**Table Of Contents**

show

* [What is Java 8?](https://www.interviewbit.com/blog/java-8-features/#what-is-java-8)
* [Top Java 8 Features With Examples](https://www.interviewbit.com/blog/java-8-features/#top-java-8-features-with-examples)
  + [Functional Interfaces And Lambda Expressions](https://www.interviewbit.com/blog/java-8-features/#functional-interfaces-and-lambda-expressions)
  + [forEach() Method In Iterable Interface](https://www.interviewbit.com/blog/java-8-features/#foreach-method-in-iterable-interface)
  + [Optional Class](https://www.interviewbit.com/blog/java-8-features/#optional-class)
  + [Default And Static Methods In Interfaces](https://www.interviewbit.com/blog/java-8-features/#default-and-static-methods-in-interfaces)
  + [Java Stream API For Bulk Data Operations On Collections](https://www.interviewbit.com/blog/java-8-features/#java-stream-api-for-bulk-data-operations-on-collections)
  + [Java Date Time API](https://www.interviewbit.com/blog/java-8-features/#java-date-time-api)
  + [Collection API Improvements](https://www.interviewbit.com/blog/java-8-features/#collection-api-improvements)
  + [Java IO Improvements](https://www.interviewbit.com/blog/java-8-features/#java-io-improvements)
  + [Miscellaneous Core API Improvements](https://www.interviewbit.com/blog/java-8-features/#miscellaneous-core-api-improvements)
  + [Base64 Encode Decode](https://www.interviewbit.com/blog/java-8-features/#base64-encode-decode-)
* [Conclusion](https://www.interviewbit.com/blog/java-8-features/#conclusion)
* [FAQs](https://www.interviewbit.com/blog/java-8-features/#faqs)
* [Additional Resources](https://www.interviewbit.com/blog/java-8-features/#additional-resources)

On March 18, 2014, Java 8 was released. Even though it was a long time ago, Java 8 is still used in many applications because Java 8 got a significant update with several new features. Let’s take a look at all of Java 8 features using sample code.

**What is Java 8?**

Oracle’s Java 8 release was a watershed moment in the history of the world’s most popular development platform. In addition to a significant improvement to the Java programming paradigm as a whole, the JVM, Java language, and libraries were all updated in a coordinated manner. Many new features were included in this update, including enhanced simplicity of use, increased productivity and security, and overall better system performance.

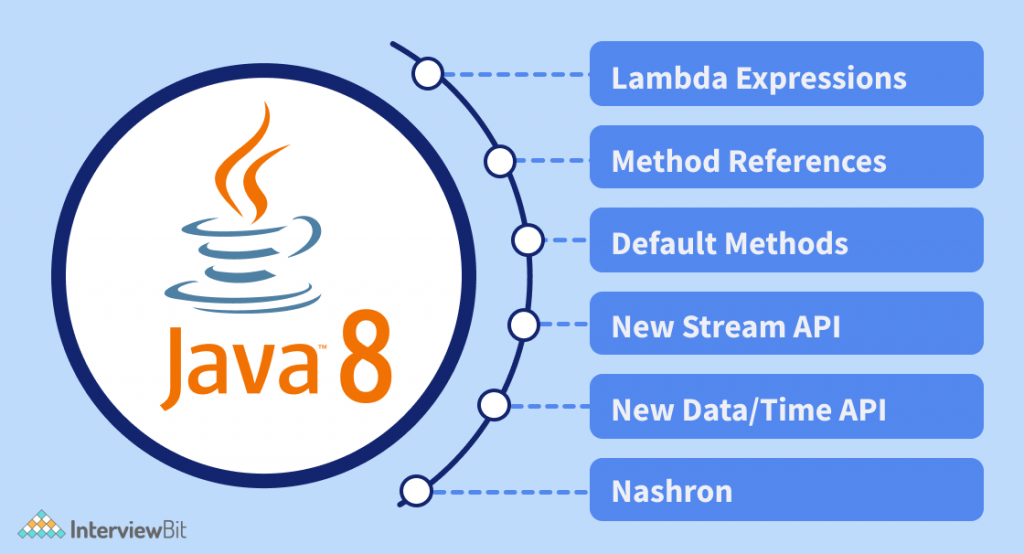
**Top Java 8 Features With Examples**

The following Java 8 features will be discussed briefly, with each being explained via the use of basic and simple examples.

**Confused about your next job?**

In 3 simple steps you can find your personalised career roadmap in Software development for FREE

[Expand in New Tab](https://www.scaler.com/career-plan/embedded?utm_source=ib&utm_medium=blog-embed&utm_campaign=career-tool)



**Functional Interfaces And Lambda Expressions**

In Java 8, a new notion called functional interfaces was introduced. A Functional Interface is an interface that has exactly one abstract method. To designate an interface as a Functional Interface, we don’t need to use the @FunctionalInterface annotation.

The @FunctionalInterface annotation prevents abstract methods from being accidentally added to functional interfaces. It’s similar to a @Override annotation, and it’s recommended that you use it. java.lang. Runnable is a fantastic example of a functional interface since it has one abstract method, run ().

One of the most appealing features of the functional interface is creating objects using lambda expressions. We can create an interface using an anonymous class, but the code is cumbersome. For example:

@FunctionalInterface

public interface FunctionalInterface\_one

{

public void firstInt\_method();

@Override

public String toString(); //Overridden from Object class

@Override

public boolean equals(Object obj); //Overridden from Object class

}

An anonymous function may be defined as a Lambda Expression (or function) (a function with no name and an identifier). Lambda Expressions are defined precisely where they are required, often as a parameter to another function.

Lambda Expressions, on the other hand, express instances of Functional Interfaces from a different viewpoint. Lambda Expressions implement functional interfaces by implementing the single abstract function provided in the functional interface.

A basic example of the Lambda Expression is:  
**(x,y) -> x+y**

The formula above accepts two parameters, x, and y, and returns their total, x+y. The procedure may be used numerous times in different locations depending on the data type of x and y. As a result, the arguments x and y will match int or Integer and string, and depending on the context; it will either add two numbers (if the parameters are int) or concatenate the two strings (when parameters are a string).

Let’s implement a program that demonstrates Lambda Expressions.

interface MyInterface

{ void abstract\_func(int x,int y);

default void default\_Fun()

{ System.out.println("This is default method");

}}

class Main

{ public static void main(String args[])

{ //lambda expression

MyInterface fobj = (int x, int y)->System.out.println(x+y);

System.out.print("The result = ");

fobj.abstract\_func(5,5);

fobj.default\_Fun();

}

}

**forEach() Method In Iterable Interface**

In Java 8, the Java.lang interface now supports a “forEach” function. Iterable that can iterate over the collection’s items. The Iterable interface has a default method called “forEach.” Collection classes use it to iterate items, which extends the Iterable interface.You may send Lambda Expression as an argument to the “forEach” method, which accepts the Functional Interface as a single parameter.

importjava.util.ArrayList;

importjava.util.List;

public class Main {

public static void main(String[] args) {

List<String> subList = new ArrayList<String>();

subList.add("Carrot");

subList.add("Potato");

subList.add("Cauliflower");

subList.add("LadyFinger");

subList.add("Tomato");

System.out.println("------------Vegetable List--------------");

subList.forEach(sub -> System.out.println(sub));

}

}

**Output:**  
————Vegetable List————–  
Carrot  
Potato  
Cauliflower  
LadyFinger  
Tomato

**Optional Class**

In Java 8, the “java.util” package included an optional class. The public final class “Optional” is used to handle NullPointerException in a Java program. You may give other code or values to execute using Optional. Thus, optional reduces the number of null checks required to avoid a nullPointerException.

You may use the Optional class to prevent the application from crashing and terminating unexpectedly. The Optional class has methods for checking the existence of a value for a given variable.

The following program demonstrates the use of the Optional class.

import java.util.Optional;

public class Main{

public static void main(String[] args) {

String[] str = new String[10];

Optional<String>checkNull =

Optional.ofNullable(str[5]);

if (checkNull.isPresent()) {

String word = str[5].toLowerCase();

System.out.print(str);

} else

System.out.println("string is null");

}

}

**Output:**  
String is null

To verify whether the string is null in this application, we utilize the Optional class’s “ofNullable” attribute. If it is, the user receives the relevant message.

**Default And Static Methods In Interfaces**

In Java 8, you may add non-abstract methods to interfaces, allowing you to create interfaces with method implementation. To construct interfaces with method implementation, use the Default and Static keywords. Lambda Expression functionality is mostly enabled through default approaches.

You may extend the functionality of your libraries’ interfaces by using default methods. This ensures that the code created for previous versions is compatible with the newer interfaces (binary compatibility).

Let’s understand the Default Method with an example:

import java.util.Optional;

interface interface\_default {

default void default\_method(){

System.out.println("We are default method of interface");

}}

class derived\_class implements interface\_default{}

class Main{ public static void main(String[] args){

derived\_class obj1 = new derived\_class();

obj1.default\_method();

}

}

**Output:**  
We are the default method of interface

We have an interface called “interface default” with a default implementation of the function default method(). Next, we create a class called “derived class” that implements the “interface default” interface.

In this class, we haven’t implemented any interface functions. Then, in the main function, we construct a “derived class” object and invoke the interface’s “default method” without specifying it in the class.

The usage of default and static methods in the interface is an example of this. If a class wishes to alter the default method, it may override it and give its own implementation.

**Java Stream API For Bulk Data Operations On Collections**

Another significant feature in Java 8 is the Stream API. The Stream API is used to handle a collection of items and allows many iterations. A Stream is a collection of items (elements) that enables you to combine multiple techniques to achieve your goals.

The java.util.stream package in Java 8 introduces a new Streams API that enables you to process components of Java Collections in parallel. Because Java is inherently sequential, and there is no direct way to implement parallel processing at the library level, the stream API will fill that void. You may use Java’s Stream API to filter items of a collection based on a defined condition. If you have a list of orders, for example, you may combine purchase and sell orders, filter orders by amount and price, and so on.

**Java Date Time API**

Under the package java.time, Java 8 offers a new date-time API. The following are the most prominent classes among them:

* Local: Simplified date-time API with no timezone management complexity.
* Zoned: specialized date-time API that can handle several time zones.

For Example:

| **Class** | **Description** |
| --- | --- |
| LocalDate | Represents a date (year, month, day (yyyy-MM-dd)) |
| LocalTime | Represents a time (hour, minute, second and nanoseconds (HH-mm-ss-ns)) |
| LocalDateTime | Represents both a date and a time (yyyy-MM-dd-HH-mm-ss-ns) |
| DateTimeFormatter | Formatter for displaying and parsing date-time objects |

**Dates**

In Java 8, the Date class has been deprecated. The following are the new courses that have been introduced:

* A date is defined by the LocalDate class. It is devoid of any indication of time or time zone.
* A time is defined by the LocalTime class. It doesn’t have a date or time-zone representation.
* A date-time is defined by the LocalDateTime class. It doesn’t have a time-zone representation

To combine time-zone information with date functions, you may utilize Lambda’s OffsetDate, OffsetTime, and OffsetDateTime classes. Another class – “ZoneId” – is used to express the timezone offset here.

**Collection API Improvements**

The Collection API in Java 8 now includes the following new methods. A few of them are listed below.

* This is a default method for the Iterator. forEachRemaining (Consumer action): This is a default method for the Iterator. It repeats the “action” for the remaining items until all of them have been processed, or the “action” throws an exception.
* The default technique for removing items from a collection is removed (Predicate filter). This removes all objects from the collection that satisfy the supplied “filter.”
* Spliterator () This collection method returns a spliterator object that may be used to traverse the items sequentially or in parallel.
* ReplaceAll (), calculate(), and merge() are methods in the Map collection.
* The performance of the HashMap class with Key collisions has been enhanced etc.

**Java IO Improvements**

The following are some of the IO enhancements made in Java 8:

* Files.list (Path dir): Returns a lazily filled stream, each element of which represents a directory entry.
* Files.lines (Path path): Reads all the lines from a stream.
* Files.find (): Returns a stream filled by a path after searching for files in the file tree rooted at a provided beginning file and many more.
* BufferedReader.lines (): Returns a stream containing all of the elements of BufferedReader’s lines and much more
* Buffered

**Miscellaneous Core API Improvements**

* ThreadLocal’s static function with initial (Supplier supplier) allows you to build an instance quickly.
* The default and static methods for natural ordering, reverse order, and other operations have been added to the “Comparator” interface.
* The min (), max (), and sum () methods are available in the Integer, Long, and Double wrapper classes.
* The logicalAnd (), logicalOr (), and logicalXor () methods have been added to the Boolean class.
* The Math class introduces a number of useful techniques.
* The JDBC-ODBC Bridge has been deactivated.
* The memory space used by PermGen is no longer available.

**Base64 Encode Decode**

For Base64 encoding, Java 8 has built-in encode and decode functions. The Base64 encoding class in Java.util.Base64.

Three Base64 encoders and decoders are provided in this class:

* The output is mapped to a set of characters between A-Za-z0-9+/ in this version. The encoder does not add a line feed to the output, and the decoder rejects any character other than the above.
* The filename safe is mapped to the set of characters between A-Za-z0-9+/, and the output is the URL.
* The output of this sort of encoder is mapped to a MIME-friendly format.

**Conclusion**

As we can see, Java 8 offers a number of important features. Other programming languages, such as Scala, have inspired some of the features. Java must improve in order to avoid becoming outdated, and although certain features are rudimentary in comparison to other programming languages.

**FAQs**

**Q1: What are the characteristics of Java 8?**  
**Ans:**Following are the characteristics of Java 8:

* Lambda expressions
* Method references
* Functional interfaces
* Stream API
* Default methods
* Base64 Encode Decode
* Static methods in interface
* Optional class and many more

**Q2: Why is Java 8 still popular?**  
**Ans:** The fact that Java 8 is an LTS (Long-Term Support) version is one of the main reasons for its continued popularity. Regrettably, not all Java versions are LTS versions! Only Java 8 (2014) and Java 11 (2018) have been recognized as having LTS since the policy was implemented. All subsequent versions, including Java 14 and the anticipated Java 15 (September 2020), will be LTS-free.

**Q3: What changed in Java 8?**  
**Ans:** Some of the significant productivity enhancements are lambda expressions, the Streams API, and new methods on existing classes.