# Java 8 Features

Oracle released a new version of Java as Java 8 in March 18, 2014. It was a revolutionary release of the Java for software development platform. It includes various upgrades to the Java programming, JVM, Tools and libraries.

## Java 8 Programming Language Enhancements

Java 8 provides following features for Java Programming:

* Lambda expressions,
* Method references,
* Functional interfaces,
* Stream API,
* Default methods,
* Base64 Encode Decode,
* Static methods in interface,
* Optional class,
* Collectors class,
* ForEach() method,
* Parallel array sorting,
* Nashorn JavaScript Engine,
* Parallel Array Sorting,
* Type and Repating Annotations,
* IO Enhancements,
* Concurrency Enhancements,
* JDBC Enhancements etc.

## Lambda Expressions

Lambda expression helps us to write our code in functional style. It provides a clear and concise way to implement SAM interface(Single Abstract Method) by using an expression. It is very useful in collection library in which it helps to iterate, filter and extract data.

For more information and examples: [click here](http://www.javatpoint.com/java-lambda-expressions)

## Method References

Java 8 Method reference is used to refer method of functional interface . It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

For more information and examples: [click here](https://www.javatpoint.com/java-8-method-reference)

## Functional Interface

An Interface that contains only one abstract method is known as functional interface. It can have any number of default and static methods. It can also declare methods of object class.

Functional interfaces are also known as Single Abstract Method Interfaces (SAM Interfaces).

For more information and examples: [click here](http://www.javatpoint.com/java-8-functional-interfaces)

## Optional

Java introduced a new class Optional in Java 8. It is a public final class which is used to deal with NullPointerException in Java application. We must import java.util package to use this class. It provides methods to check the presence of value for particular variable.

For more information and examples: [click here](http://www.javatpoint.com/java-8-optional)

## forEach

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interfaces.

It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach() method to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

For more information and examples: [click here](https://www.javatpoint.com/java-8-features)

## Date/Time API

Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes.

For more information and examples: [click here](http://www.javatpoint.com/java-date)

## Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default keyword are known as default methods. These methods are non-abstract methods and can have method body.

For more information and examples: [click here](http://www.javatpoint.com/java-default-methods)

## Nashorn JavaScript Engine

Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool **jjs** which is used to execute JavaScript code.

You can execute JavaScript code by two ways:

1. Using jjs command-line tool, and
2. By embedding into Java source code.

For more information and examples: [click here](https://www.javatpoint.com/java-nashorn)

## StringJoiner

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc.

For more information and examples: [click here](https://www.javatpoint.com/java-stringjoiner)

## Collectors

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria etc.

For more information and examples: [click here](https://www.javatpoint.com/java-8-collectors)

## Stream API

Java 8 java.util.stream package consists of classes, interfaces and an enum to allow functional-style operations on the elements. It performs lazy computation. So, it executes only when it requires.

For more information and examples: [click here](http://www.javatpoint.com/java-8-stream)

## Stream Filter

Java stream provides a method filter() to filter stream elements on the basis of given predicate. Suppose, you want to get only even elements of your list, you can do this easily with the help of filter() method.

This method takes predicate as an argument and returns a stream of resulted elements.

For more information and examples: [click here](https://www.javatpoint.com/java-8-stream-filter)

## Java Base64 Encoding and Decoding

Java provides a class Base64 to deal with encryption and decryption. You need to import java.util.Base64 class in your source file to use its methods.

This class provides three different encoders and decoders to encrypt information at each level.

For more information and examples: [click here](https://www.javatpoint.com/java-base64-encode-decode)

## Java Parallel Array Sorting

Java provides a new additional feature in Arrays class which is used to sort array elements parallelly. The parallelSort() method has added to java.util.Arrays class that uses the JSR 166 Fork/Join parallelism common pool to provide sorting of arrays. It is an overloaded method.

For more information and examples: [click here](https://www.javatpoint.com/java-8-parallel-array-sorting)

## Java 8 Security Enhancements

1) The Java Secure Socket Extension(JSSE) provider enables the protocols Transport Layer Security (TLS) 1.1 and TLS 1.2 by default on the client side.

2) A improved method AccessController.doPrivileged has been added which enables code to assert a subset of its privileges, without preventing the full traversal of the stack to check for other permissions.

3) Advanced Encryption Standard (AES) and Password-Based Encryption (PBE) algorithms, such as PBEWithSHA256AndAES\_128 and PBEWithSHA512AndAES\_256 has been added to the SunJCE provider.

4) Java Secure Socket Extension (SunJSSE) has enabled Server Name Indication (SNI) extension for client applications by default in JDK 7 and JDK 8 supports the SNI extension for server applications. The SNI extension is a feature that extends the SSL/TLS protocols to indicate what server name the client is attempting to connect to during handshaking.

5) The SunJSSE is enhanced to support Authenticated Encryption with Associated Data (AEAD) algorithms. The Java Cryptography Extension (SunJCE) provider is enhanced to support AES/GCM/NoPadding cipher implementation as well as Galois/Counter Mode (GCM) algorithm parameters.

6) A new command flag -importpassword is added to the keytool utility. It is used to accept a password and store it securely as a secret key. Classes such as java.security.DomainLoadStoreParameter andjava.security.PKCS12Attribute is added to support DKS keystore type.

7) In JDK 8, the cryptographic algorithms have been enhanced with the SHA-224 variant of the SHA-2 family of message-digest implementations.

8) Enhanced support for NSA Suite B Cryptography which includes:

* OID registration for NSA Suite B cryptography algorithms
* Support for 2048-bit DSA key pair generation and additional signature algorithms for 2048-bit DSA keys such as SHA224withDSA and SHA256withDSA.
* Lifting of the keysize restriction from 1024 to 2048 for Diffie-Hellman (DH) algorithm.

9) SecureRandom class provides the generation of cryptographically strong random numbers which is used for private or public keys, ciphers and signed messages. The getInstanceStrong() method was introduced in JDK 8, which returns an instance of the strongest SecureRandom. It should be used when you need to create RSA private and public key. SecureRandom includes following other changes:

* Two new implementations has introduced for UNIX platforms, which provide blocking and non-blocking behavior.

10) A new PKIXRevocationChecker class is included which checks the revocation status of certificates with the PKIX algorithm. It supports best effort checking, end-entity certificate checking, and mechanism-specific options.

11) The Public Key Cryptography Standards 11 (PKCS) has been expanded to include 64-bit supports for Windows.

12) Two new rcache types are added to Kerberos 5. Type none means no rcache at all, and type dfl means the DFL style file-based rcache. Also, the acceptor requested subkey is now supported. They are configured using the sun.security.krb5.rcache and sun.security.krb5.acceptor.subkey system properties.

13) In JDK 8, Kerberos 5 protocol transition and constrained delegation are supported within the same realm.

14) Java 8 has disabled weak encryption by default. The DES-related Kerberos 5 encryption types are not supported by default. These encryption types can be enabled by adding allow\_weak\_crypto=true in the krb5.conf file.

15) You can set server name to null to denote an unbound server. It means a client can request for the service using any server name. After a context is established, the server can retrieve the name as a negotiated property with the key name SASL.BOUND\_SERVER\_NAME.

16) Java Native Interface (JNI) bridge to native Java Generic Security Service (JGSS) is now supported on Mac OS X. You can set system property sun.security.jgss.native to true to enable it.

17) A new system property, jdk.tls.ephemeralDHKeySize is defined to customize the ephemeral DH key sizes. The minimum acceptable DH key size is 1024 bits, except for exportable cipher suites or legacy mode (jdk.tls.ephemeralDHKeySize=legacy).

18) Java Secure Socket Extension (JSSE) provider honors the client's cipher suite preference by default. However, the behavior can be changed to respect the server's cipher suite preference by calling SSLParameters.setUseCipherSuitesOrder(true) over the server.

## Java 8 Tools Enhancements

1) A jjs command is introduced, which invokes the Nashorn engine either in interactive shell mode, or to interpret script files.

2) The java command is capable of launching JavaFX applications, provided that the JavaFX application is packaged correctly.

3) The java command man page (both nroff and HTML) has been completely reworked. The advanced options are now divided into Runtime, Compiler, Garbage Collection, and Serviceability, according to the area that they affect. Several previously missing options are now described. There is also a section for options that were deprecated or removed since the previous release.

4) New jdeps command-line tool allows the developer to analyze class files to determine package-level or class-level dependencies.

5) You can access diagnostic commands remotely, which were previously accessible only locally via the jcmd tool. Remote access is provided using the Java Management Extensions (JMX), so diagnostic commands are exposed to a platform MBean registered to the platform MBean server. The MBean is the com.sun.management.DiagnosticCommandMBean interface.

6) A new option -tsapolicyid is included in the jarsigner tool which enables you to request a signed time stamp from a Time Stamping Authority and attach it to a signed JAR file.

7) A new method java.lang.reflect.Executable.getParameters is included which allows you to access the names of the formal parameters of any method or constructor. However, .class files do not store formal parameter names by default. To store formal parameter names in a particular .class file, and thus enable the Reflection API to retrieve formal parameter names, compile the source file with the -parameters option of the javac compiler.

8) The type rules for binary comparisons in the Java Language Specification (JLS) Section 15.21 will now be correctly enforced by javac.

9) In this release, the apt tool and its associated API contained in the package com.sun.mirror have been removed.

## Javadoc Enhancements

In Java SE 8, the following new APIs were added to the Javadoc tool.

* A new DocTree API introduce a scanner which enables you to traverse source code that is represented by an abstract syntax tree. This extends the Compiler Tree API to provide structured access to the content of javadoc comments.
* The javax.tools package contains classes and interfaces that enable you to invoke the Javadoc tool directly from a Java application, without executing a new process.
* The "Method Summary" section of the generated documentation of a class or interface has been restructured. Method descriptions in this section are grouped by type. By default, all methods are listed. You can click a tab to view methods of a particular type (static, instance, abstract, concrete, or deprecated, if they exist in the class or interface).
* The javadoc tool now has support for checking the content of javadoc comments for issues that could lead to various problems, such as invalid HTML or accessibility issues, in the files that are generated by javadoc. The feature is enabled by default, and can also be controlled by the new -Xdoclint option.

## Pack200 Enhancements

The Java class file format has been updated because of JSR 292 which Supports Dynamically Typed Languages on the Java Platform.

The Pack200 engine has been updated to ensure that Java SE 8 class files are compressed effectively. Now, it can recognize constant pool entries and new bytecodes introduced by JSR 292. As a result, compressed files created with this version of the pack200 tool will not be compatible with older versions of the unpack200 tool.

## Java 8 I/O Enhancements

In Java 8, there are several improvements to the java.nio.charset.Charset and extended charset implementations. It includes the following:

* A New SelectorProvider which may improve performance or scalability for server. The /dev/poll SelectorProvider continues to be the default. To use the Solaris event port mechanism, run with the system property java.nio.channels.spi.Selector set to the value sun.nio.ch.EventPortSelectorProvider.
* The size of <JDK\_HOME>/jre/lib/charsets.jar file is decreased.
* Performance has been improvement for the java.lang.String(byte[], ∗) constructor and the java.lang.String.getBytes() method.

## Java 8 Networking Enhancements

1) A new class java.net.URLPermission has been added. It represents a permission for accessing a resource defined by a given URL.

2) A package jdk.net has been added which contains platform specific socket options and a mechanism for setting these options on all of the standard socket types. The socket options are defined in jdk.net.ExtendedSocketOptions.

3) In class HttpURLConnection, if a security manager is installed, and if a method is called which results in an attempt to open a connection, the caller must possess either a "connect"SocketPermission to the host/port combination of the destination URL or a URLPermission that permits this request.

If automatic redirection is enabled, and this request is redirected to another destination, the caller must also have permission to connect to the redirected host/URL.

## Java 8 Concurrency Enhancements

The java.util.concurrent package added two new interfaces and four new classes.

### Java.util.concurrent Interfaces

|  |  |
| --- | --- |
| **Interface** | **Description** |
| public static interface CompletableFuture.AsynchronousCompletionTask | It is a marker interface which is used to identify asynchronous tasks produced by async methods. It may be useful for monitoring, debugging, and tracking asynchronous activities. |
| public interface CompletionStage<T> | It creates a stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes. |

### Java.util.concurrent Classes

|  |  |
| --- | --- |
| **Class** | **Description** |
| public class CompletableFuture<T> extends Object implements Future<T>, CompletionStage<T> | It is aFuture that may be explicitly completed, and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion. |
| public static class ConcurrentHashMap.KeySetView<K,V> extends Object implements Set<K>, Serializable | It is a view of a ConcurrentHashMap as a Set of keys, in which additions may optionally be enabled by mapping to a common value. |
| public abstract class CountedCompleter<T> extends ForkJoinTask<T> | A ForkJoinTask with a completion action performed when triggered and there are no remaining pending actions. |
| public class CompletionException extends RuntimeException | It throws an exception when an error or other exception is encountered in the course of completing a result or task. |

#### New Methods in java.util.concurrent.ConcurrentHashMap class

ConcurrentHashMap class introduces several new methods in its latest release. It includes 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.

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

Latest release introduces scalable, updatable, variable support through a small set of new classes DoubleAccumulator, DoubleAdder, LongAccumulator andLongAdder. It internally employ contention-reduction techniques that provide huge throughput improvements as compared to Atomic variables.

|  |  |
| --- | --- |
| **Class** | **Description** |
| public class DoubleAccumulator extends Number implements Serializable | It is used for one or more variables that together maintain a running double value updated using a supplied function. |
| public class DoubleAdder extends Number implements Serializable | It is used for one or more variables that together maintain an initially zero double sum. |
| public class LongAccumulator extends Number implements Serializable | It is used for one or more variables that together maintain a running long value updated using a supplied function. |
| public class LongAdder extends Number implements Serializable | It is used for one or more variables that together maintain an initially zero long sum. |

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

This class has added two new methods getCommonPoolParallelism() and commonPool(), which return the targeted parallelism level of the common pool, or the common pool instance, respectively.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static ForkJoinPool commonPool() | It returns the common pool instance. |
| Public static int getCommonPoolParallelism() | It returns the targeted parallelism level of the common pool. |

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

A new class StampedLock is added which is used to add 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.

|  |  |
| --- | --- |
| **Class** | **Description** |
| public class StampedLock extends Object implements Serializable | This class represents a capability-based lock with three modes for controlling read/write access. |

## Java API for XML Processing (JAXP) 1.6 Enhancements

In Java 8, Java API is added for XML Processing (JAXP) 1.6. It requires the use of the service provider loader facility which is defined by java.util.ServiceLoader to load services from service configuration files.

The rationale for this is to allow for future modularization of the Java SE platform where service providers may be deployed by means other than JAR files and perhaps without the service configuration files.

## Java Virtual Machine Enhancements

The verification of invokespecial instructions has been tightened so that only an instance initialization method in the current class or its direct super class may be invoked.

## Java Mission Control 5.3 is included in Java 8

Java Mission Control (JMC) is an advanced set of tools that enables efficient and detailed data analysis and delivers advanced, unobtrusive Java monitoring and management. JMC provides sections for common analysis areas such as code performance, memory and latency.

Babel Language Packs in Japanese and Simplified Chinese are now included by default in the Java Mission Control that is included in the JDK 8.

## Java 8 Internationalization Enhancements

### 1) Unicode Enhancements

The JDK 8 includes support for Unicode 6.2.0. It contains the following features.

* 733 new characters including Turkish Lira sign.
* 7 new scripts:
  + Meroitic Hieroglyphs
  + Meroitic Cursive
  + Sora Sompeng
  + Chakma
  + Sharada
  + Takri
  + Miao
* 11 new blocks: including 7 blocks for the new scripts listed above and 4 blocks for the following existing scripts:
* Arabic Extended-A
* Sundanese Supplement
* Meetei Mayek Extensions
* Arabic Mathematical Alphabetical Symbols

### Adoption of Unicode CLDR Data and the java.locale.providers System Property

The Unicode Consortium has released the Common Locale Data Repository (CLDR) project to "support the world's languages, with the largest and most extensive standard repository of locale data available." The CLDR is becoming the de-facto standard for locale data. The CLDR's XML-based locale data has been incorporated into the JDK 8 release, however it is disabled by default.

There are four distinct sources for locale data:

* CLDR represents the locale data provided by the Unicode CLDR project.
* HOST represents the current user's customization of the underlying operating system's settings. It works only with the user's default locale, and the customizable settings may vary depending on the OS, but primarily Date, Time, Number, and Currency formats are supported.
* SPI represents the locale sensitive services implemented in the installed SPI providers.
* JRE represents the locale data that is compatible with the prior JRE releases.

To select the desired locale data source, use the java.locale.providers system property. listing the data sources in the preferred order. For example: java.locale.providers=HOST,SPI,CLDR,JRE The default behavior is equivalent to the following setting: java.locale.providers=JRE,SPI

## Java 8 New Calendar and Locale APIs

The JDK 8 includes two new classes, several new methods, and a new return value for an existing static method.

Two new abstract classes for service providers are added to the java.util.spi package.

|  |  |
| --- | --- |
| **Class** | **Description** |
| public abstract class CalendarDataProvider extends LocaleServiceProvider | It is an abstract class for service providers that provide locale-dependent Calendar parameters. |
| public abstract class CalendarNameProvider extends LocaleServiceProvider | It is an abstract class for service providers that provide localized string representations (display names) of Calendar field values. |

A static method is now able to recognize Locale.UNICODE\_LOCALE\_EXTENSION for the numbering system.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static final DecimalFormatSymbols getInstance(Locale locale) | It is used to get the DecimalFormatSymbols instance for the specified locale. This method provides access to DecimalFormatSymbols instances for locales supported by the Java runtime itself as well as for those supported by installed DecimalFormatSymbolsProvider implementations. It throws NullPointerException if locale is null. |

Added New methods in calender API:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public boolean isSupportedLocale(Locale locale) | It returns true if the given locale is supported by this locale service provider. The given locale may contain extensions that should be taken into account for the support determination. It is define in java.util.spi.LocaleServiceProvider class |
| public String getCalendarType() | It returns the calendar type of this Calendar. Calendar types are defined by the Unicode Locale Data Markup Language (LDML) specification. It is defined in java.util.Calendar class. |

New style specifiers are added for the Calendar.getDisplayName and Calendar.getDisplayNames methods to determine the format of the Calendar name.

|  |  |
| --- | --- |
| **Specifier** | **Description** |
| public static final int SHORT\_FORMAT | It is a style specifier for getDisplayName and getDisplayNames indicating a short name used for format. |
| public static final int LONG\_FORMAT | It is a style specifier for getDisplayName and getDisplayNames indicating a long name used for format. |
| public static final int SHORT\_STANDALONE | It is a style specifier for getDisplayName and getDisplayNames indicating a short name used independently, such as a month abbreviation as calendar headers. |
| public static final int LONG\_STANDALONE | It is a style specifier for getDisplayName and getDisplayNames indicating a long name used independently, such as a month name as calendar headers. |

Two new Locale methods for dealing with a locale's (optional) extensions.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public boolean hasExtensions() | It returns true if this Locale has any extensions. |
| public Locale stripExtensions() | It returns a copy of this Locale with no extensions. If this Locale has no extensions, this Locale is returned itself. |

Two new Locale.filter methods return a list of Locale instances that match the specified criteria, as defined in RFC 4647:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static List<Locale> filter(List<Locale.LanguageRange> priorityList,Collection<Locale> locales) | It returns a list of matching Locale instances using the filtering mechanism defined in RFC 4647. This is equivalent to filter(List, Collection, FilteringMode) when mode is Locale.FilteringMode.AUTOSELECT\_FILTERING. |
| public static List<Locale> filter(List<Locale.LanguageRange> priorityList,Collection<Locale> locales, Locale.FilteringMode mode) | It returns a list of matching Locale instances using the filtering mechanism defined in RFC 4647. |

Two new Locale.filterTags methods return a list of language tags that match the specified criteria, as defined in RFC 4647.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static List<String> filterTags(List<Locale.LanguageRange> priorityList, Collection<String> tags) | It returns a list of matching languages tags using the basic filtering mechanism defined in RFC 4647. This is equivalent to filterTags(List, Collection, FilteringMode) when mode is Locale.FilteringMode.AUTOSELECT\_FILTERING. |
| public static List<String> filterTags(List<Locale.LanguageRange> priorityList, Collection<String> tags, Locale.FilteringMode mode) | It returns a list of matching languages tags using the basic filtering mechanism defined in RFC 4647. |

Two new lookup methods return the best-matching locale or language tag using the lookup mechanism defined in RFC 4647.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static Locale lookup(List<Locale.LanguageRange> priorityList, Collection<Locale> locales) | It returns a Locale instance for the best-matching language tag using the lookup mechanism defined in RFC 4647. |
| Public static String lookupTag(List<Locale.LanguageRange> priorityList,Collection<String> tags) | It returns the best-matching language tag using the lookup mechanism defined in RFC 4647. |

## Other Java 8 Version Enhancements

## Enhancements in JDK 8u5

1) The frequency in which the security prompts are shown for an application has been reduced.

## Enhancements in JDK 8u11

1) An option to suppress offers from sponsors when the JRE is installed or updated is available in the Advanced tab of the Java Control Panel.

2) The Entry-Point attribute can be included in the JAR file manifest to identify one or more classes as a valid entry point for your RIA(Rich Internet application).

## Enhancements in JDK 8u20

1) The javafxpackager tool has been renamed to javapackager. This tool has been enhanced with new arguments for self-contained application bundlers.

Follwing enhancements are related to the java tool:

* An experimental JIT compiler option related to Restricted Transactional Memory (RTM) has been added.
* Several options related to string deduplication have been added.
* Several options related to Advanced Encryption Standard (AES) intrinsics have been added.
* Combinations of garbage collection options have been deprecated.

2) Garbage Collection Tuning Guide has been added to the Java HotSpot Virtual Machine. It describes the garbage collectors included with the Java HotSpot VM and helps you to decide which garbage collector can best optimize the performance of your application, especially if it handles large amounts of data (multiple gigabytes), has many threads, and has high transaction rates.

## Enhancements in JDK 8u31

1) In this release, the SSLv3 protocol is removed from the Java Control Panel Advanced options.

## Enhancements in JDK 8u40

### Java tool

1) The -XX:+CheckEndorsedAndExtDirs has been added because the endorsed-standards override mechanism (JDK-8065675) and the extension mechanism (JDK-8065702) have been deprecated. The option helps identify any existing uses of these mechanisms and is supported in JDK 7u80 and JDK 8u40.

2) Java Flight Recorder (JFR) offers a variety of ways to unlock commercial features and enable JFR during the runtime of an application.

It includes java command line options such as jcmd diagnostic commands and Graphical User Interface (GUI) controls within Java Mission Control. This flexibility enables you to provide the appropriate options at startup, or interact with JFR later.

3) The option -XX:StartFlightRecording=parameter=value has a new parameter, dumponexit={true|false}, which specifies whether a dump file of JFR data should be generated when the JVM terminates in a controlled manner.

4) The options related to Restricted Transactional Memory (RTM) are no longer experimental. These options include -XX:RTMAbortRatio=abort\_ratio, -XX:RTMRetryCount=number\_of\_retries, -XX:+UseRTMDeopt, and -XX:+UseRTMLocking.

5) In Java 8, Application Class Data Sharing (AppCDS) has been introduced. AppCDS extends CDS (Class Data Sharing) to enable classes from the standard extensions directories and the application class path to be placed in the shared archive. This is a commercial feature and is no longer considered experimental.

6) New options -XX:+ResourceManagement and -XX:ResourceManagementSampleInterval=value have been added.

7) Additional information about large pages has been added. Large Pages, also known as huge pages, are memory pages that are significantly larger than the standard memory page size. Large pages optimize processor Translation-Lookaside Buffers. The Linux options -XX:+UseHugeTLBFS, -XX:+UseSHM, and -XX:+UseTransparentHugePages have been documented.

8) The option -XX:ObjectAlignmentInBytes=alignment has been documented.

### JJS tool

1) The option --optimistic-types=[true|false] has been added. It enables or disables optimistic type assumptions with deoptimizing recompilation.

2) The option --language=[es5] has been added to the jjs tool. It specifies the ECMAScript language version.

### Javapackager tool

1) New arguments are available for OS X bundlers. The mac.CFBundleVersion argument identifies the internal version number to be used.

2) The mac.dmg.simple argument indicates if DMG customization steps that depend on executing AppleScript code are skipped.

### Jcmd tool

Jcmd tool is used to dynamically interact with Java Flight Recorder (JFR). You can use it to unlock commercial features, enable/start/stop flight recordings, and obtain various status messages from the system.

### Jstat tool

The jstat tool has been updated with information about compressed class space which is a special part of metaspace.

### Virtual machine

The Scalable Native Memory Tracking HotSpot VM feature helps diagnose VM memory leaks and clarify users when memory leaks are not in the VM. Native Memory Tracker can be run without self-shutdown on large systems and without causing a significant performance impact beyond what is considered acceptable for small programs.

# Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

## Functional Interface

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

## Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

## Java Lambda Expression Syntax

1. (argument-list) -> {body}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

**No Parameter Syntax**

1. () -> {
2. //Body of no parameter lambda
3. }

**One Parameter Syntax**

1. (p1) -> {
2. //Body of single parameter lambda
3. }

**Two Parameter Syntax**

1. (p1,p2) -> {
2. //Body of multiple parameter lambda
3. }

Let's see a scenario where we are not implementing Java lambda expression. Here, we are implementing an interface without using lambda expression.

## Without Lambda Expression

1. **interface** Drawable{
2. **public** **void** draw();
3. }
4. **public** **class** LambdaExpressionExample {
5. **public** **static** **void** main(String[] args) {
6. **int** width=10;
8. //without lambda, Drawable implementation using anonymous class
9. Drawable d=**new** Drawable(){
10. **public** **void** draw(){System.out.println("Drawing "+width);}
11. };
12. d.draw();
13. }
14. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample)

Output:

Drawing 10

## Java Lambda Expression Example

Now, we are going to implement the above example with the help of Java lambda expression.

1. @FunctionalInterface  //It is optional
2. **interface** Drawable{
3. **public** **void** draw();
4. }
6. **public** **class** LambdaExpressionExample2 {
7. **public** **static** **void** main(String[] args) {
8. **int** width=10;
10. //with lambda
11. Drawable d2=()->{
12. System.out.println("Drawing "+width);
13. };
14. d2.draw();
15. }
16. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample2)

Output:

Drawing 10

A lambda expression can have zero or any number of arguments. Let's see the examples:

## Java Lambda Expression Example: No Parameter

1. **interface** Sayable{
2. **public** String say();
3. }
4. **public** **class** LambdaExpressionExample3{
5. **public** **static** **void** main(String[] args) {
6. Sayable s=()->{
7. **return** "I have nothing to say.";
8. };
9. System.out.println(s.say());
10. }
11. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample3)

Output:

I have nothing to say.

## Java Lambda Expression Example: Single Parameter

1. **interface** Sayable{
2. **public** String say(String name);
3. }
5. **public** **class** LambdaExpressionExample4{
6. **public** **static** **void** main(String[] args) {
8. // Lambda expression with single parameter.
9. Sayable s1=(name)->{
10. **return** "Hello, "+name;
11. };
12. System.out.println(s1.say("Sonoo"));
14. // You can omit function parentheses
15. Sayable s2= name ->{
16. **return** "Hello, "+name;
17. };
18. System.out.println(s2.say("Sonoo"));
19. }
20. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample4)

Output:

Hello, Sonoo

Hello, Sonoo

## Java Lambda Expression Example: Multiple Parameters

1. **interface** Addable{
2. **int** add(**int** a,**int** b);
3. }
5. **public** **class** LambdaExpressionExample5{
6. **public** **static** **void** main(String[] args) {
8. // Multiple parameters in lambda expression
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Multiple parameters with data type in lambda expression
13. Addable ad2=(**int** a,**int** b)->(a+b);
14. System.out.println(ad2.add(100,200));
15. }
16. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample5)

Output:

30

300

## Java Lambda Expression Example: with or without return keyword

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

1. **interface** Addable{
2. **int** add(**int** a,**int** b);
3. }
5. **public** **class** LambdaExpressionExample6 {
6. **public** **static** **void** main(String[] args) {
8. // Lambda expression without return keyword.
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Lambda expression with return keyword.
13. Addable ad2=(**int** a,**int** b)->{
14. **return** (a+b);
15. };
16. System.out.println(ad2.add(100,200));
17. }
18. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample6)

Output:

30

300

## Java Lambda Expression Example: Foreach Loop

1. **import** java.util.\*;
2. **public** **class** LambdaExpressionExample7{
3. **public** **static** **void** main(String[] args) {
5. List<String> list=**new** ArrayList<String>();
6. list.add("ankit");
7. list.add("mayank");
8. list.add("irfan");
9. list.add("jai");
11. list.forEach(
12. (n)->System.out.println(n)
13. );
14. }
15. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample7)

Output:

ankit

mayank

irfan

jai

## Java Lambda Expression Example: Multiple Statements

1. @FunctionalInterface
2. **interface** Sayable{
3. String say(String message);
4. }
6. **public** **class** LambdaExpressionExample8{
7. **public** **static** **void** main(String[] args) {
9. // You can pass multiple statements in lambda expression
10. Sayable person = (message)-> {
11. String str1 = "I would like to say, ";
12. String str2 = str1 + message;
13. **return** str2;
14. };
15. System.out.println(person.say("time is precious."));
16. }
17. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample8)

Output:

I would like to say, time is precious.

## Java Lambda Expression Example: Creating Thread

You can use lambda expression to run thread. In the following example, we are implementing run method by using lambda expression.

1. **public** **class** LambdaExpressionExample9{
2. **public** **static** **void** main(String[] args) {
4. //Thread Example without lambda
5. Runnable r1=**new** Runnable(){
6. **public** **void** run(){
7. System.out.println("Thread1 is running...");
8. }
9. };
10. Thread t1=**new** Thread(r1);
11. t1.start();
12. //Thread Example with lambda
13. Runnable r2=()->{
14. System.out.println("Thread2 is running...");
15. };
16. Thread t2=**new** Thread(r2);
17. t2.start();
18. }
19. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample9)

Output:

Thread1 is running...

Thread2 is running...

Java lambda expression can be used in the collection framework. It provides efficient and concise way to iterate, filter and fetch data. Following are some lambda and collection examples provided.

## Java Lambda Expression Example: Comparator

1. **import** java.util.ArrayList;
2. **import** java.util.Collections;
3. **import** java.util.List;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
8. **public** Product(**int** id, String name, **float** price) {
9. **super**();
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** LambdaExpressionExample10{
16. **public** **static** **void** main(String[] args) {
17. List<Product> list=**new** ArrayList<Product>();
19. //Adding Products
20. list.add(**new** Product(1,"HP Laptop",25000f));
21. list.add(**new** Product(3,"Keyboard",300f));
22. list.add(**new** Product(2,"Dell Mouse",150f));
24. System.out.println("Sorting on the basis of name...");
26. // implementing lambda expression
27. Collections.sort(list,(p1,p2)->{
28. **return** p1.name.compareTo(p2.name);
29. });
30. **for**(Product p:list){
31. System.out.println(p.id+" "+p.name+" "+p.price);
32. }
34. }
35. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample10)

Output:

Sorting on the basis of name...

2 Dell Mouse 150.0

1 HP Laptop 25000.0

3 Keyboard 300.0

## Java Lambda Expression Example: Filter Collection Data

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **import** java.util.stream.Stream;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
8. **public** Product(**int** id, String name, **float** price) {
9. **super**();
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** LambdaExpressionExample11{
16. **public** **static** **void** main(String[] args) {
17. List<Product> list=**new** ArrayList<Product>();
18. list.add(**new** Product(1,"Samsung A5",17000f));
19. list.add(**new** Product(3,"Iphone 6S",65000f));
20. list.add(**new** Product(2,"Sony Xperia",25000f));
21. list.add(**new** Product(4,"Nokia Lumia",15000f));
22. list.add(**new** Product(5,"Redmi4 ",26000f));
23. list.add(**new** Product(6,"Lenevo Vibe",19000f));
25. // using lambda to filter data
26. Stream<Product> filtered\_data = list.stream().filter(p -> p.price > 20000);
28. // using lambda to iterate through collection
29. filtered\_data.forEach(
30. product -> System.out.println(product.name+": "+product.price)
31. );
32. }
33. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample11)

Output:

Iphone 6S: 65000.0

Sony Xperia: 25000.0

Redmi4 : 26000.0

## Java Lambda Expression Example: Event Listener

1. **import** javax.swing.JButton;
2. **import** javax.swing.JFrame;
3. **import** javax.swing.JTextField;
4. **public** **class** LambdaEventListenerExample {
5. **public** **static** **void** main(String[] args) {
6. JTextField tf=**new** JTextField();
7. tf.setBounds(50, 50,150,20);
8. JButton b=**new** JButton("click");
9. b.setBounds(80,100,70,30);
11. // lambda expression implementing here.
12. b.addActionListener(e-> {tf.setText("hello swing");});
14. JFrame f=**new** JFrame();
15. f.add(tf);f.add(b);
16. f.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);
17. f.setLayout(**null**);
18. f.setSize(300, 200);
19. f.setVisible(**true**);
21. }
23. }

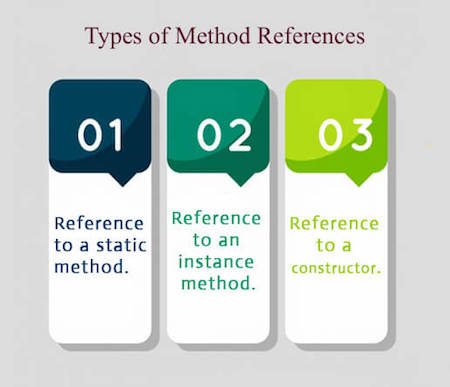
# Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

## Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.



## 1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

1. ContainingClass::staticMethodName

### Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

1. **interface** Sayable{
2. **void** say();
3. }
4. **public** **class** MethodReference {
5. **public** **static** **void** saySomething(){
6. System.out.println("Hello, this is static method.");
7. }
8. **public** **static** **void** main(String[] args) {
9. // Referring static method
10. Sayable sayable = MethodReference::saySomething;
11. // Calling interface method
12. sayable.say();
13. }
14. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=MethodReference)

Output:

Hello, this is static method.

### Example 2

In the following example, we are using predefined functional interface Runnable to refer static method.

1. **public** **class** MethodReference2 {
2. **public** **static** **void** ThreadStatus(){
3. System.out.println("Thread is running...");
4. }
5. **public** **static** **void** main(String[] args) {
6. Thread t2=**new** Thread(MethodReference2::ThreadStatus);
7. t2.start();
8. }
9. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=MethodReference2)

Output:

Thread is running...

### Example 3

You can also use predefined functional interface to refer methods. In the following example, we are using BiFunction interface and using it's apply() method.

1. **import** java.util.function.BiFunction;
2. **class** Arithmetic{
3. **public** **static** **int** add(**int** a, **int** b){
4. **return** a+b;
5. }
6. }
7. **public** **class** MethodReference3 {
8. **public** **static** **void** main(String[] args) {
9. BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;
10. **int** result = adder.apply(10, 20);
11. System.out.println(result);
12. }
13. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=MethodReference3)

Output:

30

### Example 4

You can also override static methods by referring methods. In the following example, we have defined and overloaded three add methods.

1. **import** java.util.function.BiFunction;
2. **class** Arithmetic{
3. **public** **static** **int** add(**int** a, **int** b){
4. **return** a+b;
5. }
6. **public** **static** **float** add(**int** a, **float** b){
7. **return** a+b;
8. }
9. **public** **static** **float** add(**float** a, **float** b){
10. **return** a+b;
11. }
12. }
13. **public** **class** MethodReference4 {
14. **public** **static** **void** main(String[] args) {
15. BiFunction<Integer, Integer, Integer>adder1 = Arithmetic::add;
16. BiFunction<Integer, Float, Float>adder2 = Arithmetic::add;
17. BiFunction<Float, Float, Float>adder3 = Arithmetic::add;
18. **int** result1 = adder1.apply(10, 20);
19. **float** result2 = adder2.apply(10, 20.0f);
20. **float** result3 = adder3.apply(10.0f, 20.0f);
21. System.out.println(result1);
22. System.out.println(result2);
23. System.out.println(result3);
24. }
25. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=MethodReference4)

Output:

30

30.0

30.0

## 2) Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

1. containingObject::instanceMethodName

### Example 1

In the following example, we are referring non-static methods. You can refer methods by class object and anonymous object.

1. **interface** Sayable{
2. **void** say();
3. }
4. **public** **class** InstanceMethodReference {
5. **public** **void** saySomething(){
6. System.out.println("Hello, this is non-static method.");
7. }
8. **public** **static** **void** main(String[] args) {
9. InstanceMethodReference methodReference = **new** InstanceMethodReference(); // Creating object
10. // Referring non-static method using reference
11. Sayable sayable = methodReference::saySomething;
12. // Calling interface method
13. sayable.say();
14. // Referring non-static method using anonymous object
15. Sayable sayable2 = **new** InstanceMethodReference()::saySomething; // You can use anonymous object also
16. // Calling interface method
17. sayable2.say();
18. }
19. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=InstanceMethodReference)

Output:

Hello, this is non-static method.

Hello, this is non-static method.

### Example 2

In the following example, we are referring instance (non-static) method. Runnable interface contains only one abstract method. So, we can use it as functional interface.

1. **public** **class** InstanceMethodReference2 {
2. **public** **void** printnMsg(){
3. System.out.println("Hello, this is instance method");
4. }
5. **public** **static** **void** main(String[] args) {
6. Thread t2=**new** Thread(**new** InstanceMethodReference2()::printnMsg);
7. t2.start();
8. }
9. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=InstanceMethodReference2)

Output:

Hello, this is instance method

### Example 3

In the following example, we are using BiFunction interface. It is a predefined interface and contains a functional method apply(). Here, we are referring add method to apply method.

1. **import** java.util.function.BiFunction;
2. **class** Arithmetic{
3. **public** **int** add(**int** a, **int** b){
4. **return** a+b;
5. }
6. }
7. **public** **class** InstanceMethodReference3 {
8. **public** **static** **void** main(String[] args) {
9. BiFunction<Integer, Integer, Integer>adder = **new** Arithmetic()::add;
10. **int** result = adder.apply(10, 20);
11. System.out.println(result);
12. }
13. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=InstanceMethodReference3)

Output:

30

## 3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

1. ClassName::**new**

### Example

1. **interface** Messageable{
2. Message getMessage(String msg);
3. }
4. **class** Message{
5. Message(String msg){
6. System.out.print(msg);
7. }
8. }
9. **public** **class** ConstructorReference {
10. **public** **static** **void** main(String[] args) {
11. Messageable hello = Message::**new**;
12. hello.getMessage("Hello");
13. }
14. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ConstructorReference)

Output:

Hello

# Java Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

### Example 1

1. @FunctionalInterface
2. **interface** sayable{
3. **void** say(String msg);
4. }
5. **public** **class** FunctionalInterfaceExample **implements** sayable{
6. **public** **void** say(String msg){
7. System.out.println(msg);
8. }
9. **public** **static** **void** main(String[] args) {
10. FunctionalInterfaceExample fie = **new** FunctionalInterfaceExample();
11. fie.say("Hello there");
12. }
13. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=FunctionalInterfaceExample)

Output:

Hello there

A functional interface can have methods of object class. See in the following example.

### Example 2

2. @FunctionalInterface
3. **interface** sayable{
4. **void** say(String msg);   // abstract method
5. // It can contain any number of Object class methods.
6. **int** hashCode();
7. String toString();
8. **boolean** equals(Object obj);
9. }
10. **public** **class** FunctionalInterfaceExample2 **implements** sayable{
11. **public** **void** say(String msg){
12. System.out.println(msg);
13. }
14. **public** **static** **void** main(String[] args) {
15. FunctionalInterfaceExample2 fie = **new** FunctionalInterfaceExample2();
16. fie.say("Hello there");
17. }
18. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=FunctionalInterfaceExample2)

Output:

Hello there

### Invalid Functional Interface

A functional interface can extends another interface only when it does not have any abstract method.

1. **interface** sayable{
2. **void** say(String msg);   // abstract method
3. }
4. @FunctionalInterface
5. **interface** Doable **extends** sayable{
6. // Invalid '@FunctionalInterface' annotation; Doable is not a functional interface
7. **void** doIt();
8. }

Output:

compile-time error

### Example 3

In the following example, a functional interface is extending to a non-functional interface.

1. **interface** Doable{
2. **default** **void** doIt(){
3. System.out.println("Do it now");
4. }
5. }
6. @FunctionalInterface
7. **interface** Sayable **extends** Doable{
8. **void** say(String msg);   // abstract method
9. }
10. **public** **class** FunctionalInterfaceExample3 **implements** Sayable{
11. **public** **void** say(String msg){
12. System.out.println(msg);
13. }
14. **public** **static** **void** main(String[] args) {
15. FunctionalInterfaceExample3 fie = **new** FunctionalInterfaceExample3();
16. fie.say("Hello there");
17. fie.doIt();
18. }
19. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=FunctionalInterfaceExample3)

Output:

Hello there

Do it now

### Java Predefined-Functional Interfaces

Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.

You can also define your own custom functional interface. Following is the list of functional interface which are placed in java.util.function package.

|  |  |
| --- | --- |
| **Interface** | **Description** |
| [BiConsumer<T,U>](https://www.javatpoint.com/java-biconsumer-interface) | It represents an operation that accepts two input arguments and returns no result. |
| [Consumer<T>](https://www.javatpoint.com/java-consumer-interface) | It represents an operation that accepts a single argument and returns no result. |
| [Function<T,R>](https://www.javatpoint.com/java-function-interface) | It represents a function that accepts one argument and returns a result. |
| [Predicate<T>](https://www.javatpoint.com/java-predicate-interface) | It represents a predicate (boolean-valued function) of one argument. |
| BiFunction<T,U,R> | It represents a function that accepts two arguments and returns a a result. |
| BinaryOperator<T> | It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands. |
| BiPredicate<T,U> | It represents a predicate (boolean-valued function) of two arguments. |
| BooleanSupplier | It represents a supplier of boolean-valued results. |
| DoubleBinaryOperator | It represents an operation upon two double type operands and returns a double type value. |
| DoubleConsumer | It represents an operation that accepts a single double type argument and returns no result. |
| DoubleFunction<R> | It represents a function that accepts a double type argument and produces a result. |
| DoublePredicate | It represents a predicate (boolean-valued function) of one double type argument. |
| DoubleSupplier | It represents a supplier of double type results. |
| DoubleToIntFunction | It represents a function that accepts a double type argument and produces an int type result. |
| DoubleToLongFunction | It represents a function that accepts a double type argument and produces a long type result. |
| DoubleUnaryOperator | It represents an operation on a single double type operand that produces a double type result. |
| IntBinaryOperator | It represents an operation upon two int type operands and returns an int type result. |
| IntConsumer | It represents an operation that accepts a single integer argument and returns no result. |
| IntFunction<R> | It represents a function that accepts an integer argument and returns a result. |
| IntPredicate | It represents a predicate (boolean-valued function) of one integer argument. |
| IntSupplier | It represents a supplier of integer type. |
| IntToDoubleFunction | It represents a function that accepts an integer argument and returns a double. |
| IntToLongFunction | It represents a function that accepts an integer argument and returns a long. |
| IntUnaryOperator | It represents an operation on a single integer operand that produces an integer result. |
| LongBinaryOperator | It represents an operation upon two long type operands and returns a long type result. |
| LongConsumer | It represents an operation that accepts a single long type argument and returns no result. |
| LongFunction<R> | It represents a function that accepts a long type argument and returns a result. |
| LongPredicate | It represents a predicate (boolean-valued function) of one long type argument. |
| LongSupplier | It represents a supplier of long type results. |
| LongToDoubleFunction | It represents a function that accepts a long type argument and returns a result of double type. |
| LongToIntFunction | It represents a function that accepts a long type argument and returns an integer result. |
| LongUnaryOperator | It represents an operation on a single long type operand that returns a long type result. |
| ObjDoubleConsumer<T> | It represents an operation that accepts an object and a double argument, and returns no result. |
| ObjIntConsumer<T> | It represents an operation that accepts an object and an integer argument. It does not return result. |
| ObjLongConsumer<T> | It represents an operation that accepts an object and a long argument, it returns no result. |
| Supplier<T> | It represents a supplier of results. |
| ToDoubleBiFunction<T,U> | It represents a function that accepts two arguments and produces a double type result. |
| ToDoubleFunction<T> | It represents a function that returns a double type result. |
| ToIntBiFunction<T,U> | It represents a function that accepts two arguments and returns an integer. |
| ToIntFunction<T> | It represents a function that returns an integer. |
| ToLongBiFunction<T,U> | It represents a function that accepts two arguments and returns a result of long type. |
| ToLongFunction<T> | It represents a function that returns a result of long type. |
| UnaryOperator<T> | It represents an operation on a single operand that returnsa a result of the same type as its operand. |

# Java 8 Stream

Java provides a new additional package in Java 8 called java.util.stream. This package consists of classes, interfaces and enum to allows functional-style operations on the elements. You can use stream by importing java.util.stream package.

Stream provides following features:

* Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
* Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

## Java Stream Interface Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| boolean allMatch(Predicate<? super T> predicate) | It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated. |
| boolean anyMatch(Predicate<? super T> predicate) | It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated. |
| static <T> Stream.Builder<T> builder() | It returns a builder for a Stream. |
| <R,A> R collect(Collector<? super T,A,R> collector) | It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning. |
| <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) | It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result. |
| static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b) | It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked. |
| long count() | It returns the count of elements in this stream. This is a special case of a reduction. |
| Stream<T> distinct() | It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream. |
| static <T> Stream<T> empty() | It returns an empty sequential Stream. |
| Stream<T> filter(Predicate<? super T> predicate) | It returns a stream consisting of the elements of this stream that match the given predicate. |
| Optional<T> findAny() | It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty. |
| Optional<T> findFirst() | It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned. |
| <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper) | It returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper) | It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper) | It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper) | It returns a LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| void forEach(Consumer<? super T> action) | It performs an action for each element of this stream. |
| void forEachOrdered(Consumer<? super T> action) | It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order. |
| static <T> Stream<T> generate(Supplier<T> s) | It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc. |
| static <T> Stream<T> iterate(T seed,UnaryOperator<T> f) | It returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc. |
| Stream<T> limit(long maxSize) | It returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length. |
| <R> Stream<R> map(Function<? super T,? extends R> mapper) | It returns a stream consisting of the results of applying the given function to the elements of this stream. |
| DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper) | It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream. |
| IntStream mapToInt(ToIntFunction<? super T> mapper) | It returns an IntStream consisting of the results of applying the given function to the elements of this stream. |
| LongStream mapToLong(ToLongFunction<? super T> mapper) | It returns a LongStream consisting of the results of applying the given function to the elements of this stream. |
| Optional<T> max(Comparator<? super T> comparator) | It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction. |
| Optional<T> min(Comparator<? super T> comparator) | It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction. |
| boolean noneMatch(Predicate<? super T> predicate) | It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated. |
| @SafeVarargs static <T> Stream<T> of(T... values) | It returns a sequential ordered stream whose elements are the specified values. |
| static <T> Stream<T> of(T t) | It returns a sequential Stream containing a single element. |
| Stream<T> peek(Consumer<? super T> action) | It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. |
| Optional<T> reduce(BinaryOperator<T> accumulator) | It performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any. |
| T reduce(T identity, BinaryOperator<T> accumulator) | It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. |
| <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) | It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions. |
| Stream<T> skip(long n) | It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned. |
| Stream<T> sorted() | It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed. |
| Stream<T> sorted(Comparator<? super T> comparator) | It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator. |
| Object[] toArray() | It returns an array containing the elements of this stream. |
| <A> A[] toArray(IntFunction<A[]> generator) | It returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing. |

### Java Example: Filtering Collection without using Stream

In the following example, we are filtering data without using stream. This approach we are used before the stream package was released.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. List<Float> productPriceList = **new** ArrayList<Float>();
22. **for**(Product product: productsList){
24. // filtering data of list
25. **if**(product.price<30000){
26. productPriceList.add(product.price);    // adding price to a productPriceList
27. }
28. }
29. System.out.println(productPriceList);   // displaying data
30. }
31. }

Output:

[25000.0, 28000.0, 28000.0]

### Java Stream Example: Filtering Collection by using Stream

Here, we are filtering data by using stream. You can see that code is optimized and maintained. Stream provides fast execution.

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
13. **public** **class** JavaStreamExample {
14. **public** **static** **void** main(String[] args) {
15. List<Product> productsList = **new** ArrayList<Product>();
16. //Adding Products
17. productsList.add(**new** Product(1,"HP Laptop",25000f));
18. productsList.add(**new** Product(2,"Dell Laptop",30000f));
19. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
20. productsList.add(**new** Product(4,"Sony Laptop",28000f));
21. productsList.add(**new** Product(5,"Apple Laptop",90000f));
22. List<Float> productPriceList2 =productsList.stream()
23. .filter(p -> p.price > 30000)// filtering data
24. .map(p->p.price)        // fetching price
25. .collect(Collectors.toList()); // collecting as list
26. System.out.println(productPriceList2);
27. }
28. }

Output:

[90000.0]

### Java Stream Iterating Example

You can use stream to iterate any number of times. Stream provides predefined methods to deal with the logic you implement. In the following example, we are iterating, filtering and passed a limit to fix the iteration.

1. **import** java.util.stream.\*;
2. **public** **class** JavaStreamExample {
3. **public** **static** **void** main(String[] args){
4. Stream.iterate(1, element->element+1)
5. .filter(element->element%5==0)
6. .limit(5)
7. .forEach(System.out::println);
8. }
9. }

Output:

5

10

15

20

25

### Java Stream Example: Filtering and Iterating Collection

In the following example, we are using filter() method. Here, you can see code is optimized and very concise.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. // This is more compact approach for filtering data
22. productsList.stream()
23. .filter(product -> product.price == 30000)
24. .forEach(product -> System.out.println(product.name));
25. }
26. }

Output:

Dell Laptop

### Java Stream Example : reduce() Method in Collection

This method takes a sequence of input elements and combines them into a single summary result by repeated operation. For example, finding the sum of numbers, or accumulating elements into a list.

In the following example, we are using reduce() method, which is used to sum of all the product prices.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. // This is more compact approach for filtering data
22. Float totalPrice = productsList.stream()
23. .map(product->product.price)
24. .reduce(0.0f,(sum, price)->sum+price);   // accumulating price
25. System.out.println(totalPrice);
26. // More precise code
27. **float** totalPrice2 = productsList.stream()
28. .map(product->product.price)
29. .reduce(0.0f,Float::sum);   // accumulating price, by referring method of Float class
30. System.out.println(totalPrice2);
32. }
33. }

Output:

201000.0

201000.0

### Java Stream Example: Sum by using Collectors Methods

We can also use collectors to compute sum of numeric values. In the following example, we are using Collectors class and it?s specified methods to compute sum of all the product prices.

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
13. **public** **class** JavaStreamExample {
14. **public** **static** **void** main(String[] args) {
15. List<Product> productsList = **new** ArrayList<Product>();
16. //Adding Products
17. productsList.add(**new** Product(1,"HP Laptop",25000f));
18. productsList.add(**new** Product(2,"Dell Laptop",30000f));
19. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
20. productsList.add(**new** Product(4,"Sony Laptop",28000f));
21. productsList.add(**new** Product(5,"Apple Laptop",90000f));
22. // Using Collectors's method to sum the prices.
23. **double** totalPrice3 = productsList.stream()
24. .collect(Collectors.summingDouble(product->product.price));
25. System.out.println(totalPrice3);
27. }
28. }

Output:

201000.0

### Java Stream Example: Find Max and Min Product Price

Following example finds min and max product price by using stream. It provides convenient way to find values without using imperative approach.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. // max() method to get max Product price
22. Product productA = productsList.stream()
23. .max((product1, product2)->
24. product1.price > product2.price ? 1: -1).get();
26. System.out.println(productA.price);
27. // min() method to get min Product price
28. Product productB = productsList.stream()
29. .max((product1, product2)->
30. product1.price < product2.price ? 1: -1).get();
31. System.out.println(productB.price);
33. }
34. }

Output:

90000.0

25000.0

### Java Stream Example: count() Method in Collection

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. // count number of products based on the filter
22. **long** count = productsList.stream()
23. .filter(product->product.price<30000)
24. .count();
25. System.out.println(count);
26. }
27. }

Output:

3

stream allows you to collect your result in any various forms. You can get you result as set, list or map and can perform manipulation on the elements.

### Java Stream Example : Convert List into Set

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
14. **public** **class** JavaStreamExample {
15. **public** **static** **void** main(String[] args) {
16. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
25. // Converting product List into Set
26. Set<Float> productPriceList =
27. productsList.stream()
28. .filter(product->product.price < 30000)   // filter product on the base of price
29. .map(product->product.price)
30. .collect(Collectors.toSet());   // collect it as Set(remove duplicate elements)
31. System.out.println(productPriceList);
32. }
33. }

Output:

[25000.0, 28000.0]

### Java Stream Example : Convert List into Map

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
14. **public** **class** JavaStreamExample {
15. **public** **static** **void** main(String[] args) {
16. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
25. // Converting Product List into a Map
26. Map<Integer,String> productPriceMap =
27. productsList.stream()
28. .collect(Collectors.toMap(p->p.id, p->p.name));
30. System.out.println(productPriceMap);
31. }
32. }

Output:

{1=HP Laptop, 2=Dell Laptop, 3=Lenevo Laptop, 4=Sony Laptop, 5=Apple Laptop}

### Method Reference in stream

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
9. **public** Product(**int** id, String name, **float** price) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
15. **public** **int** getId() {
16. **return** id;
17. }
18. **public** String getName() {
19. **return** name;
20. }
21. **public** **float** getPrice() {
22. **return** price;
23. }
24. }
26. **public** **class** JavaStreamExample {
28. **public** **static** **void** main(String[] args) {
30. List<Product> productsList = **new** ArrayList<Product>();
32. //Adding Products
33. productsList.add(**new** Product(1,"HP Laptop",25000f));
34. productsList.add(**new** Product(2,"Dell Laptop",30000f));
35. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
36. productsList.add(**new** Product(4,"Sony Laptop",28000f));
37. productsList.add(**new** Product(5,"Apple Laptop",90000f));
39. List<Float> productPriceList =
40. productsList.stream()
41. .filter(p -> p.price > 30000) // filtering data
42. .map(Product::getPrice)         // fetching price by referring getPrice method
43. .collect(Collectors.toList());  // collecting as list
44. System.out.println(productPriceList);
45. }
46. }

Output:

[90000.0]

# Java Stream Filter

Java stream provides a method filter() to filter stream elements on the basis of given predicate. Suppose you want to get only even elements of your list then you can do this easily with the help of filter method.

This method takes predicate as an argument and returns a stream of consisting of resulted elements.

## Signature

The signature of Stream filter() method is given below:

1. Stream<T> filter(Predicate<? **super** T> predicate)

### Parameter

**predicate:** It takes Predicate reference as an argument. Predicate is a functional interface. So, you can also pass lambda expression here.

### Return

It returns a new stream.

### Java Stream filter() example

In the following example, we are fetching and iterating filtered data.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. productsList.stream()
22. .filter(p ->p.price> 30000)   // filtering price
23. .map(pm ->pm.price)          // fetching price
24. .forEach(System.out::println);  // iterating price
25. }
26. }

Output:

90000.0

### Java Stream filter() example 2

In the following example, we are fetching filtered data as a list.

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
13. **public** **class** JavaStreamExample {
14. **public** **static** **void** main(String[] args) {
15. List<Product> productsList = **new** ArrayList<Product>();
16. //Adding Products
17. productsList.add(**new** Product(1,"HP Laptop",25000f));
18. productsList.add(**new** Product(2,"Dell Laptop",30000f));
19. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
20. productsList.add(**new** Product(4,"Sony Laptop",28000f));
21. productsList.add(**new** Product(5,"Apple Laptop",90000f));
22. List<Float> pricesList =  productsList.stream()
23. .filter(p ->p.price> 30000)   // filtering price
24. .map(pm ->pm.price)          // fetching price
25. .collect(Collectors.toList());
26. System.out.println(pricesList);
27. }
28. }

Output:

[90000.0]

# Java Base64 Encode and Decode

Java provides a class Base64 to deal with encryption. You can encrypt and decrypt your data by using provided methods. You need to import java.util.Base64 in your source file to use its methods.

This class provides three different encoders and decoders to encrypt information at each level. You can use these methods at the following levels.

## Basic Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 and RFC 2045 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

## URL and Filename Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

## MIME

It uses the Base64 alphabet as specified in RFC 2045 for encoding and decoding operations. 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.

### Nested Classes of Base64

|  |  |
| --- | --- |
| **Class** | **Description** |
| Base64.Decoder | This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. |
| Base64.Encoder | This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045. |

### Base64 Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static Base64.Decoder getDecoder() | It returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme. |
| public static Base64.Encoder getEncoder() | It returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme. |
| public static Base64.Decoder getUrlDecoder() | It returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme. |
| public static Base64.Decoder getMimeDecoder() | It returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme. |
| public static Base64.Encoder getMimeEncoder() | It Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme. |
| public static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator) | It returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators. |
| public static Base64.Encoder getUrlEncoder() | It returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme. |

### Base64.Decoder Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public byte[] decode(byte[] src) | It decodes all bytes from the input byte array using the Base64 encoding scheme, writing the results into a newly-allocated output byte array. The returned byte array is of the length of the resulting bytes. |
| public byte[] decode(String src) | It decodes a Base64 encoded String into a newly-allocated byte array using the Base64 encoding scheme. |
| public int decode(byte[] src, byte[] dst) | It decodes all bytes from the input byte array using the Base64 encoding scheme, writing the results into the given output byte array, starting at offset 0. |
| public ByteBuffer decode(ByteBuffer buffer) | It decodes all bytes from the input byte buffer using the Base64 encoding scheme, writing the results into a newly-allocated ByteBuffer. |
| public InputStream wrap(InputStream is) | It returns an input stream for decoding Base64 encoded byte stream. |

### Base64.Encoder Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public byte[] encode(byte[] src) | It encodes all bytes from the specified byte array into a newly-allocated byte array using the Base64 encoding scheme. The returned byte array is of the length of the resulting bytes. |
| public int encode(byte[] src, byte[] dst) | It encodes all bytes from the specified byte array using the Base64 encoding scheme, writing the resulting bytes to the given output byte array, starting at offset 0. |
| public String encodeToString(byte[] src) | It encodes the specified byte array into a String using the Base64 encoding scheme. |
| public ByteBuffer encode(ByteBuffer buffer) | It encodes all remaining bytes from the specified byte buffer into a newly-allocated ByteBuffer using the Base64 encoding scheme. Upon return, the source buffer's position will be updated to its limit; its limit will not have been changed. The returned output buffer's position will be zero and its limit will be the number of resulting encoded bytes. |
| public OutputStream wrap(OutputStream os) | It wraps an output stream for encoding byte data using the Base64 encoding scheme. |
| public Base64.Encoder withoutPadding() | It returns an encoder instance that encodes equivalently to this one, but without adding any padding character at the end of the encoded byte data. |

### Java Base64 Example: Basic Encoding and Decoding

1. **import** java.util.Base64;
2. publicclass Base64BasicEncryptionExample {
3. publicstaticvoid main(String[] args) {
4. // Getting encoder
5. Base64.Encoder encoder = Base64.getEncoder();
6. // Creating byte array
7. bytebyteArr[] = {1,2};
8. // encoding byte array
9. bytebyteArr2[] = encoder.encode(byteArr);
10. System.out.println("Encoded byte array: "+byteArr2);
11. bytebyteArr3[] = newbyte[5];                // Make sure it has enough size to store copied bytes
12. intx = encoder.encode(byteArr,byteArr3);    // Returns number of bytes written
13. System.out.println("Encoded byte array written to another array: "+byteArr3);
14. System.out.println("Number of bytes written: "+x);
16. // Encoding string
17. String str = encoder.encodeToString("JavaTpoint".getBytes());
18. System.out.println("Encoded string: "+str);
19. // Getting decoder
20. Base64.Decoder decoder = Base64.getDecoder();
21. // Decoding string
22. String dStr = **new** String(decoder.decode(str));
23. System.out.println("Decoded string: "+dStr);
24. }
25. }

Output:

Encoded byte array: [B@6bc7c054

Encoded byte array written to another array: [B@232204a1

Number of bytes written: 4

Encoded string: SmF2YVRwb2ludA==

Decoded string: JavaTpoint

### Java Base64 Example: URL Encoding and Decoding

1. **import** java.util.Base64;
2. publicclass Base64BasicEncryptionExample {
3. publicstaticvoid main(String[] args) {
4. // Getting encoder
5. Base64.Encoder encoder = Base64.getUrlEncoder();
6. // Encoding URL
7. String eStr = encoder.encodeToString("http://www.javatpoint.com/java-tutorial/".getBytes());
8. System.out.println("Encoded URL: "+eStr);
9. // Getting decoder
10. Base64.Decoder decoder = Base64.getUrlDecoder();
11. // Decoding URl
12. String dStr = **new** String(decoder.decode(eStr));
13. System.out.println("Decoded URL: "+dStr);
14. }
15. }

Output:

Encoded URL: aHR0cDovL3d3dy5qYXZhdHBvaW50LmNvbS9qYXZhLXR1dG9yaWFsLw==

Decoded URL: http://www.javatpoint.com/java-tutorial/

### Java Base64 Example: MIME Encoding and Decoding

1. **package** Base64Encryption;
2. **import** java.util.Base64;
3. publicclass Base64BasicEncryptionExample {
4. publicstaticvoid main(String[] args) {
5. // Getting MIME encoder
6. Base64.Encoder encoder = Base64.getMimeEncoder();
7. String message = "Hello, \nYou are informed regarding your inconsistency of work";
8. String eStr = encoder.encodeToString(message.getBytes());
9. System.out.println("Encoded MIME message: "+eStr);
11. // Getting MIME decoder
12. Base64.Decoder decoder = Base64.getMimeDecoder();
13. // Decoding MIME encoded message
14. String dStr = **new** String(decoder.decode(eStr));
15. System.out.println("Decoded message: "+dStr);
16. }
17. }

Output:

Encoded MIME message: SGVsbG8sIApZb3UgYXJlIGluZm9ybWVkIHJlZ2FyZGluZyB5b3VyIGluY29uc2lzdGVuY3kgb2Yg

d29yaw==

Decoded message: Hello,

You are informed regarding your inconsistency of work

# Java Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

### Java Default Method Example

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

Let's see a simple

1. **interface** Sayable{
2. // Default method
3. **default** **void** say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. **void** sayMore(String msg);
8. }
9. **public** **class** DefaultMethods **implements** Sayable{
10. **public** **void** sayMore(String msg){        // implementing abstract method
11. System.out.println(msg);
12. }
13. **public** **static** **void** main(String[] args) {
14. DefaultMethods dm = **new** DefaultMethods();
15. dm.say();   // calling default method
16. dm.sayMore("Work is worship");  // calling abstract method
18. }
19. }

Output:

Hello, this is default method

Work is worship

## Static Methods inside Java 8 Interface

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

1. **interface** Sayable{
2. // default method
3. **default** **void** say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. **void** sayMore(String msg);
8. // static method
9. **static** **void** sayLouder(String msg){
10. System.out.println(msg);
11. }
12. }
13. **public** **class** DefaultMethods **implements** Sayable{
14. **public** **void** sayMore(String msg){     // implementing abstract method
15. System.out.println(msg);
16. }
17. **public** **static** **void** main(String[] args) {
18. DefaultMethods dm = **new** DefaultMethods();
19. dm.say();                       // calling default method
20. dm.sayMore("Work is worship");      // calling abstract method
21. Sayable.sayLouder("Helloooo...");   // calling static method
22. }
23. }

Output:

Hello there

Work is worship

Helloooo...

## Abstract Class vs Java 8 Interface

After having default and static methods inside the interface, we think about the need of abstract class in Java. An interface and an abstract class is almost similar except that you can create constructor in the abstract class whereas you can't do this in interface.

1. **abstract** **class** AbstractClass{
2. **public** AbstractClass() {        // constructor
3. System.out.println("You can create constructor in abstract class");
4. }
5. **abstract** **int** add(**int** a, **int** b); // abstract method
6. **int** sub(**int** a, **int** b){      // non-abstract method
7. **return** a-b;
8. }
9. **static** **int** multiply(**int** a, **int** b){  // static method
10. **return** a\*b;
11. }
12. }
13. **public** **class** AbstractTest **extends** AbstractClass{
14. **public** **int** add(**int** a, **int** b){        // implementing abstract method
15. **return** a+b;
16. }
17. **public** **static** **void** main(String[] args) {
18. AbstractTest a = **new** AbstractTest();
19. **int** result1 = a.add(20, 10);    // calling abstract method
20. **int** result2 = a.sub(20, 10);    // calling non-abstract method
21. **int** result3 = AbstractClass.multiply(20, 10); // calling static method
22. System.out.println("Addition: "+result1);
23. System.out.println("Substraction: "+result2);
24. System.out.println("Multiplication: "+result3);
25. }
26. }

Output:

You can create constructor in abstract class

Addition: 30

Substraction: 10

Multiplication: 200

# Java forEach loop

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

## forEach() Signature in Iterable Interface

1. **default** **void** forEach(Consumer<**super** T>action)

### Java 8 forEach() example 1

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** ForEachExample {
4. **public** **static** **void** main(String[] args) {
5. List<String> gamesList = **new** ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing lambda expression--------------");
11. gamesList.forEach(games -> System.out.println(games));
13. }
14. }

Output:

------------Iterating by passing lambda expression--------------

Football

Cricket

Chess

Hocky

### Java 8 forEach() example 2

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** ForEachExample {
4. **public** **static** **void** main(String[] args) {
5. List<String> gamesList = **new** ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing method reference---------------");
11. gamesList.forEach(System.out::println);
12. }
13. }

Output:

------------Iterating by passing method reference---------------

Football

Cricket

Chess

Hocky

## Java Stream forEachOrdered() Method

Along with forEach() method, Java provides one more method forEachOrdered(). It is used to iterate elements in the order specified by the stream.

### Singnature:

1. **void** forEachOrdered(Consumer<? **super** T> action)

### Java Stream forEachOrdered() Method Example

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** ForEachOrderedExample {
4. **public** **static** **void** main(String[] args) {
5. List<String> gamesList = **new** ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing lambda expression---------------");
11. gamesList.stream().forEachOrdered(games -> System.out.println(games));
12. System.out.println("------------Iterating by passing method reference---------------");
13. gamesList.stream().forEachOrdered(System.out::println);
14. }
16. }

Output:

------------Iterating by passing lambda expression---------------

Football

Cricket

Chess

Hocky

------------Iterating by passing method reference---------------

Football

Cricket

Chess

Hocky

# Java Collectors

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

Java Collectors class provides various methods to deal with elements

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static <T> Collector<T,?,Double> averagingDouble(ToDoubleFunction<? super T> mapper) | It returns a Collector that produces the arithmetic mean of a double-valued function applied to the input elements. If no elements are present, the result is 0. |
| public static <T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> op) | It returns a Collector which performs a reduction of its input elements under a specified BinaryOperator using the provided identity. |
| public static <T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> op) | It returns a Collector which performs a reduction of its input elements under a specified BinaryOperator. The result is described as an Optional<T>. |
| public static <T,U> Collector<T,?,U> reducing(U identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op) | It returns a Collector which performs a reduction of its input elements under a specified mapping function and BinaryOperator. This is a generalization of reducing(Object, BinaryOperator) which allows a transformation of the elements before reduction. |
| public static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier) | It returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map. |
| public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? Super T,A,D> downstream) | It returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. |
| public static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream) | It returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. The Map produced by the Collector is created with the supplied factory function. |
| public static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier) | It returns a concurrent Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function. |
| public static <T,K,A,D> Collector<T,?,ConcurrentMap<K,D>> groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream) | It returns a concurrent Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. |
| public static <T,K,A,D,M extends ConcurrentMap<K,D>> Collector<T,?,M> groupingByConcurrent(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream) | It returns a concurrent Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. The ConcurrentMap produced by the Collector is created with the supplied factory function. |
| public static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate) | It returns a Collector which partitions the input elements according to a Predicate, and organizes them into a Map<Boolean, List<T>>. There are no guarantees on the type, mutability, serializability, or thread-safety of the Map returned. |
| public static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(Predicate<? super T> predicate, Collector<? Super T,A,D> downstream) | It returns a Collector which partitions the input elements according to a Predicate, reduces the values in each partition according to another Collector, and organizes them into a Map<Boolean, D> whose values are the result of the downstream reduction. |
| public static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper) | It returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction) | It returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier) | It returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper) | It returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction) | It returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T,K,U,M extends ConcurrentMap<K,U>> Collector<T,?,M> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier) | It returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements. |
| public static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper) | It returns a Collector which applies an int-producing mapping function to each input element, and returns summary statistics for the resulting values. |
| public static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper) | It returns a Collector which applies an long-producing mapping function to each input element, and returns summary statistics for the resulting values. |
| public static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper) | It returns a Collector which applies an double-producing mapping function to each input element, and returns summary statistics for the resulting values. |

### Java Collectors Example: Fetching data as a List

1. **import** java.util.stream.Collectors;
2. **import** java.util.List;
3. **import** java.util.ArrayList;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
9. **public** Product(**int** id, String name, **float** price) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** CollectorsExample {
16. **public** **static** **void** main(String[] args) {
17. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
24. List<Float> productPriceList =
25. productsList.stream()
26. .map(x->x.price)         // fetching price
27. .collect(Collectors.toList());  // collecting as list
28. System.out.println(productPriceList);
29. }
30. }

Output:

[25000.0, 30000.0, 28000.0, 28000.0, 90000.0]

### Java Collectors Example: Converting Data as a Set

1. **import** java.util.stream.Collectors;
2. **import** java.util.Set;
3. **import** java.util.List;
4. **import** java.util.ArrayList;
5. classProduct{
6. intid;
7. String name;
8. floatprice;
10. **public** Product(intid, String name, floatprice) {
11. **this**.id = id;
12. **this**.name = name;
13. **this**.price = price;
14. }
15. }
16. publicclass CollectorsExample {
17. publicstaticvoid main(String[] args) {
18. List<Product>productsList = **new** ArrayList<Product>();
19. //Adding Products
20. productsList.add(newProduct(1,"HP Laptop",25000f));
21. productsList.add(newProduct(2,"Dell Laptop",30000f));
22. productsList.add(newProduct(3,"Lenevo Laptop",28000f));
23. productsList.add(newProduct(4,"Sony Laptop",28000f));
24. productsList.add(newProduct(5,"Apple Laptop",90000f));
25. Set<Float>productPriceList =
26. productsList.stream()
27. .map(x->x.price)         // fetching price
28. .collect(Collectors.toSet());   // collecting as list
29. System.out.println(productPriceList);
30. }
31. }

Output:

[25000.0, 30000.0, 28000.0, 90000.0]

### Java Collectors Example: using sum method

1. **import** java.util.stream.Collectors;
2. **import** java.util.List;
3. **import** java.util.ArrayList;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
9. **public** Product(**int** id, String name, **float** price) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** CollectorsExample {
16. **public** **static** **void** main(String[] args) {
17. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
24. Double sumPrices =
25. productsList.stream()
26. .collect(Collectors.summingDouble(x->x.price));  // collecting as list
27. System.out.println("Sum of prices: "+sumPrices);
28. Integer sumId =
29. productsList.stream().collect(Collectors.summingInt(x->x.id));
30. System.out.println("Sum of id's: "+sumId);
31. }
32. }

Output:

Sum of prices: 201000.0

Sum of id's: 15

### Java Collectors Example: Getting Product Average Price

1. **import** java.util.stream.Collectors;
2. **import** java.util.List;
3. **import** java.util.ArrayList;
4. **class** Product{
5. **int** id;
6. String name;
7. **float** price;
9. **public** Product(**int** id, String name, **float** price) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. }
15. **public** **class** CollectorsExample {
16. **public** **static** **void** main(String[] args) {
17. List<Product> productsList = **new** ArrayList<Product>();
18. //Adding Products
19. productsList.add(**new** Product(1,"HP Laptop",25000f));
20. productsList.add(**new** Product(2,"Dell Laptop",30000f));
21. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
22. productsList.add(**new** Product(4,"Sony Laptop",28000f));
23. productsList.add(**new** Product(5,"Apple Laptop",90000f));
24. Double average = productsList.stream()
25. .collect(Collectors.averagingDouble(p->p.price));
26. System.out.println("Average price is: "+average);
27. }
28. }

Output:

Average price is: 40200.0

### Java Collectors Example: Counting Elements

1. **import** java.util.stream.Collectors;
2. **import** java.util.List;
3. **import** java.util.ArrayList;
4. **class** Product{
5. intid;
6. String name;
7. floatprice;
9. **public** Product(intid, String name, floatprice) {
10. **this**.id = id;
11. **this**.name = name;
12. **this**.price = price;
13. }
14. publicint getId() {
15. returnid;
16. }
17. **public** String getName() {
18. returnname;
19. }
20. publicfloat getPrice() {
21. returnprice;
22. }
23. }
24. publicclass CollectorsExample {
25. publicstaticvoid main(String[] args) {
26. List<Product>productsList = **new** ArrayList<Product>();
27. //Adding Products
28. productsList.add(**new** Product(1,"HP Laptop",25000f));
29. productsList.add(**new** Product(2,"Dell Laptop",30000f));
30. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
31. productsList.add(**new** Product(4,"Sony Laptop",28000f));
32. productsList.add(**new** Product(5,"Apple Laptop",90000f));
33. Long noOfElements = productsList.stream()
34. .collect(Collectors.counting());
35. System.out.println("Total elements : "+noOfElements);
36. }
37. }

Output:

Total elements : 5

# Java StringJoiner

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc. You can also pass prefix and suffix to the char sequence.

### StringJoiner Constructors

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| Public StringJoiner(CharSequence delimiter) | It constructs a StringJoiner with no characters in it, with no prefix or suffix, and a copy of the supplied delimiter. It throws NullPointerException if delimiter is null. |
| Public StringJoiner(CharSequence delimiter,CharSequence prefix,CharSequence suffix) | It constructs a StringJoiner with no characters in it using copies of the supplied prefix, delimiter and suffix. It throws NullPointerException if prefix, delimiter, or suffix is null. |

### StringJoiner Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| Public StringJoiner add(CharSequence newElement) | It adds a copy of the given CharSequence value as the next element of the StringJoiner value. If newElement is null,"null" is added. |
| Public StringJoiner merge(StringJoiner other) | It adds the contents of the given StringJoiner without prefix and suffix as the next element if it is non-empty. If the given StringJoiner is empty, the call has no effect. |
| Public int length() | It returns the length of the String representation of this StringJoiner. |
| Public StringJoiner setEmptyValue(CharSequence emptyValue) | It sets the sequence of characters to be used when determining the string representation of this StringJoiner and no elements have been added yet, that is, when it is empty. |

### Java StringJoiner Example

1. // importing StringJoiner class
2. **import** java.util.StringJoiner;
3. **public** **class** StringJoinerExample {
4. **public** **static** **void** main(String[] args) {
5. StringJoiner joinNames = **new** StringJoiner(","); // passing comma(,) as delimiter
7. // Adding values to StringJoiner
8. joinNames.add("Rahul");
9. joinNames.add("Raju");
10. joinNames.add("Peter");
11. joinNames.add("Raheem");
13. System.out.println(joinNames);
14. }
15. }

Output:

Rahul,Raju,Peter,Raheem

### Java StringJoiner Example: adding prefix and suffix

1. // importing StringJoiner class
2. **import** java.util.StringJoiner;
3. **public** **class** StringJoinerExample {
4. **public** **static** **void** main(String[] args) {
5. StringJoiner joinNames = **new** StringJoiner(",", "[", "]");   // passing comma(,) and square-brackets as delimiter
7. // Adding values to StringJoiner
8. joinNames.add("Rahul");
9. joinNames.add("Raju");
10. joinNames.add("Peter");
11. joinNames.add("Raheem");
13. System.out.println(joinNames);
14. }
15. }

Output:

[Rahul,Raju,Peter,Raheem]

### StringJoiner Example: Merge Two StringJoiner

The merge() method merges two StringJoiner objects excluding of prefix and suffix of second StringJoiner object.

1. // importing StringJoiner class
2. **import** java.util.StringJoiner;
3. **public** **class** StringJoinerExample {
4. **public** **static** **void** main(String[] args) {
6. StringJoiner joinNames = **new** StringJoiner(",", "[", "]");   // passing comma(,) and square-brackets as delimiter
8. // Adding values to StringJoiner
9. joinNames.add("Rahul");
10. joinNames.add("Raju");
12. // Creating StringJoiner with :(colon) delimiter
13. StringJoiner joinNames2 = **new** StringJoiner(":", "[", "]");  // passing colon(:) and square-brackets as delimiter
15. // Adding values to StringJoiner
16. joinNames2.add("Peter");
17. joinNames2.add("Raheem");
19. // Merging two StringJoiner
20. StringJoiner merge = joinNames.merge(joinNames2);
21. System.out.println(merge);
22. }
23. }

Output:

[Rahul,Raju,Peter:Raheem]

### StringJoiner Example: StringJoiner Methods

1. // importing StringJoiner class
2. **import** java.util.StringJoiner;
3. **public** **class** StringJoinerExample {
4. **public** **static** **void** main(String[] args) {
5. StringJoiner joinNames = **new** StringJoiner(","); // passing comma(,) as delimiter
7. // Prints nothing because it is empty
8. System.out.println(joinNames);
10. // We can set default empty value.
11. joinNames.setEmptyValue("It is empty");
12. System.out.println(joinNames);

15. // Adding values to StringJoiner
16. joinNames.add("Rahul");
17. joinNames.add("Raju");
18. System.out.println(joinNames);
20. // Returns length of StringJoiner
21. **int** length = joinNames.length();
22. System.out.println("Length: "+length);
24. // Returns StringJoiner as String type
25. String str = joinNames.toString();
26. System.out.println(str);
28. // Now, we can apply String methods on it
29. **char** ch = str.charAt(3);
30. System.out.println("Character at index 3: "+ch);
32. // Adding one more element
33. joinNames.add("Sorabh");
34. System.out.println(joinNames);
36. // Returns length
37. **int** newLength = joinNames.length();
38. System.out.println("New Length: "+newLength);
39. }
40. }

Output:

It is empty

Rahul,Raju

Length: 10

Rahul,Raju

Character at index 3: u

Rahul,Raju,Sorabh

New Length: 17

# Java Optional Class

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

## Java Optional Class Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static <T> Optional<T> empty() | It returns an empty Optional object. No value is present for this Optional. |
| public static <T> Optional<T> of(T value) | It returns an Optional with the specified present non-null value. |
| public static <T> Optional<T> ofNullable(T value) | It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional. |
| public T get() | If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException. |
| public boolean isPresent() | It returns true if there is a value present, otherwise false. |
| public void ifPresent(Consumer<? super T> consumer) | If a value is present, invoke the specified consumer with the value, otherwise do nothing. |
| public Optional<T> filter(Predicate<? super T> predicate) | If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional. |
| public <U> Optional<U> map(Function<? super T,? 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. Otherwise return an empty Optional. |
| public <U> Optional<U> flatMap(Function<? super T,Optional<U> mapper) | If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. |
| public T orElse(T other) | It returns the value if present, otherwise returns other. |
| public T orElseGet(Supplier<? extends T> other) | It returns the value if present, otherwise invoke other and return the result of that invocation. |
| public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X extends Throwable | It returns the contained value, if present, otherwise throw an exception to be created by the provided supplier. |
| public boolean equals(Object obj) | Indicates whether some other object is "equal to" this Optional or not. The other object is considered equal if:   * It is also an Optional and; * Both instances have no value present or; * the present values are "equal to" each other via equals(). |
| public int hashCode() | It returns the hash code value of the present value, if any, or returns 0 (zero) if no value is present. |
| public String toString() | It returns a non-empty string representation of this Optional suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions. |

### Example: Java Program without using Optional

In the following example, we are not using Optional class. This program terminates abnormally and throws a nullPointerException.

1. **public** **class** OptionalExample {
2. **public** **static** **void** main(String[] args) {
3. String[] str = **new** String[10];
4. String lowercaseString = str[5].toLowerCase();
5. System.out.print(lowercaseString);
6. }
7. }

Output:

Exception in thread "main" java.lang.NullPointerException

at lambdaExample.OptionalExample.main(OptionalExample.java:6)

To avoid the abnormal termination, we use Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

### Java Optional Example: If Value is not Present

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. Optional<String> checkNull = Optional.ofNullable(str[5]);
6. **if**(checkNull.isPresent()){  // check for value is present or not
7. String lowercaseString = str[5].toLowerCase();
8. System.out.print(lowercaseString);
9. }**else**
10. System.out.println("string value is not present");
11. }
12. }

Output:

string value is not present

### Java Optional Example: If Value is Present

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";// Setting value for 5th index
6. Optional<String> checkNull = Optional.ofNullable(str[5]);
7. **if**(checkNull.isPresent()){  // It Checks, value is present or not
8. String lowercaseString = str[5].toLowerCase();
9. System.out.print(lowercaseString);
10. }**else**
11. System.out.println("String value is not present");
12. }
13. }

Output:

java optional class example

### Another Java Optional Example

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
6. Optional<String> checkNull = Optional.ofNullable(str[5]);
7. checkNull.ifPresent(System.out::println);   // printing value by using method reference
8. System.out.println(checkNull.get());    // printing value by using get method
9. System.out.println(str[5].toLowerCase());
10. }
11. }

Output:

JAVA OPTIONAL CLASS EXAMPLE

JAVA OPTIONAL CLASS EXAMPLE

java optional class example

### Java Optional Methods Example

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
6. // It returns an empty instance of Optional class
7. Optional<String> empty = Optional.empty();
8. System.out.println(empty);
9. // It returns a non-empty Optional
10. Optional<String> value = Optional.of(str[5]);
11. // If value is present, it returns an Optional otherwise returns an empty Optional
12. System.out.println("Filtered value: "+value.filter((s)->s.equals("Abc")));
13. System.out.println("Filtered value: "+value.filter((s)->s.equals("JAVA OPTIONAL CLASS EXAMPLE")));
14. // It returns value of an Optional. if value is not present, it throws an NoSuchElementException
15. System.out.println("Getting value: "+value.get());
16. // It returns hashCode of the value
17. System.out.println("Getting hashCode: "+value.hashCode());
18. // It returns true if value is present, otherwise false
19. System.out.println("Is value present: "+value.isPresent());
20. // It returns non-empty Optional if value is present, otherwise returns an empty Optional
21. System.out.println("Nullable Optional: "+Optional.ofNullable(str[5]));
22. // It returns value if available, otherwise returns specified value,
23. System.out.println("orElse: "+value.orElse("Value is not present"));
24. System.out.println("orElse: "+empty.orElse("Value is not present"));
25. value.ifPresent(System.out::println);   // printing value by using method reference
26. }
27. }

Output:

Optional.empty

Filtered value: Optional.empty

Filtered value: Optional[JAVA OPTIONAL CLASS EXAMPLE]

Getting value: JAVA OPTIONAL CLASS EXAMPLE

Getting hashCode: -619947648

Is value present: true

Nullable Optional: Optional[JAVA OPTIONAL CLASS EXAMPLE]

orElse: JAVA OPTIONAL CLASS EXAMPLE

orElse: Value is not present

JAVA OPTIONAL CLASS EXAMPLE

# Java Nashorn

Nashorn is a JavaScript engine. It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine). Java provides a command-line tool jjs which is used to execute JavaScript code.

You can execute JavaScript code by using jjs command-line tool and by embedding into Java source code.

### Example: Executing by Using Terminal

Following is the step by step process to execute JavaScript code at the JVM.

1) Create a file hello.js.

2) Write and save the following code into the file.

1. var hello = function(){
2. print("Hello Nashorn");
3. };
4. hello();

3) Open terminal

4) Write command **jjs hello.js** and press enter.

After executing command, you will see the below output.

Output:

Hello Nashorn

### Example: Executing JavaScript file in Java Code

You can execute JavaScript file directly from your Java file. In the following code, we are reading a file hello.js with the help of FileReader class.

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. // Reading Nashorn file
8. ee.eval(**new** FileReader("js/hello.js"));
9. }
10. }

Output:

Hello Nashorn

### Example: Embedding JavaScript Code in Java Source File

You can embed your JavaScript code in Java source file. Java compiler will not complaint but it is not good practice when you have large source code. In the following example, we are evaluating JavaScript code.

1. **import** javax.script.\*;
2. **public** **class** NashornExample {
3. **public** **static** **void** main(String[] args) **throws** Exception{
4. // Creating script engine
5. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
6. // Evaluating Nashorn code
7. ee.eval("print('Hello Nashorn');");
8. }
9. }

Output:

Hello Nashorn

### Example: Embedding JavaScript Expression

You can embed JavaScript expressions and variables in JavaScript code. In the following code we are embedding a variable to string. To execute this program you need to pass a flag -scripting in command-line.

File: hello.js

1. var hello = function(msg){
2. print("Hello ${msg}");
3. };
4. hello("Nashron");

**Command:** jjs -scripting hello.js

Output:

Hello Nashorn

## Heredocs

In Nashorn, heredocs are simply multi-line strings. You can create it with << followed by a special termination marker, which is EOF. You can also embed JavaScript expressions in ${...} expressions.

### Example : Heredocs in JavaScript File

file: hello.js

1. var message = <<EOF
2. This is a java script file
3. it contains multiple lines
4. of code.
5. let's execute.
6. EOF
7. print(message)

**Command:** jjs -scripting hello.js

Output:

This is a java script file

it contains multiple lines

of code.

let's execute.

### Example: Setting JavaScript variable in Java File

You can pass value to JavaScript variable in the Java file. In the followed example, we are binding and passing variable to JavaScript file.

File: hello.js

1. print("Hello "+name);

File: NashornExample.java

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. //Binding script and Define scope of script
8. Bindings bind = ee.getBindings(ScriptContext.ENGINE\_SCOPE);
9. bind.put("name", "Nashorn");
10. // Reading Nashorn file
11. ee.eval(**new** FileReader("js/hello.js"));
12. }
13. }

Output:

Hello Nashorn

## Import Java Package in JavaScript File

Java provides a facility to import Java package inside the JavaScript code. Here, we are using two approaches to import Java packages.

### Example1: Import Java Package in JavaScript File

File: hello.js

1. print(java.lang.Math.sqrt(4));

Output:

2

### Example2: Import Java Package in JavaScript File

File: hello.js

1. var importFile = **new** JavaImporter(java.util);
2. var a = **new** importFile.ArrayList();
3. a.add(12);
4. a.add(20);
5. print(a);
6. print(a.getClass());

Output:

[12, 20]

class java.util.ArrayList

### Example3: Import Java Package in JavaScript File

you can import multiple packages at the same time.

File: hello.js

1. var importIt = **new** JavaImporter(java.lang.String,java.util,java.io);
2. with (importIt) {
3. var linkedHS = **new** LinkedHashSet();
4. linkedHS.add(**new** File("abc"));
5. linkedHS.add(**new** File("hello.js"));
6. linkedHS.add("india".toUpperCase());
7. }
8. print(linkedHS);

Output:

[abc, hello.js, INDIA]

## Calling JavaScript function inside Java code

You can call JavaScript function inside the Java file. In the followed example, we are calling JavaScript functions.

### Example: Calling function inside Java code

File: hello.js

1. var functionDemo1 = function(){
2. print("This is JavaScript function");
3. }
4. var functionDemo2 = function(message){
5. print("Hello "+message);
6. }

File: NashornExample.java

1. **import** javax.script.\*;
2. **import** java.io.\*;
3. **public** **class** NashornExample {
4. **public** **static** **void** main(String[] args) **throws** Exception{
5. // Creating script engine
6. ScriptEngine ee = **new** ScriptEngineManager().getEngineByName("Nashorn");
7. // Reading Nashorn file
8. ee.eval(**new** FileReader("js/hello.js"));
9. Invocable invocable = (Invocable)ee;
10. // calling a function
11. invocable.invokeFunction("functionDemo1");
12. // calling a function and passing variable as well.
13. invocable.invokeFunction("functionDemo2","Nashorn");
14. }
15. }

Output:

This is JavaScript function

Hello Nashorn

# Java Parallel Array Sorting

Java provides a new additional feature in Array class which is used to sort array elements parallel.New methods has added to java.util.Arrays package that use the JSR 166 Fork/Join parallelism common pool to provide sorting of arrays in parallel.The methods are called parallelSort() and are overloaded for all the primitive data types and Comparable objects.

The following table contains Arrays overloaded sorting methods.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public static void parallelSort(byte[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(byte[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(char[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(char[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(double[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(double[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(float[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(float[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(int[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(int[] a,int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(long[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(long[] a, int fromIndex, int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static void parallelSort(short[] a) | It sorts the specified array into ascending numerical order. |
| public static void parallelSort(short[] a,int fromIndex,int toIndex) | It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty. |
| public static <T extends Comparable<? super T>> void parallelSort(T[] a) | Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array). |
| public static <T7gt; void parallelSort(T[] a,Comparator<? super T> cmp) | It sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the array). |
| public static <T extends Comparable<? super T>> void parallelSort(T[] a,int fromIndex, int toIndex) | It sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be sorted is empty.) All elements in this range must implement the Comparable interface. Furthermore, all elements in this range must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array). |
| public static <T> void parallelSort(T[] a, int fromIndex, int toIndex, Comparator<? super T> cmp) | It sorts the specified range of the specified array of objects according to the order induced by the specified comparator. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be sorted is empty.) All elements in the range must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the range). |

### Java Parallel Array Sorting Example

1. **import** java.util.Arrays;
2. **public** **class** ParallelArraySorting {
3. **public** **static** **void** main(String[] args) {
4. // Creating an integer array
5. **int**[] arr = {5,8,1,0,6,9};
6. // Iterating array elements
7. **for** (**int** i : arr) {
8. System.out.print(i+" ");
9. }
10. // Sorting array elements parallel
11. Arrays.parallelSort(arr);
12. System.out.println("\nArray elements after sorting");
13. // Iterating array elements
14. **for** (**int** i : arr) {
15. System.out.print(i+" ");
16. }
17. }
18. }

Output:

5 8 1 0 6 9

Array elements after sorting

0 1 5 6 8 9

## Java Parallel Array Sorting Example: Passing Start and End Index

In the following example, we are passing starting and end index of the array. The first index is inclusive and end index is exclusive i.e. if we pass 0 as start index and 4 as end index, only 0 to 3 index elements will be sorted.

It throws IllegalArgumentException if start index > end index.

It throws ArrayIndexOutOfBoundsException if start index < 0 or end index > a.length.

1. **import** java.util.Arrays;
2. **public** **class** ParallelArraySorting {
3. **public** **static** **void** main(String[] args) {
4. // Creating an integer array
5. **int**[] arr = {5,8,1,0,6,9,50,-3};
6. // Iterating array elements
7. **for** (**int** i : arr) {
8. System.out.print(i+" ");
9. }
10. // Sorting array elements parallel and passing start, end index
11. Arrays.parallelSort(arr,0,4);
12. System.out.println("\nArray elements after sorting");
13. // Iterating array elements
14. **for** (**int** i : arr) {
15. System.out.print(i+" ");
16. }
17. }
18. }

Output:

5 8 1 0 6 9 50 -3

Array elements after sorting

0 1 5 8 6 9 50 -3

# Java Type Inference

Type inference is a feature of Java which provides ability to compiler to look at each method invocation and corresponding declaration to determine the type of arguments.

Java provides improved version of type inference in Java 8. the following example explains, how we can use type inference in our code:

Here, we are creating arraylist by mentioning integer type explicitly at both side. The following approach is used earlier versions of Java.

1. List<Integer> list = **new** ArrayList<Integer>();

In the following declaration, we are mentioning type of arraylist at one side. This approach was introduce in Java 7. Here, you can left second side as blank diamond and compiler will infer type of it by type of reference variable.

1. List<Integer> list2 = **new** ArrayList<>();

### Improved Type Inference

In Java 8, you can call specialized method without explicitly mentioning of type of arguments.

1. showList(**new** ArrayList<>());

## Java Type Inference Example

You can use type inference with generic classes and methods.

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** TypeInferenceExample {
4. **public** **static** **void** showList(List<Integer>list){
5. **if**(!list.isEmpty()){
6. list.forEach(System.out::println);
7. }**else** System.out.println("list is empty");
8. }
9. **public** **static** **void** main(String[] args) {
10. // An old approach(prior to Java 7) to create a list
11. List<Integer> list1 = **new** ArrayList<Integer>();
12. list1.add(11);
13. showList(list1);
14. // Java 7
15. List<Integer> list2 = **new** ArrayList<>(); // You can left it blank, compiler can infer type
16. list2.add(12);
17. showList(list2);
18. // Compiler infers type of ArrayList, in Java 8
19. showList(**new** ArrayList<>());
20. }
21. }

Output:

11

12

list is empty

You can also create your own custom generic class and methods. In the following example, we are creating our own generic class and method.

## Java Type Inference Example 2

1. **class** GenericClass<X> {
2. X name;
3. **public** **void** setName(X name){
4. **this**.name = name;
5. }
6. **public** X getName(){
7. returnname;
8. }
9. **public** String genericMethod(GenericClass<String> x){
10. x.setName("John");
11. returnx.name;
12. }
13. }
15. **public** **class** TypeInferenceExample {
16. **public** **static** **void** main(String[] args) {
17. GenericClass<String> genericClass = **new** GenericClass<String>();
18. genericClass.setName("Peter");
19. System.out.println(genericClass.getName());
21. GenericClass<String> genericClass2 = **new** GenericClass<>();
22. genericClass2.setName("peter");
23. System.out.println(genericClass2.getName());
25. // New improved type inference
26. System.out.println(genericClass2.genericMethod(**new** GenericClass<>()));
27. }
28. }

Output:

Peter

peter

John

# Method Parameter Reflection

Java provides a new feature in which you can get the names of formal parameters of any method or constructor. The java.lang.reflect package contains all the required classes like Method and Parameter to work with parameter reflection.

### Method class

It provides information about single method on a class or interface. The reflected method may be a class method or an instance method.

### Method Class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| public boolean equals(Object obj) | It compares this Method against the specified object. It returns true if the objects are the same. Two Methods are the same if they were declared by the same class and have the same name and formal parameter types and return type. |
| public AnnotatedType getAnnotatedReturnType() | It returns an AnnotatedType object that represents the use of a type to specify the return type of the method/constructor. |
| public <T extends Annotation> T getAnnotation(Class<T> annotationClass) | It returns this element's annotation for the specified type if such an annotation is present otherwise returns null. NullPointerException - if the given annotation class is null |
| public Annotation[] getDeclaredAnnotations() | It returns annotations that are directly present on this element. This method ignores inherited annotations. If there are no annotations directly present on this element, the return value is an array of length 0. The caller of this method is free to modify the returned array. it will have no effect on the arrays returned to other callers. |
| public Class<?> getDeclaringClass() | It returns the Class object representing the class or interface that declares the executable represented by this object. |
| public Object getDefaultValue() | It returns the default value for the annotation member represented by this Method instance. |
| public Class<?>[] getExceptionTypes() | It returns an array of Class objects that represent the types of exceptions declared to be thrown by the underlying executable represented by this object. |
| public Type[] getGenericExceptionTypes() | It returns an array of Type objects that represent the exceptions declared to be thrown by this executable object. It returns an array of length 0 if the underlying executable declares no exceptions in its throws clause. It throws following exceptions: **GenericSignatureFormatError** - if the generic method signature does not conform to the format specified in The Java Virtual Machine Specification. **TypeNotPresentException** - if the underlying executable's throws clause refers to a non-existent type declaration. **MalformedParameterizedTypeException** - if the underlying executable's throws clause refers to a parameterized type that cannot be instantiated for any reason. |
| public Type[] getGenericParameterTypes() | It returns an array of Type objects that represent the formal parameter types. It throws following exceptions: **GenericSignatureFormatError** - if the generic method signature does not conform to the format specified in The Java Virtual Machine Specification. **TypeNotPresentException** - if any of the parameter types of the underlying executable refers to a non-existent type declaration. **MalformedParameterizedTypeException** - if any of the underlying executable's parameter types refer to a parameterized type that cannot be instantiated for any reason. |
| public int getModifiers() | It returns the Java language modifiers for the executable represented by this object. |
| public String getName() | It returns the name of the method represented by this Method object as a String. |
| public Annotation[][] getParameterAnnotations() | It returns an array of arrays that represent the annotations on the formal and implicit parameters, in declaration order, of the executable represented by this object. |
| public int getParameterCount() | It returns the number of formal parameters for the executable represented by this object. |
| public Class<?>[] getParameterTypes() | It returns an array of Class objects that represent the formal parameter types. in declaration order, of the executable represented by this object. It returns an array of length 0 if the underlying executable takes no parameters. |
| public Class<?> getReturnType() | It returns a Class object that represents the formal return type of the method represented by this Method object. |
| public TypeVariable<Method>[] getTypeParameters() | It returns an array of TypeVariable objects that represent the type variables declared by the generic declaration represented by this GenericDeclaration object, in declaration order. It throws GenericSignatureFormatError, if the generic signature of this generic declaration does not conform to the format specified in The Java Virtual Machine Specification |
| public int hashCode() | It returns a hashcode for this Method. The hashcode is computed as the exclusive-or of the hashcodes for the underlying method's declaring class name and the method's name. |
| public Object invoke(Object obj, Object... args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException | It invokes the underlying method represented by this Method object, on the specified object with the specified parameters. If the underlying method is static, the specified obj argument is ignored. It may be null. If the number of formal parameters required by the underlying method is 0, the supplied args array may be of length 0 or null. If the underlying method is an instance method, it is invoked using dynamic method lookup as documented in The Java Language Specification. If the underlying method is static, the class that declared the method is initialized if it has not already been initialized. If the method completes normally, the value it returns is returned to the caller of invoke. |
| public boolean isBridge() | It returns true if this method is a bridge method. otherwise returns false. |
| public boolean isDefault() | It returns true if this method is a default method otherwise returns false. A default method is a public non-abstract instance method, that is, a non-static method with a body, declared in an interface type. |
| public boolean isSynthetic() | It returns true if this executable is a synthetic construct; returns false otherwise. |
| public boolean isVarArgs() | It returns true if this executable was declared to take a variable number of arguments; returns false otherwise. |
| public String toGenericString() | It returns a string describing this Method, including type parameters. |
| public String toString() | It returns a string. |

## Parameter class

Parameter class provides information about method parameters, including its name and modifiers. It also provides an alternate means of obtaining attributes for the parameter.

### Parameter Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| public boolean equals(Object obj) | It compares based on the executable and the index. |
| public AnnotatedType getAnnotatedType() | It returns an AnnotatedType object that represents the use of a type to specify the type of the formal parameter represented by this Parameter. |
| public <T extends Annotation> T getAnnotation(Class<T> annotationClass) | It returns this element's annotation for the specified type if such an annotation is present, else null. It throws NullPointerException, if the given annotation class is null. |
| public Annotation[] getAnnotations() | It returns annotations that are present on this element. If there are no annotations present on this element, the return value is an array of length 0. |
| public <T extends Annotation> T[] getAnnotationsByType(Class<T> annotationClass) | It returns annotations that are associated with this element. If there are no annotations associated with this element, the return value is an array of length 0. The difference between this method and AnnotatedElement.getAnnotation(Class) is that this method detects if its argument is a repeatable annotation type (JLS 9.6), and if so, attempts to find one or more annotations of that type by "looking through" a container annotation. It throws NullPointerException, if the given annotation class is null. |
| public <T extends Annotation> T getDeclaredAnnotation(Class<T> annotationClass) | It returns this element's annotation for the specified type if such an annotation is directly present, else null. This method ignores inherited annotations. It throws NullPointerException, if the given annotation class is null. |
| public Annotation[] getDeclaredAnnotations() | It returns annotations that are directly present on this element. This method ignores inherited annotations. If there are no annotations directly present on this element, the return value is an array of length 0. |
| public <T extends Annotation> T[] getDeclaredAnnotationsByType(Class<T> annotationClass) | It returns this element's annotations for the specified type if such annotations are either directly present or indirectly present. This method ignores inherited annotations. If there are no specified annotations directly or indirectly present on this element, the return value is an array of length 0. The difference between this method and AnnotatedElement.getDeclaredAnnotation(Class) is that this method detects if its argument is a repeatable annotation type (JLS 9.6), and if so, attempts to find one or more annotations of that type by "looking through" a container annotation if one is present. The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers. It throws NullPointerException, if the given annotation class is null |
| public Executable getDeclaringExecutable() | It returns the Executable which declares this parameter. |
| public int getModifiers() | It returns the modifier flags for the parameter represented by this Parameter object. |
| public String getName() | It returns the name of the parameter. If the parameter's name is present, this method returns the name provided by the class file. Otherwise, this method synthesizes a name of the form argN, where N is the index of the parameter in the descriptor of the method which declares the parameter. |
| public Type getParameterizedType() | It returns a Type object that identifies the parameterized type for the parameter represented by this Parameter object. |
| public Class<?> getType() | It returns a Class object that identifies the declared type for the parameter represented by this Parameter object. |
| public int hashCode()mul int arg0 int arg1 add int arg0 int arg1 | It returns a hash code based on the executable's hash code and the index. |
| public boolean isImplicit() | It returns true if this parameter is implicitly declared in source code. Otherwise, returns false. |
| public boolean isNamePresent() | It returns true if the parameter has a name according to the class file, otherwise, returns false. |
| public boolean isSynthetic() | It returns true if this parameter is neither implicitly nor explicitly declared in source code. Otherwise returns false. |
| public boolean isVarArgs() | It returns true if this parameter represents a variable argument list; returns false otherwise. |
| public String toString() | It returns a string describing this parameter. The format is the modifiers for the parameter, if any, in canonical order as recommended by The Java? Language Specification. |

## Java Method Parameter Reflection Example

*File: Calculate.java*

1. **public** **class** Calculate {
2. **int** add(**int** a, **int** b){
3. **return** (a+b);
4. }
5. **int** mul(**int** a, **int** b){
6. **return** (b\*a);
7. }
8. }

#### Note - before compiling and executing below code, first compile Calculate class by following command:

1. javac -parameters Calculate.java

**-parameter** flag in the above command is used to store parameters in the Calculate class file. By default .class does not store parameters and returns argsN as parameter name, where N is a number of parameters in the method.

*File: ParameterReflection.java*

1. **import** java.lang.reflect.Method;
2. **import** java.lang.reflect.Parameter;
3. **public** **class** ParameterReflection {
4. **public** **static** **void** main(String[] args) {
5. // Creating object of a class
6. Calculate calculate = **new** Calculate();
7. // instantiating Class class
8. Classcls = calculate.getClass();
9. // Getting declared methods inside the Calculate class
10. Method[] method = cls.getDeclaredMethods(); // It returns array of methods
11. // Iterating method array
12. **for** (Method method2 : method) {
13. System.out.print(method2.getName());    // getting name of method
14. // Getting parameters of each method
15. Parameter parameter[] = method2.getParameters(); // It returns array of parameters
16. // Iterating parameter array
17. **for** (Parameter parameter2 : parameter) {
18. System.out.print(""+parameter2.getParameterizedType()); // returns type of parameter
19. System.out.print(""+parameter2.getName()); // returns parameter name
20. }
21. System.out.println();
22. }
23. }
24. }

Output:

mul int a int b

add int a int b

Above code will produce the below output if you don't use **-parameters** flag to compile the Calculate.java file.

Output:

mul int arg0 int arg1

add int arg0 int arg1

# Java Type and Repeating Annotations

## Java Type Annotations

Java 8 has included two new features repeating and type annotations in its prior annotations topic. In early Java versions, you can apply annotations only to declarations. After releasing of Java SE 8 , annotations can be applied to any type use. It means that annotations can be used anywhere you use a type. For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this:

1. @NonNull String str;

Following are the examples of type annotations:

1. @NonNull List<String>
2. List<@NonNull String> str
3. Arrays<@NonNegative Integer> sort
4. @Encrypted File file
5. @Open Connection connection
6. **void** divideInteger(**int** a, **int** b) **throws** @ZeroDivisor ArithmeticException

#### Note - Java created type annotations to support improved analysis of Java programs. It supports way of ensuring stronger type checking.

## Java Repeating Annotations

In Java 8 release, Java allows you to repeating annotations in your source code. It is helpful when you want to reuse annotation for the same class. You can repeat an annotation anywhere that you would use a standard annotation.

For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. In order for the compiler to do this, two declarations are required in your code.

1. Declare a repeatable annotation type
2. Declare the containing annotation type

### 1) Declare a repeatable annotation type

Declaring of repeatable annotation type must be marked with the @Repeatable meta-annotation. In the following example, we have defined a custom @Game repeatable annotation type.

1. @Repeatable(Games.**class**)
2. @interfaceGame{
3. String name();
4. String day();
5. }

The value of the @Repeatable meta-annotation, in parentheses, is the type of the container annotation that the Java compiler generates to store repeating annotations. In the following example, the containing annotation type is Games. So, repeating @Game annotations is stored in an @Games annotation.

### 2) Declare the containing annotation type

Containing annotation type must have a value element with an array type. The component type of the array type must be the repeatable annotation type. In the following example, we are declaring Games containing annotation type:

1. @interfaceGames{
2. Game[] value();
3. }

#### Note - Compiler will throw a compile-time error, if you apply the same annotation to a declaration without first declaring it as repeatable.

## Java Repeating Annotations Example

1. // Importing required packages for repeating annotation
2. **import** java.lang.annotation.Repeatable;
3. **import** java.lang.annotation.Retention;
4. **import** java.lang.annotation.RetentionPolicy;
5. // Declaring repeatable annotation type
6. @Repeatable(Games.**class**)
7. @interfaceGame{
8. String name();
9. String day();
10. }
11. // Declaring container for repeatable annotation type
12. @Retention(RetentionPolicy.RUNTIME)
13. @interfaceGames{
14. Game[] value();
15. }
16. // Repeating annotation
17. @Game(name = "Cricket",  day = "Sunday")
18. @Game(name = "Hockey",   day = "Friday")
19. @Game(name = "Football", day = "Saturday")
20. **public** **class** RepeatingAnnotationsExample {
21. **public** **static** **void** main(String[] args) {
22. // Getting annotation by type into an array
23. Game[] game = RepeatingAnnotationsExample.**class**.getAnnotationsByType(Game.**class**);
24. **for** (Gamegame2 : game) {    // Iterating values
25. System.out.println(game2.name()+" on "+game2.day());
26. }
27. }
28. }

OUTPUT:

Cricket on Sunday

Hockey on Friday

Football on Saturday

# Java 8 JDBC Improvements

In Java 8, Java made two major changes in JDBC API.

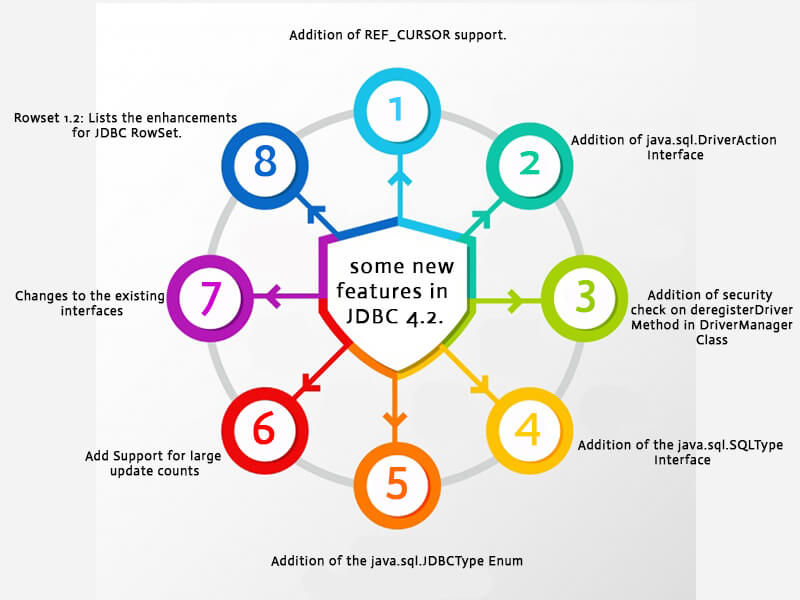
#### 1) The JDBC-ODBC Bridge has been removed.

Oracle does not support the JDBC-ODBC Bridge. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

#### 2) Added some new features in JDBC 4.2.

Java JDBC 4.2 introduces the following features:

* Addition of REF\_CURSOR support.
* Addition of java.sql.DriverAction Interface
* Addition of security check on deregisterDriver Method in DriverManager Class
* Addition of the java.sql.SQLType Interface
* Addition of the java.sql.JDBCType Enum
* Add Support for large update counts
* Changes to the existing interfaces
* Rowset 1.2: Lists the enhancements for JDBC RowSet.



## Java JDBC DriverAction

It is an interface that must be implemented when a Driver wants to be notified by DriverManager. It is added in java.sql package and contains only one abstract method.

### DriverAction Method

|  |  |
| --- | --- |
| **Method** | **Description** |
| void deregister() | This method called by DriverManager.deregisterDriver(Driver) to notify the JDBC driver that it was de-registered. |

The deregister method is intended only to be used by JDBC Drivers and not by applications.

JDBC drivers are recommended not to implement the DriverAction in a public class.

If there are active connections to the database at the time that the deregister method is called, it is implementation specific as to whether the connections are closed or allowed to continue. Once this method is called, it is implementation specific as to whether the driver may limit the ability to create new connections to the database, invoke other Driver methods or throw a SQLException.

## Java JDBC4.2 DriverAction Example

1. **import** java.sql.\*;
2. // implementing DriverAction interface
3. **class** JdbcExample **implements** DriverAction{
4. // implementing deregister method of DriverAction interface
5. @Override
6. **public** **void** deregister() {
7. System.out.println("Driver deregistered");
8. }
9. **public** **static** **void** main(String args[]){
10. **try**{
11. // Creating driver instance
12. Driver driver = **new** com.mysql.jdbc.Driver();
13. // Creating Action Driver
14. DriverAction da = **new** JdbcExample();
15. // Registering driver by passing driver and driverAction
16. DriverManager.registerDriver(driver, da);
17. // Creating connection
18. Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","mysql");
19. //Here student is database name, root is username and password is mysql
20. Statement stmt=con.createStatement();
21. // Executing SQL query
22. ResultSet rs=stmt.executeQuery("select \* from user");
23. **while**(rs.next()){
24. System.out.println(rs.getInt(1)+""+rs.getString(2)+""+rs.getString(3));
25. }
26. // Closing connection
27. con.close();
28. // Calling deregisterDriver method
29. DriverManager.deregisterDriver(driver);
30. }**catch**(Exception e){ System.out.println(e);}
31. }
33. }

Output:

1 Arun 25

2 irfan 22

3 Neraj kumar 25

Driver deregistered

## Java JDBC SQLType

This interface is used to identify a generic SQL type, JDBC type or a vendor specific data type.

It provides following methods.

|  |  |
| --- | --- |
| **Method** | **Description** |
| String getName() | It returns the SQLType name that represents a SQL data type. |
| String getVendor() | It returns the name of the vendor that supports this data type. The value returned typically is the package name for this vendor. |
| Integer getVendorTypeNumber() | It returns the vendor specific type number for the data type. |

## Java JDBCType

It is an Enumeration which defines the constants that are used to identify generic SQL types, called JDBC types. It extends java.lang.Enum and implements java.sql.SQLType.

## JDBCType Fields

The following table contains constants defined in the JDBCType.

|  |  |
| --- | --- |
| **Enum constant** | **Description** |
| public static final JDBCType ARRAY | It identifies the generic SQL type ARRAY. |
| public static final JDBCType BIGINT | It identifies the generic SQL type BIGINT. |
| public static final JDBCType BIT | It identifies the generic SQL type BIT. |
| public static final JDBCType BLOB | It identifies the generic SQL type BLOB. |
| public static final JDBCType BOOLEAN | It identifies the generic SQL type BOOLEAN. |
| public static final JDBCType CHAR | It identifies the generic SQL type CHAR. |
| public static final JDBCType CLOB | It identifies the generic SQL type CLOB. |
| public static final JDBCType DATALINK | It identifies the generic SQL type DATALINK. |
| public static final JDBCType DATE | It identifies the generic SQL type DATE. |
| public static final JDBCType DECIMAL | It identifies the generic SQL type DECIMAL. |
| public static final JDBCType DISTINCT | It identifies the generic SQL type DISTINCT. |
| public static final JDBCType DOUBLE | It identifies the generic SQL type DOUBLE. |
| public static final JDBCType FLOAT | It identifies the generic SQL type FLOAT. |
| public static final JDBCType INTEGER | It identifies the generic SQL type INTEGER. |
| public static final JDBCType JAVA\_OBJECT | It indicates that the SQL type is database-specific and gets mapped to a Java object that can be accessed via the methods getObject and setObject. |
| Public static final JDBCType LONGNVARCHAR | It identifies the generic SQL type LONGNVARCHAR. |
| public static final JDBCType NCHAR | It identifies the generic SQL type NCHAR. |
| public static final JDBCType NCLOB | It identifies the generic SQL type NCLOB. |
| public static final JDBCType NULL | It identifies the generic SQL value NULL. |
| public static final JDBCType NUMERIC | It identifies the generic SQL type NUMERIC. |
| public static final JDBCType NVARCHAR | It identifies the generic SQL type NVARCHAR. |
| public static final JDBCType OTHER | It indicates that the SQL type is database-specific and gets mapped to a Java object that can be accessed via the methods getObject and setObject. |
| public static final JDBCType REAL | It identifies the generic SQL type REAL.Identifies the generic SQL type VARCHAR. |
| public static final JDBCType REF | It identifies the generic SQL type REF. |
| public static final JDBCType REF\_CURSOR | It identifies the generic SQL type REF\_CURSOR. |
| public static final JDBCType ROWID | It identifies the SQL type ROWID. |
| public static final JDBCType SMALLINT | It identifies the generic SQL type SMALLINT. |
| public static final JDBCType SQLXML | It identifies the generic SQL type SQLXML. |
| public static final JDBCType STRUCT | It identifies the generic SQL type STRUCT. |
| public static final JDBCType TIME | It identifies the generic SQL type TIME. |
| public static final JDBCType TIME\_WITH\_TIMEZONE | It identifies the generic SQL type TIME\_WITH\_TIMEZONE. |
| public static final JDBCType TIMESTAMP | It identifies the generic SQL type TIMESTAMP. |
| public static final JDBCType TIMESTAMP\_WITH\_TIMEZONE | It identifies the generic SQL type TIMESTAMP\_WITH\_TIMEZONE. |
| public static final JDBCType TINYINT | It identifies the generic SQL type TINYINT. |
| public static final JDBCType VARBINARY | It identifies the generic SQL type VARBINARY. |
| public static final JDBCType VARCHAR | It identifies the generic SQL type VARCHAR. |

## JDBCType Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| public String getName() | It returns the SQLType name that represents a SQL data type. |
| public String getVendor() | It returns the name of the vendor that supports this data type. |
| public Integer getVendorTypeNumber() | It returns the vendor specific type number for the data type. |
| public static JDBCType valueOf(int type) | It returns the JDBCType that corresponds to the specified Types value. It throws IllegalArgumentException, if this enum type has no constant with the specified Types value. |
| public static JDBCType valueOf(String name) | It returns the enum constant of this type with the specified name. The string must match exactly an identifier used to declare an enum constant in this type. It throws IllegalArgumentException, if this enum type has no constant with the specified name. It throws NullPointerException, if the argument is null. |
| public static JDBCType[] values() | It returns an array containing the constants of this enum type, in the order they are declared. This method may be used to iterate over the constants. |

# Java Predicate Example – Predicate Filter

By Lokesh Gupta | Filed Under: [Java 8](https://howtodoinjava.com/java8/)

In mathematics, a **predicate** is commonly understood to be a **boolean-valued function** 'P: X? {true, false}', called the predicate on X. Informally, a strong. It can be thought of as an operator or function that returns a value that is either true or false.

## Java 8 Predicates Usage

In Java 8, [Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html) is a [functional interface](https://howtodoinjava.com/java8/functional-interface-tutorial/) and can therefore be used as the assignment target for a [**lambda expression**](https://howtodoinjava.com/java8/complete-lambda-expressions-tutorial-in-java/) or method reference. So, where you think, we can use these true/false returning functions in day to day programming? I will say you can use predicates anywhere where you need to evaluate a condition on group/collection of similar objects such that evaluation can result either in true or false.

For example, you can use prdicates in these **realtime usecases**

1. Find all children borned after a particular date
2. Pizzas ordered a specific time
3. Employees greater than certain age and so on..

#### Java Predicate Class

So **java predicates** seems to be interesting thing. Let’s go deeper.

As I said, Predicate is **functional interface**. It mean we can pass lambda expressions wherever predicate is expected. For example one such method is filter() method from [**Stream**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) interface.

|  |
| --- |
| /\*\*   \* Returns a stream consisting of the elements of this stream that match   \* the given predicate.   \*   \* <p>This is an <a href="package-summary.html#StreamOps">intermediate   \* operation</a>.   \*   \* @param predicate a non-interfering stateless predicate to apply to each element to determine if it   \* should be included in the new returned stream.   \* @return the new stream   \*/  Stream<T> filter(Predicate<? super T> predicate); |

We can assume a stream as a mechanism to create a sequence of elements supporting sequential and parallel aggregate operations. It means we can anytime collect and perform some operation of all elements present in the stream in one call.

So, essentially we can use stream and predicate to –

* first filter certain elements from a group, and
* then perform some operation on filtered elements.

## Using Predicate on a collection

To demonstrate, we have an Employee class as below:

|  |
| --- |
| Employee.java |
| package predicateExample;    public class Employee {       public Employee(Integer id, Integer age, String gender, String fName, String lName){         this.id = id;         this.age = age;         this.gender = gender;         this.firstName = fName;         this.lastName = lName;     }       private Integer id;     private Integer age;     private String gender;     private String firstName;     private String lastName;       //Please generate Getter and Setters       //To change body of generated methods, choose Tools | Templates.      @Override      public String toString() {          return this.id.toString()+" - "+this.age.toString();      }  } |

#### 1. All Employees who are male and age more than 21

|  |
| --- |
| public static Predicate<Employee> isAdultMale()  {      return p -> p.getAge() > 21 && p.getGender().equalsIgnoreCase("M");  } |

#### 2. All Employees who are female and age more than 18

|  |
| --- |
| public static Predicate<Employee> isAdultFemale()  {      return p -> p.getAge() > 18 && p.getGender().equalsIgnoreCase("F");  } |

#### 3. All Employees whose age is more than a given age

|  |
| --- |
| public static Predicate<Employee> isAgeMoreThan(Integer age)  {      return p -> p.getAge() > age;  } |

You can build more of them as an when needed. So far so good. Far using above methods I have included above 3 methods in **EmployeePredicates.java** :

|  |
| --- |
| EmployeePredicates.java |
| package predicateExample;    import java.util.List;  import java.util.function.Predicate;  import java.util.stream.Collectors;    public class EmployeePredicates  {      public static Predicate<Employee> isAdultMale() {          return p -> p.getAge() > 21 && p.getGender().equalsIgnoreCase("M");      }        public static Predicate<Employee> isAdultFemale() {          return p -> p.getAge() > 18 && p.getGender().equalsIgnoreCase("F");      }        public static Predicate<Employee> isAgeMoreThan(Integer age) {          return p -> p.getAge() > age;      }        public static List<Employee> filterEmployees (List<Employee> employees,                                                  Predicate<Employee> predicate)      {          return employees.stream()                      .filter( predicate )                      .collect(Collectors.<Employee>toList());      }  } |

You see I have created another utility method **filterEmployees()** to show **java predicate filter**. It is basically to make code clean and less repetitive. You can also write more than one predicates to make **predicate chain**, as we do in [builder pattern](https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/).

So, in this function we pass the list of employees and we pass a predicate, then this function will return a new collection of employees satisfying the condition mentioned in **parameter predicate**.

|  |
| --- |
| TestEmployeePredicates.java |
| package predicateExample;    import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  import static predicateExample.EmployeePredicates.\*;    public class TestEmployeePredicates  {      public static void main(String[] args)      {          Employee e1 = new Employee(1,23,"M","Rick","Beethovan");          Employee e2 = new Employee(2,13,"F","Martina","Hengis");          Employee e3 = new Employee(3,43,"M","Ricky","Martin");          Employee e4 = new Employee(4,26,"M","Jon","Lowman");          Employee e5 = new Employee(5,19,"F","Cristine","Maria");          Employee e6 = new Employee(6,15,"M","David","Feezor");          Employee e7 = new Employee(7,68,"F","Melissa","Roy");          Employee e8 = new Employee(8,79,"M","Alex","Gussin");          Employee e9 = new Employee(9,15,"F","Neetu","Singh");          Employee e10 = new Employee(10,45,"M","Naveen","Jain");            List<Employee> employees = new ArrayList<Employee>();          employees.addAll(Arrays.asList(new Employee[]{e1,e2,e3,e4,e5,e6,e7,e8,e9,e10}));            System.out.println( filterEmployees(employees, isAdultMale()) );            System.out.println( filterEmployees(employees, isAdultFemale()) );            System.out.println( filterEmployees(employees, isAgeMoreThan(35)) );            //Employees other than above collection of "isAgeMoreThan(35)"          //can be get using negate()          System.out.println(filterEmployees(employees, isAgeMoreThan(35).negate()));      }  }    Output:    [1 - 23, 3 - 43, 4 - 26, 8 - 79, 10 - 45]  [5 - 19, 7 - 68]  [3 - 43, 7 - 68, 8 - 79, 10 - 45]  [1 - 23, 2 - 13, 4 - 26, 5 - 19, 6 - 15, 9 - 15] |

Predicates are really very good addition in Java 8 and I am going to use it whenever I will get chance.

## Final Thoughts on Predicates in Java 8

1. They move your conditions (sometimes business logic) to a central place. This helps in unit-testing them separately.
2. Any change need not be duplicated into multiple places. Java predicate improves code maintenance.
3. The code e.g. “filterEmployees(employees, isAdultFemale())” is very much readable than writing a if-else block.

# Java 8 – Date and Time Examples

By Lokesh Gupta | Filed Under: [Java 8](https://howtodoinjava.com/java8/)

A big part of developer community has been complaining about **Date and Calendar classes**. Reasons were many such as hard to understand, hard to use and not flexible. Date class has even become obsolete and java docs suggest to use Calendar class instead of Date class. And on top of all, **Date comparison** is buggy and I have also faced such issue in past.



Moving forward, JAVA 8 ([Lambda](https://howtodoinjava.com/java8/complete-lambda-expressions-tutorial-in-java/)) is expected to release the new Date and Time APIs/classes ([**JSR-310**](https://java.net/projects/jsr-310/)), also called as **ThreeTen**, which will simply change the way you have been doing till date. This A key part of this is providing a new API that is dramatically easier to use and less error prone.

It will provide some highly demanded features such as:

* All the key public classes are immutable and thread-safe
* Defined terminology and behavior that other areas in computing can adopt

I wrote this post on 15th May 2013. Now today on 18th Mar 2014, java 8 is finally released and available for early access. I have re-validated and verified all the outputs in post examples. They work like charm as they did in May last year. Only change encountered was in TemporalAdjuster.java. Previously it was a class, now it is a **@FunctionalInterface**. So, I have corrected the related example and used the class “TemporalAdjusters.java“.

Table of Contents

New classes to represent local date and timezone

New classes to represent timestamp and duration

Added utility classes over existing enums

Date adjusters introduced

Building dates will be easier

New class to simulate system/machine clock

Timezone handling related changes

Date formatting changes

References

## New classes to represent local date and timezone

The new classes intended to replace Date class are LocalDate, LocalTime and LocalDateTime.

#### LocalDate

The [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html) class represents a date. There is no representation of a time or time-zone.

|  |
| --- |
| LocalDate localDate = LocalDate.now();  System.out.println(localDate.toString());                //2013-05-15  System.out.println(localDate.getDayOfWeek().toString()); //WEDNESDAY  System.out.println(localDate.getDayOfMonth());           //15  System.out.println(localDate.getDayOfYear());            //135  System.out.println(localDate.isLeapYear());              //false  System.out.println(localDate.plusDays(12).toString());   //2013-05-27 |

#### LocalTime

The [LocalTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html) class represents a time. There is no representation of a date or time-zone.

|  |
| --- |
| //LocalTime localTime = LocalTime.now();     //toString() in format 09:57:59.744  LocalTime localTime = LocalTime.of(12, 20);  System.out.println(localTime.toString());    //12:20  System.out.println(localTime.getHour());     //12  System.out.println(localTime.getMinute());   //20  System.out.println(localTime.getSecond());   //0  System.out.println(localTime.MIDNIGHT);      //00:00  System.out.println(localTime.NOON);          //12:00 |

#### LocalDateTime

The [LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) class represents a date-time. There is no representation of a time-zone.

|  |
| --- |
| LocalDateTime localDateTime = LocalDateTime.now();  System.out.println(localDateTime.toString());      //2013-05-15T10:01:14.911  System.out.println(localDateTime.getDayOfMonth()); //15  System.out.println(localDateTime.getHour());       //10  System.out.println(localDateTime.getNano());       //911000000 |

If you want to use the date functionality with zone information, then Lambda provide you extra 3 classes similar to above one i.e. OffsetDate, OffsetTime and OffsetDateTime. Timezone offset can be represented in “+05:30” or “Europe/Paris” formats. This is done via using another class i.e. ZoneId.

|  |
| --- |
| OffsetDateTime offsetDateTime = OffsetDateTime.now();  System.out.println(offsetDateTime.toString());              //2013-05-15T10:10:37.257+05:30    offsetDateTime = OffsetDateTime.now(ZoneId.of(&quot;+05:30&quot;));  System.out.println(offsetDateTime.toString());              //2013-05-15T10:10:37.258+05:30    offsetDateTime = OffsetDateTime.now(ZoneId.of(&quot;-06:30&quot;));  System.out.println(offsetDateTime.toString());              //2013-05-14T22:10:37.258-06:30    ZonedDateTime zonedDateTime =                  ZonedDateTime.now(ZoneId.of(&quot;Europe/Paris&quot;));  System.out.println(zonedDateTime.toString());               //2013-05-15T06:45:45.290+02:00[Europe/Paris] |

## New classes to represent timestamp and duration

#### Instant

For representing the specific timestamp ant any moment, the class needs to be used is [Instant](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html). The Instant class represents an instant in time to an accuracy of nanoseconds. Operations on an Instant include comparison to another Instant and adding or subtracting a duration.

|  |
| --- |
| Instant instant = Instant.now();  System.out.println(instant.toString());                                 //2013-05-15T05:20:08.145Z  System.out.println(instant.plus(Duration.ofMillis(5000)).toString());   //2013-05-15T05:20:13.145Z  System.out.println(instant.minus(Duration.ofMillis(5000)).toString());  //2013-05-15T05:20:03.145Z  System.out.println(instant.minusSeconds(10).toString());                //2013-05-15T05:19:58.145Z |

#### Duration

[Duration](https://docs.oracle.com/javase/8/docs/api/java/time/Duration.html) class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

|  |
| --- |
| Duration duration = Duration.ofMillis(5000);  System.out.println(duration.toString());     //PT5S    duration = Duration.ofSeconds(60);  System.out.println(duration.toString());     //PT1M    duration = Duration.ofMinutes(10);  System.out.println(duration.toString());     //PT10M    duration = Duration.ofHours(2);  System.out.println(duration.toString());     //PT2H    duration = Duration.between(Instant.now(), Instant.now().plus(Duration.ofMinutes(10)));  System.out.println(duration.toString());  //PT10M |

Duration deals with small unit of time such as milliseconds, seconds, minutes and hour. They are more suitable for interacting with application code.

#### Period

To interact with human, you need to get bigger durations which are presented with [Period](https://docs.oracle.com/javase/8/docs/api/java/time/Period.html) class.

|  |
| --- |
| Period period = Period.ofDays(6);  System.out.println(period.toString());    //P6D    period = Period.ofMonths(6);  System.out.println(period.toString());    //P6M    period = Period.between(LocalDate.now(),              LocalDate.now().plusDays(60));  System.out.println(period.toString());   //P1M29D |

## Added utility classes over existing enums

The current Java SE platform uses int constants for months, day-of-week and am-pm etc. Now a lot of extra utility classes have been added which work on top of these enums. I am taking an example such a class [DayOfWeek](https://docs.oracle.com/javase/8/docs/api/java/time/DayOfWeek.html). This class is a wrapper of day enums and can be used consistently with other classes also.

#### DayOfWeek

|  |
| --- |
| //day-of-week to represent, from 1 (Monday) to 7 (Sunday)  System.out.println(DayOfWeek.of(2));                    //TUESDAY    DayOfWeek day = DayOfWeek.FRIDAY;  System.out.println(day.getValue());                     //5    LocalDate localDate = LocalDate.now();  System.out.println(localDate.with(DayOfWeek.MONDAY));  //2013-05-13  i.e. when was monday in current week ? |

Other such classes are Month, MonthDay, Year, YearMonth and many more.

## Date adjusters

Date adjusters are another beautiful and useful addition in date handling tools. It easily solves the problems like : How do you find last day of the month? Or the next working day? Or a week on Tuesday?

Lets see in code.

|  |
| --- |
| LocalDate date = LocalDate.of(2013, Month.MAY, 15);                     //Today    LocalDate endOfMonth = date.with(TemporalAdjusters.lastDayOfMonth());  System.out.println(endOfMonth.toString());                              //2013-05-31    LocalDate nextTue = date.with(TemporalAdjusters.next(DayOfWeek.TUESDAY));  System.out.println(nextTue.toString());                                 //2013-05-21 |

## Creating date objects

Creating date objects now can be done using [builder pattern](https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/) also. The builder pattern allows the object you want to be built up using individual parts. This is achieved using the methods prefixed by “at”.

|  |
| --- |
| //Builder pattern used to make date object   OffsetDateTime date1 = Year.of(2013)                          .atMonth(Month.MAY).atDay(15)                          .atTime(0, 0)                          .atOffset(ZoneOffset.of(&quot;+03:00&quot;));   System.out.println(date1);                                     //2013-05-15T00:00+03:00    //factory method used to make date object  OffsetDateTime date2 = OffsetDateTime.                          of(2013, 5, 15, 0, 0, 0, 0, ZoneOffset.of(&quot;+03:00&quot;));  System.out.println(date2);                                      //2013-05-15T00:00+03:00 |

## New class to simulate system/machine clock

A new class [Clock](https://docs.oracle.com/javase/8/docs/api/java/time/Clock.html) is proposed in new release. This **simulates the system clock functionality**. I loved this feature most of all others. The reason is while doing unit testing. you are often required to test a API in future date. For this we had been forwarding the system clock for next date, and then again restart the server and test the application.

Now, no need to do this. Use Clock class to simulate this scenario.

|  |
| --- |
| Clock clock = Clock.systemDefaultZone();  System.out.println(clock);                      //SystemClock[Asia/Calcutta]  System.out.println(clock.instant().toString()); //2013-05-15T06:36:33.837Z  System.out.println(clock.getZone());            //Asia/Calcutta    Clock anotherClock = Clock.system(ZoneId.of(&quot;Europe/Tiraspol&quot;));  System.out.println(anotherClock);                       //SystemClock[Europe/Tiraspol]  System.out.println(anotherClock.instant().toString());  //2013-05-15T06:36:33.857Z  System.out.println(anotherClock.getZone());             //Europe/Tiraspol    Clock forwardedClock  = Clock.tick(anotherClock, Duration.ofSeconds(600));  System.out.println(forwardedClock.instant().toString());  //2013-05-15T06:30Z |

## Timezone Changes

Timezone related handling is done by 3 major classes. These are [ZoneOffset](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneOffset.html), [TimeZone](https://docs.oracle.com/javase/8/docs/api/java/util/TimeZone.html), [ZoneRules](https://docs.oracle.com/javase/8/docs/api/java/time/zone/ZoneRules.html).

* The ZoneOffset class represents a fixed offset from UTC in seconds. This is normally represented as a string of the format “±hh:mm”.
* The TimeZone class represents the identifier for a region where specified time zone rules are defined.
* The ZoneRules are the actual set of rules that define when the zone-offset changes.

|  |
| --- |
| //Zone rules  System.out.println(ZoneRules.of(ZoneOffset.of(&quot;+02:00&quot;)).isDaylightSavings(Instant.now()));  System.out.println(ZoneRules.of(ZoneOffset.of(&quot;+02:00&quot;)).isFixedOffset()); |

## Date Formatting

Date formatting is supported via two classes mainly i.e. DateTimeFormatterBuilder and DateTimeFormatter. DateTimeFormatterBuilder works on builder pattern to build custom patterns where as DateTimeFormatter provides necessary input in doing so.

|  |
| --- |
| DateTimeFormatterBuilder formatterBuilder = new DateTimeFormatterBuilder();  formatterBuilder.append(DateTimeFormatter.ISO\_LOCAL\_DATE\_TIME)                  .appendLiteral(&quot;-&quot;)                  .appendZoneOrOffsetId();  DateTimeFormatter formatter = formatterBuilder.toFormatter();  System.out.println(formatter.format(ZonedDateTime.now())); |

These are major changes which I was able to identify and worked on.