %d %d at %d:%d%n", ldt.getMonth(), ldt.getDayOfMonth(),

ldt.getYear(), ldt.getHour(), ldt.getMinute());

The output will be similar to the following:

MAY 30 2013 at 18:21

Either a ZonedDateTime or an OffsetTimeZone object can be converted to an Instant object, as each maps to an exact moment on the timeline. However, the reverse is not true. To convert an Instant object to a ZonedDateTime or an OffsetDateTime object requires supplying time zone, or time zone offset, information.

### Parsing and Formatting

The temporal-based classes in the Date-Time API provide parse methods for parsing a string that contains date and time information. These classes also provide format methods for formatting temporal-based objects for display. In both cases, the process is similar: you provide a pattern to the DateTimeFormatter to create a formatter object. This formatter is then passed to the parse or format method.

The DateTimeFormatter class provides numerous [predefined formatters](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#predefined), or you can define your own.

The parse and the format methods throw an exception if a problem occurs during the conversion process. Therefore, your parse code should catch the DateTimeParseException error and your format code should catch the DateTimeException error. For more information on exception handing, see [Catching and Handling Exceptions](https://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html).

The DateTimeFormatter class is both immutable and thread-safe; it can (and should) be assigned to a static constant where appropriate.

**Version Note:** The java.time date-time objects can be used directly with java.util.Formatter and String.format by using the familiar pattern-based formatting that was used with the legacy java.util.Date and java.util.Calendar classes.

#### Parsing

The one-argument [parse(CharSequence)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#parse-java.lang.CharSequence-) method in the LocalDate class uses the ISO\_LOCAL\_DATE formatter. To specify a different formatter, you can use the two-argument [parse(CharSequence, DateTimeFormatter)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#parse-java.lang.CharSequence-java.time.format.DateTimeFormatter-) method. The following example uses the predefined BASIC\_ISO\_DATEformatter, which uses the format 19590709 for July 9, 1959.

String in = ...;

LocalDate date = LocalDate.parse(in, DateTimeFormatter.BASIC\_ISO\_DATE);

You can also define a formatter using your own pattern. The following code, from the [Parse](https://docs.oracle.com/javase/tutorial/datetime/iso/examples/Parse.java) example, creates a formatter that applies a format of "MMM d yyyy". This format specifies three characters to represent the month, one digit to represent day of the month, and four digits to represent the year. A formatter created using this pattern would recognize strings such as "Jan 3 2003" or "Mar 23 1994". However, to specify the format as "MMM dd yyyy", with two characters for day of the month, then you would have to always use two characters, padding with a zero for a one-digit date: "Jun 03 2003".

String input = ...;

**try** {

DateTimeFormatter formatter =

DateTimeFormatter.ofPattern("MMM d yyyy");

LocalDate date = LocalDate.parse(input, formatter);

System.out.printf("%s%n", date);

}

**catch** (DateTimeParseException exc) {

System.out.printf("%s is not parsable!%n", input);

**throw** exc; // Rethrow the exception.

}

// 'date' has been successfully parsed

The documentation for the DateTimeFormatter class specifies the [full list of symbols](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#patterns) that you can use to specify a pattern for formatting or parsing.

The StringConverter example on the [Non-ISO Date Conversion](https://docs.oracle.com/javase/tutorial/datetime/iso/nonIso.html) page provides another example of a date formatter.

#### Formatting

The [format(DateTimeFormatter)](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html#format-java.time.format.DateTimeFormatter-) method converts a temporal-based object to a string representation using the specified format. The following code, from the [Flight](https://docs.oracle.com/javase/tutorial/datetime/iso/examples/Flight.java) example, converts an instance of ZonedDateTime using the format "MMM d yyy hh:mm a". The date is defined in the same manner as was used for the previous parsing example, but this pattern also includes the hour, minutes, and a.m. and p.m. components.

ZoneId leavingZone = ...;

ZonedDateTime departure = ...;

**try** {

DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM d yyyy hh:mm a");

String out = departure.format(format);

System.out.printf("LEAVING: %s (%s)%n", out, leavingZone);

}

**catch** (DateTimeException exc) {

System.out.printf("%s can't be formatted!%n", departure);

**throw** exc;

}

The output for this example, which prints both the arrival and departure time, is as follows:

LEAVING: Jul 20 2013 07:30 PM (America/Los\_Angeles)

ARRIVING: Jul 21 2013 10:20 PM (Asia/Tokyo)

### Temporal Adjuster

The [TemporalAdjuster](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjuster.html) interface, in the java.time.temporal package, provides methods that take a Temporal value and return an adjusted value. The adjusters can be used with any of the temporal-based types.

If an adjuster is used with a ZonedDateTime, then a new date is computed that preserves the original time and time zone values.

#### Predefined Adjusters

The [TemporalAdjusters](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjusters.html) class (note the plural) provides a set of predefined adjusters for finding the first or last day of the month, the first or last day of the year, the last Wednesday of the month, or the first Tuesday after a specific date, to name a few examples. The predefined adjusters are defined as static methods and are designed to be used with the [static import](https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html#staticimport) statement.

The following example uses several TemporalAdjusters methods, in conjunction with the with method defined in the temporal-based classes, to compute new dates based on the original date of 15 October 2000:

LocalDate date = LocalDate.of(2000, Month.OCTOBER, 15);

DayOfWeek dotw = date.getDayOfWeek();

System.***out***.printf("%s is on a %s%n", date, dotw);

System.***out***.printf("first day of Month: %s%n",

date.with(TemporalAdjusters.firstDayOfMonth()));

System.***out***.printf("first Monday of Month: %s%n",

date.with(TemporalAdjusters.firstInMonth(DayOfWeek.MONDAY)));

System.***out***.printf("last day of Month: %s%n",

date.with(TemporalAdjusters.lastDayOfMonth()));

System.***out***.printf("first day of next Month: %s%n",

date.with(TemporalAdjusters.firstDayOfNextMonth()));

System.***out***.printf("first day of next Year: %s%n", date.with(TemporalAdjusters.firstDayOfNextYear()));

System.***out***.printf("first day of Year: %s%n",

date.with(TemporalAdjusters.firstDayOfYear()));

This produces the following output:

2000-10-15 is on a SUNDAY

first day of Month: 2000-10-01

first Monday of Month: 2000-10-02

last day of Month: 2000-10-31

first day of next Month: 2000-11-01

first day of next Year: 2001-01-01

first day of Year: 2000-01-01

#### Custom Adjusters

You can also create your own custom adjuster. To do this, you create a class that implements the TemporalAdjuster interface with a[adjustInto(Temporal)](https://docs.oracle.com/javase/8/docs/api/java/time/temporal/TemporalAdjuster.html#adjustInto-java.time.temporal.Temporal-) method. The [PaydayAdjuster](https://docs.oracle.com/javase/tutorial/datetime/iso/examples/PaydayAdjuster.java) class from the [NextPayday](https://docs.oracle.com/javase/tutorial/datetime/iso/examples/NextPayday.java) example is a custom adjuster. The PaydayAdjuster evaluates the passed-in date and returns the next payday, assuming that payday occurs twice a month: on the 15th, and again on the last day of the month. If the computed date occurs on a weekend, then the previous Friday is used. The current calendar year is assumed.

/\*\*

\* The adjustInto method accepts a Temporal instance

\* and returns an adjusted LocalDate. If the passed in

\* parameter is not a LocalDate, then a DateTimeException is thrown.

\*/

**public** Temporal adjustInto(Temporal input) {

LocalDate date = LocalDate.from(input);

**int** day;

**if** (date.getDayOfMonth() < 15) {

day = 15;

} **else** {

day = date.with(TemporalAdjusters.lastDayOfMonth()).getDayOfMonth();

}

date = date.withDayOfMonth(day);

**if** (date.getDayOfWeek() == DayOfWeek.SATURDAY ||

date.getDayOfWeek() == DayOfWeek.SUNDAY) {

date = date.with(TemporalAdjusters.previous(DayOfWeek.FRIDAY));

}

**return** input.with(date);

}

The adjuster is invoked in the same manner as a predefined adjuster, using the with method. The following line of code is from theNextPayday example:

LocalDate nextPayday = date.with(**new** PaydayAdjuster());

In 2013, both June 15 and June 30 occur on the weekend. Running the NextPayday example with the respective dates of June 3 and June 18 (in 2013), gives the following results:

Given the date: 2013 Jun 3

the next payday: 2013 Jun 14

Given the date: 2013 Jun 18

the next payday: 2013 Jun 28

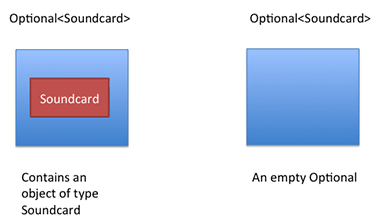
# Optional

public final class **Optional<T>** extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)

A container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.

Additional methods that depend on the presence or absence of a contained value are provided, such as [orElse()](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#orElse-T-) (return a default value if value not present) and [ifPresent()](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#ifPresent-java.util.function.Consumer-) (execute a block of code if the value is present). This is a [value-based](https://docs.oracle.com/javase/8/docs/api/java/lang/doc-files/ValueBased.html) class; use of identity-sensitive operations (including reference equality (==), identity hash code, or synchronization) on instances of Optional may have unpredictable results and should be avoided.

Java SE 8 introduces a new class called java.util.Optional<T> that is inspired from the ideas of Haskell and Scala. It is a class that encapsulates an optional value, as illustrated in Listing 2 below and in Figure 1. You can view Optional as a single-value container that either contains a value or doesn't (it is then said to be "empty"), as illustrated in Figure 2.



|  |  |
| --- | --- |
| Modifier and Type | Method and Description |
| static <T> [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<T> | [**empty**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#empty--)()  Returns an empty Optional instance. |
| boolean | [**equals**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#equals-java.lang.Object-)([**Object**](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html) obj)  Indicates whether some other object is "equal to" this Optional. |
| [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<[T](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)> | [**filter**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#filter-java.util.function.Predicate-)([**Predicate**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)> predicate)  If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional. |
| <U> [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<U> | [**flatMap**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#flatMap-java.util.function.Function-)([**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html),[**Optional**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<U>> mapper)  If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. |
| [T](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) | [**get**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#get--)()  If a value is present in this Optional, returns the value, otherwise throwsNoSuchElementException. |
| int | [**hashCode**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#hashCode--)()  Returns the hash code value of the present value, if any, or 0 (zero) if no value is present. |
| void | [**ifPresent**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#ifPresent-java.util.function.Consumer-)([**Consumer**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)> consumer)  If a value is present, invoke the specified consumer with the value, otherwise do nothing. |
| boolean | [**isPresent**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#isPresent--)()  Return true if there is a value present, otherwise false. |
| <U> [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<U> | [**map**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#map-java.util.function.Function-)([**Function**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html),? extends U> mapper)  If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result. |
| static <T> [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<T> | [**of**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#of-T-)(T value)  Returns an Optional with the specified present non-null value. |
| static <T> [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<T> | [**ofNullable**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#ofNullable-T-)(T value)  Returns an Optional describing the specified value, if non-null, otherwise returns an emptyOptional. |
| [T](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) | [**orElse**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#orElse-T-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) other)  Return the value if present, otherwise return other. |
| [T](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) | [**orElseGet**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#orElseGet-java.util.function.Supplier-)([**Supplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html)<? extends [**T**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)> other)  Return the value if present, otherwise invoke other and return the result of that invocation. |
| <X extends [Throwable](https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html)> [T](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) | [**orElseThrow**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#orElseThrow-java.util.function.Supplier-)([**Supplier**](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html)<? extends X> exceptionSupplier)  Return the contained value, if present, otherwise throw an exception to be created by the provided supplier. |
| [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) | [**toString**](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#toString--)()  Returns a non-empty string representation of this Optional suitable for debugging. |

Example:

Computer comp = **new** Computer();

Optional<Soundcard> snd = comp.getSoundcard();

snd.ifPresent(System.***out***::println);

And here is the Computer class:

**private** Soundcard soundcard ;

**public** Optional<Soundcard> getSoundcard() {

**return** Optional.*ofNullable*(soundcard);

}

# Concurrency

The [java.util.concurrent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html) package contains two new interfaces and four new classes:

* Interface [CompletableFuture.AsynchronousCompletionTask](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.AsynchronousCompletionTask.html): A marker interface identifying asynchronous tasks produced by async methods.
* Interface [CompletionStage<T>](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html): A stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes.
* Class [CompletableFuture<T>](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html): A Future that may be explicitly completed (setting its value and status), and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.
* Class [ConcurrentHashMap.KeySetView<K,V>](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html): A view of a ConcurrentHashMap as a Set of keys, in which additions may optionally be enabled by mapping to a common value.
* Class [CountedCompleter<T>](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountedCompleter.html): A ForkJoinTask with a completion action performed when triggered and there are no remaining pending actions.
* Class [CompletionException](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionException.html): Exception thrown when an error or other exception is encountered in the course of completing a result or task.

The [Collections Framework](https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html) has undergone a major revision in Java 8 to add aggregate operations based on the newly added [streams facility](https://docs.oracle.com/javase/8/docs/technotes/guides/language/lambda_api_jdk8.html) and [lambda expressions](https://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html#javase8). As a result, the[ConcurrentHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html) class introduces over 30 new methods in this release. These include various forEach methods (forEach, forEachKey, forEachValue, and forEachEntry), search methods (search, searchKeys, searchValues, and searchEntries) and a large number of reduction methods (reduce, reduceToDouble, reduceToLong etc.)

Other miscellaneous methods (mappingCount and newKeySet) have been added as well. As a result of the JDK 8 changes, ConcurrentHashMaps (and classes built from them) are now more useful as caches. These changes include methods to compute values for keys when they are not present, plus improved support for scanning (and possibly evicting) entries, as well as better support for maps with large numbers of elements.

#### New classes in java.util.concurrent.atomic

Maintaining a single count, sum, etc. that is updated by possibly many threads is a common scalability problem. This release introduces scalable updatable variable support through a small set of new classes (DoubleAccumulator, DoubleAdder, LongAccumulator, LongAdder), which internally employ contention-reduction techniques that provide huge throughput improvements as compared to Atomic variables. This is made possible by relaxing atomicity guarantees in a way that is acceptable in most applications.

* [DoubleAccumulator](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/DoubleAccumulator.html): One or more variables that together maintain a running double value updated using a supplied function.
* [DoubleAdder](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/DoubleAdder.html): One or more variables that together maintain an initially zero double sum.
* [LongAccumulator](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAccumulator.html): One or more variables that together maintain a running long value updated using a supplied function.
* [LongAdder](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html): One or more variables that together maintain an initially zero long sum.

#### New methods in java.util.concurrent.ForkJoinPool

A static commonPool() method is now available and appropriate for most applications. The common pool is used by any ForkJoinTask that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use, and reinstated upon subsequent use). Two new methods ([getCommonPoolParallelism()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#getCommonPoolParallelism%28%29) and [commonPool()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#commonPool%28%29)) have been added, which return the targeted parallelism level of the common pool, or the common pool instance, respectively.

#### New class java.util.concurrent.locks.StampedLock

A new StampedLock class adds a capability-based lock with three modes for controlling read/write access (writing, reading, and optimistic reading). This class also supports methods that conditionally provide conversions across the three modes. For complete details, see the [java.util.concurrent.StampedLock](https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/package-summary.html) API documentation.

Example:

Runnable runnable = () -> {

**try** {

String name = Thread.*currentThread*().getName();

System.***out***.println("Foo " + name);

TimeUnit.***SECONDS***.sleep(1);

System.***out***.println("Bar " + name);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

};

Thread thread = **new** Thread(runnable);

thread.start();

Outputs: Foo Thread-0

Bar Thread-0

Example:

ExecutorService executor = Executors.*newSingleThreadExecutor*();

executor.submit(() -> {

String threadName = Thread.*currentThread*().getName();

System.***out***.println("Hello " + threadName);

});

# Tools

## jdeps

Java class dependency analyzer.

Synopsis

**jdeps** [*options*] *classes* ...

***options***

Command-line options. See [Options](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html" \l "BACCCJEH).

***classes***

Name of the classes to analyze. You can specify a class that can be found in the class path, by its file name, a directory, or a JAR file.

Description

The jdeps command shows the package-level or class-level dependencies of Java class files. The input class can be a path name to a .class file, a directory, a JAR file, or it can be a fully qualified class name to analyze all class files. The options determine the output. By default, jdeps outputs the dependencies to the system output. It can generate the dependencies in DOT language (see the -dotoutput option).

# Nashorn, JavaScript Engine

The Nashorn javascript engine can either be used programmatically from java programs or by utilizing the command line tool jjs, which is located in $JAVA\_HOME/bin. If you plan to work with jjs you might want to put a symbolic link for simple access.

Example:

ScriptEngine engine = **new** ScriptEngineManager().getEngineByName("nashorn");

**try** {

engine.eval("print('Hello World!');");

} **catch** (ScriptException e) {

e.printStackTrace();

}

Example:

**try** {

engine.eval (**new** FileReader ("script.js"));

} **catch** (FileNotFoundException | ScriptException e) {

e.printStackTrace();

}

Example:

//get handle of JavaScript file’s context

Invocable invocable = (Invocable) engine;

//invoke function defined in the file’s context and pass the argument.

Object result;

**try** {

result = invocable.invokeFunction ("fun1", "Mr. X");

} **catch** (NoSuchMethodException e) {

e.printStackTrace();

} **catch** (ScriptException e) {

e.printStackTrace();

}

//print the result and its type.

System.***out***.println (result);

System.***out***.println (result.getClass ());