Java 11 features with examples

**Java 11 features**

Java is one of the most popular programming languages widely used for building robust and scalable software applications. It released many important features, developers have access to a wide range of new features and improvements that can help them write better code with greater efficiency.

Oracle released Java 11 in September 2018. It was the second long-term support (LTS) version of Java. It means that supports bug fixes and security updates for a more extended period, making it a stable and reliable choice for developers.

Overall, **Java 11** offers a wide range of features and enhancements that make it an exciting choice for developers looking to build modern, scalable applications.

**Oracle vs. Open JDK**

**Oracle JDK**and **OpenJDK** have two different distributions of the**Java Development Kit (JDK)** and **Java Runtime Environment (JRE).** Both used by developers widely used by developers.

**Oracle JDK** is a proprietary distribution of Java that is developed and maintained by Oracle Corporation. It includes additional features, such as Java Flight Recorder, Java Mission Control, and JavaFX, which are not available in the OpenJDK distribution.

The Java community develops and maintains OpenJDK, it’s a free and open-source distribution of Java. It is licensed under GNU, which allows users to use, modify, and distribute it freely. OpenJDK serves as the reference implementation for the Java language and most Linux distributions use it as their default JDK.

The main difference between the two is that Oracle JDK includes some additional features and comes with a commercial license, while OpenJDK is free and open-source.

In practice, the choice between Oracle JDK and OpenJDK often depends on the specific needs of the project or organization. If cost or licensing considerations are a concern, then OpenJDK may be the better choice.

**Let’s Discuss all features of java 11 one by one**

**Java 11** introduced several new developer features that can help improve the developer experience and productivity.

**Developer features in Java 11**

[**New String Methods in java**](https://javagoal.com/java-11-string-new-methods/#1)**:**

[**New String Methods in java**](https://javagoal.com/java-11-string-new-methods/#1)**:**The **string class**has a lot of methods to deal with [**String in java**](https://javagoal.com/string-in-java/) but **Java 11** introduces several new methods into the [String class](https://javagoal.com/string-in-java/#1), such as **isBlank()**, **strip()**, **stripLeading()**, **stripTrailing(),** and r**epeat(int count)**

[**Local variable type inference**](https://javagoal.com/java-11-features/#1)

To make it a better feature, Java 11 introduces the **“var” keyword** for **local variable type inference**, which allows developers to declare variables without explicitly specifying the type.

[**New File Methods**](https://javagoal.com/java-11-features/#3)

To provide more flexibility to **file methods**, Java 11 introduced new methods for reading and writing files using the readString() and writeString() methods, which can simplify file I/O operations.

[**Collection to Array**](https://javagoal.com/java-11-features/#2)

Java 11 introduces a new method **toArray(IntFunction<T[]> generator)** in the [**Collection interface**](https://javagoal.com/collection-in-java/#5), which allows collections to be easily converted into arrays

[**Predicate Not Method**](https://javagoal.com/java-11-features/#4)

Java 11 introduced a new method **“not()”** in the [**Predicate interface**](https://javagoal.com/java-8-predicate/) that returns a new predicate that represents the negation of the original predicate. This simplifies code that requires negating a predicate.

**HTTP Client API**

Java 11 introduced a new HTTP Client API that makes it easier for developers to send HTTP requests and handle responses asynchronously.

**Nest-Based Access Control**

Java 11 introduces Nest-Based Access Control, which allows classes that are logically part of the same code entity to access each other’s private members without the need for reflection.

**Enhanced Unicode Support**

Java 11 includes several enhancements to Unicode support, such as support for Unicode 10.0.0, additional scripts, and the ability to handle non-BMP Unicode characters.

**Flight Recorder**

Java 11 contains Java Flight Recorder, which is a profiling and diagnostic tool for collecting and analyzing runtime information about the JVM and the Java applications running on it.

**Low Overhead Heap Profiling**

Java 11 includes a new low-overhead heap profiling feature that can help developers diagnose memory-related issues in their applications more easily.

**Dynamic Class-File Constants**

Java 11 introduces a new constant-pool form, “Dynamic”, that allows constants to be dynamically computed at runtime.

**Epsilon**

Java 11 includes a new experimental garbage collector called Epsilon, which designes to be a no-op collector that simply allocates and discards memory without performing any actual garbage collection.

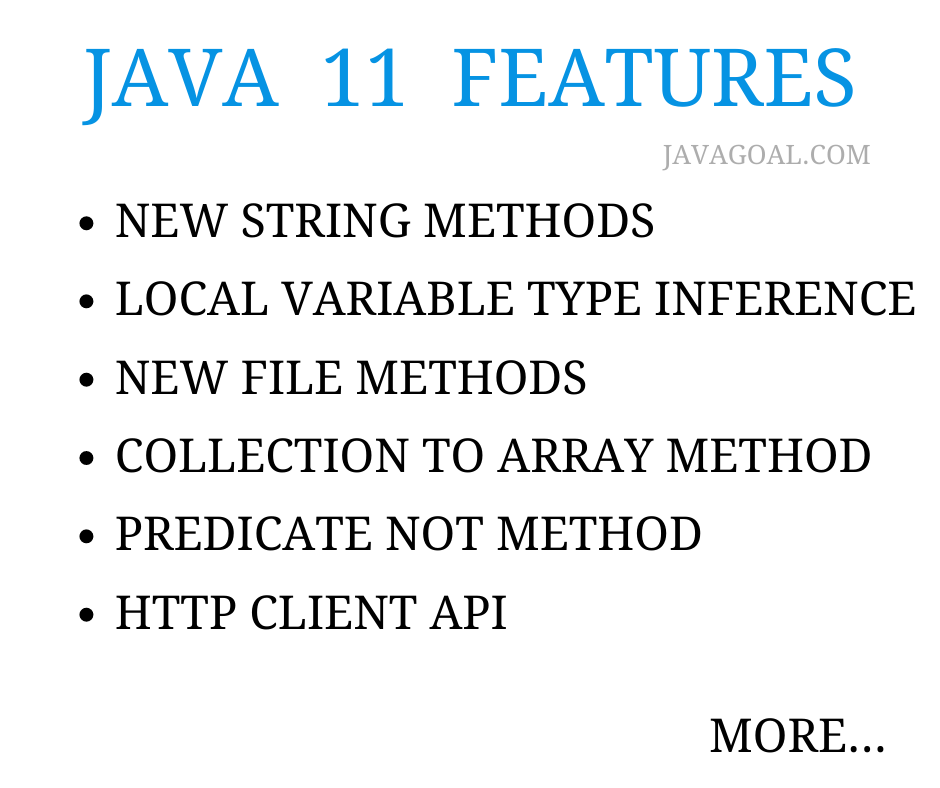
**ZGC**

Java 11 introduces a new experimental garbage collector called ZGC, which is designed to be a scalable, low-latency collector that can handle heaps ranging from a few hundred megabytes to several terabytes in size.

**ChaCha20 and Poly1305 Cryptographic Algorithms**

Java 11 supports the ChaCha20 and Poly1305 cryptographic algorithms, which can provide faster and more secure encryption and decryption than some of the older algorithms.

**Launch Single-File Source-Code Programs**



**Local Variable Type Inference**

Local Variable Type Inference was introduced in java 10. This feature allows developers to declare and initialize local variables without explicitly stating their data type. But there was some improvement area. So **Java 11** continued to support this feature and introduced some improvements to make it more usable. In this post, we will see these improvements with examples.

**Local variable type inference for lambda expressions in java 11**

In java, the [**lambda expressions**](https://javagoal.com/java-8-lambda-expressions/) feature was introduced in [**java 8**](https://javagoal.com/java-8-tutorial/) that allows developers to write functional-style code. But we couldn’t use [**var**](https://javagoal.com/local-variable-type-inference-java-var/) as the parameter type for lambda expressions before java 11. After java 11, the compiler can identify the type of a lambda expression parameter based on its context. So now we can use var as the parameter type for lambda expressions.

Let’s take an example and use a lambda expression to sort a list of integers in descending order. First of all, we will see it with java 10 and then move the code to java 11.

import java.util.Arrays;

import java.util.List;

public **class** VarInLambda {

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

List<Integer> numbers = Arrays.asList(3, 5, 2, 6, 1, 4);

numbers.sort((Integer a, Integer b) -> b.compareTo(a));

System.out.println(numbers);

}

}

***Output:****[6, 5, 4, 3, 2, 1]*

In the above code, we used the **Integer type** parameters for the lambda expression. Now let’s see it with the enhanced inference in Java 11, the parameter types can be omitted and replaced with var:

import java.util.Arrays;

import java.util.List;

public **class** VarInLambda {

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

List<Integer> numbers = Arrays.asList(3, 5, 2, 6, 1, 4);

numbers.sort((var a, var b) -> b.compareTo(a));

System.out.println(numbers);

}

}

***Output:****[6, 5, 4, 3, 2, 1]*

**Using var with Anonymous Inner Classes**

Java 11 did one more improvement in local variable type inference. Now developers can use the **var keyword with anonymous inner classes**. Before java 11, anonymous inner classes required an explicit type declaration for their variables, which could make the code more verbose and harder to read. With the ability to use var in anonymous inner classes, developers can now write more concise and readable code.

Let’s take an example of anonymous class.

Runnable task = **new** Runnable() {

@Override

public **void** run() {

System.out.println("Task executed");

}

};

In the above code, We used to type with an anonymous inner class that implements the Runnable interface. The type declaration for the variable can be omitted:

var task = **new** Runnable() {

@Override

public **void** run() {

System.out.println("Task executed");

}

};

**Converting a Collection to an Array**

Java 11, introduces a new method in the **Collection interface** i.e. **toArray(IntFunction generator) method.** This method uses to convert a collection into an array. In this section, we will read about **Converting a Collection to an Array in java 11**

The **toArray() method** has been a part of the Collection interface since Java 1.2. But Java 11 provides a new method that allows for more control over the returned array. So now the **toArray() method**is an overloaded method. The method introduced in java 11 takes an **IntFunction** as an argument, which specifies the type of the returned array.

default <T> T[] toArray(IntFunction<T[]> generator)

It’s a **default method** that takes only one parameter. The **IntFunction**is a **functional interface** that takes an integer value and returns an array of type T[]. It is useful where the size of the collection is known beforehand. because it provides more control over the returned array size.

import java.util.ArrayList;

import java.util.List;

public **class** CollectionToArrayExample {

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

List<String> list = **new** ArrayList<>();

list.add("apple");

list.add("banana");

list.add("orange");

String[] array = list.stream().toArray(String[]::**new**);

**for** (String s : array) {

System.out.println(s);

}

}

}

**Output:** apple  
banana  
orange

**How was this type of task performed before Java 11?**

Object[] toArray()

<T> T[] toArray(T[] a)

The 1st overload method (Object[] toArray()) returns an array of type Object[] that contains all of the elements in the collection.

The second overload method (T[] toArray(T[] a)) took an array of type T[] as an argument, and returns that same array if it was large enough to hold all of the elements in the collection, or a new array of the same type if it was not.

import java.util.ArrayList;

import java.util.List;

public **class** CollectionToArrayExample {

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

List<String> list = **new** ArrayList<>();

list.add("apple");

list.add("banana");

list.add("orange");

String[] array = **new** String[list.size()];

array = list.toArray(array);

**for** (String s : array) {

System.out.println(s);

}

}

}

***Output:****apple  
banana  
orange*

**Java 11 Files New Methods**

Java 11 introduced some new methods in the Files class and these methods provide better handling. These methods are:

1. **readString(Path path, Charset cs) method**
2. **writeString(Path path, CharSequence csq, OpenOption… options) method**

**readString(Path path, Charset cs) method**

The **readString(Path path, Charset cs) method** exists in the **java.nio.file.Files** class in **Java 11**. It uses to read the content from a file into a string using the specified charset. Let’s try to explain it with some scenarios where we can use it.

**Scenario 1: Reading a Text File**Suppose we want to read a text file that contains data and process it in a java application.

import java.nio.charset.StandardCharsets;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.io.IOException;

public **class** FileReader {

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

Path filePath = Paths.get("data.txt");

String fileContent = Files.readString(filePath, StandardCharsets.UTF\_8);

System.out.println(fileContent);

}

}

Here we create a **Path object** that represents the file we want to read. After that we can use the **readString() method** to read the content of the file into a string using the UTF-8 charset.

**Scenario 2: Parsing a CSV File**  
Let’s try to read the **CSV file** that you need to parse in your Java application.

import java.nio.charset.StandardCharsets;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.io.IOException;

public **class** CsvParser {

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

Path filePath = Paths.get("data.csv");

String fileContent = Files.readString(filePath, StandardCharsets.UTF\_8);

String[] rows = fileContent.split("\n");

**for** (String row : rows) {

String[] columns = row.split(",");

System.out.println("Name: " + columns[0] + ", Age: " + columns[1]);

}

}

}

We used the **readString() method** to read the CSV file into a string using the UTF-8 charset. By use of the [**split() method**](https://javagoal.com/string-methods-in-java/#13)the string is into rows using the newline character as the separator and split each row into columns using the comma character as the separator.

**writeString(Path path, CharSequence csq, OpenOption… options)**

The **writeString(Path path, CharSequence csq, OpenOption… options) method** was introduced in the **java.nio.file.Files class** in **Java 11**. It writes a string to a file using the specified charset and options.

**Scenario 1: Writing a Text File**  
When we want to write to a text file in your Java application. Then we can use the **writeString() method** to write a string to a file.

import java.nio.charset.StandardCharsets;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.io.IOException;

public **class** FileWriter {

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

Path filePath = Paths.get("data.txt");

String fileContent = "Hello, world!";

Files.writeString(filePath, fileContent, StandardCharsets.UTF\_8);

}

}

**Scenario 2: Appending to a Text File**  
Let’s try to read a text file that already contains some data. After that, we want to append some more data to the end of the file in your Java application.

import java.nio.charset.StandardCharsets;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.io.IOException;

import java.nio.file.StandardOpenOption;

public **class** FileAppender {

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

Path filePath = Paths.get("data.txt");

String fileContent = "World!";

Files.writeString(filePath, fileContent, StandardCharsets.UTF\_8, StandardOpenOption.APPEND);

}

}

**How was this type of task performed before Java 11?**

Let’s see how we read the file before java 11:

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

public **class** FileReaderExample {

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

BufferedReader reader = **null**;

**try** {

reader = **new** BufferedReader(**new** FileReader("data.txt"));

String line = **null**;

**while** ((line = reader.readLine()) != **null**) {

System.out.println(line);

}

} **catch** (IOException e) {

e.printStackTrace();

} finally {

**try** {

**if** (reader != **null**) {

reader.close();

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

The **readString() method** allows us to read the entire contents of a file into a String with a single method call. It’s much simpler and more convenient than **BufferedReader**to read the file line by line.

**Advantages of readString() method over the BufferedReader approach**

**Simplicity**: The **readString() method** makes the code simpler and easier to read because we don’t need to deal with a BufferedReader and a loop.

**Efficiency**: As we have seen above example **readString() method** reads the entire file at once and we don’t need to take the overhead of repeatedly calling **readLine()** and concatenating strings.

**Error handling**: The **readString() method** provide support to handle errors more consistently than the BufferedReader approach.

**Let’s see how we wrote the file before java 11:**

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

public **class** FileWriterExample {

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

FileWriter writer = **null**;

**try** {

File file = **new** File("data.txt");

writer = **new** FileWriter(file);

String fileContent = "Hello, world!";

writer.write(fileContent);

} **catch** (IOException e) {

e.printStackTrace();

} finally {

**try** {

**if** (writer != **null**) {

writer.close();

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public **class** BufferedWriterExample {

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

BufferedWriter writer = **null**;

**try** {

writer = **new** BufferedWriter(**new** FileWriter("data.txt"));

String fileContent = "Hello, world!";

writer.write(fileContent);

} **catch** (IOException e) {

e.printStackTrace();

} finally {

**try** {

**if** (writer != **null**) {

writer.close();

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

**Java 11 Predicate Not Method Example**

Java 11 introduces a new method i.e. called the**“not() method”**which provides an easy way to negate a predicate. It presents in the [**Predicate interface**](https://javagoal.com/java-8-predicate/). A **Predicate interface** is a functional interface that takes an argument and returns a boolean value. Mostly it uses to filter [**collections**](https://javagoal.com/collection-in-java/)or [**streams**](https://javagoal.com/java-8-stream/) of data.

Let’s understand what it does Negate a predicate is required using the “negate()” method. It’s a **default method** and **static method** that returns a new predicate that is the negation of the original predicate. Let’s see the syntax of not() method:

static <T> Predicate<T> not(Predicate<? super T> target)

Let’s see how does Predicate not() method work. Suppose we want to check even or odd numbers by use of a predicate.

import java.util.function.Predicate;

public **class** Example {

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

Predicate<Integer> isEven = num -> num % 2 == 0;

Predicate<Integer> isOdd = isEven.not();

System.out.println(isOdd.test(5)); // Output: true

}

}

Here, we define a predicate called “isEven” that checks whether an integer is even. Now we want to create another predicate called “isOdd”, So we can use the “**not()**” method to create a new predicate called “isOdd” that checks whether an integer is odd.

We can use the “**not()**” method as in chained with other methods, like “and()” and “or()”, to create more complex predicates.

import java.util.function.Predicate;

public **class** Example {

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

Predicate<Integer> isEven = num -> num % 2 == 0;

Predicate<Integer> isDivisibleBy3 = num -> num % 3 == 0;

Predicate<Integer> isEvenAndDivisibleBy3 = isEven.not().and(isDivisibleBy3);

System.out.println(isEvenAndDivisibleBy3.test(6)); // Output: true

System.out.println(isEvenAndDivisibleBy3.test(9)); // Output: false

}

}

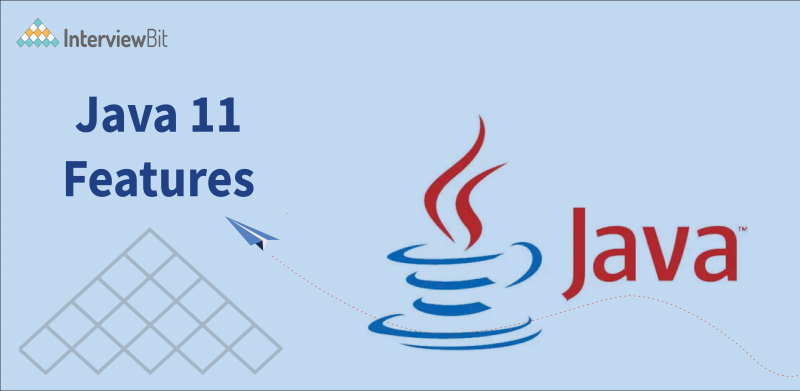
**Limitations**

1. It only works for predicates that return a boolean value.  
2. It’s not always easy to read and understand the code when multiple negations are involved.

**OR**

**Top Java 11 Features You Must Know [2023]**

March 28, 2023



**Table Of Contents**

show

* [Introduction](https://www.interviewbit.com/blog/java-11-features/#introduction)
* [Top Java 11 Features You Must Know](https://www.interviewbit.com/blog/java-11-features/#top-java-11-features-you-must-know)
  + [1. New HTTP (and WebSocket) Client](https://www.interviewbit.com/blog/java-11-features/#1-new-http-and-websocket-client)
  + [2. Nested classes attributes’ visibility management](https://www.interviewbit.com/blog/java-11-features/#2-nested-classes-attributes%E2%80%99-visibility-management)
  + [3. Epsilon: A No-Op Garbage Collector](https://www.interviewbit.com/blog/java-11-features/#3-epsilon-a-no-op-garbage-collector-)
  + [4. New Characters, Scripts, and Blocks](https://www.interviewbit.com/blog/java-11-features/#4-new-characters-scripts-and-blocks)
  + [5. Java Flight Recorder](https://www.interviewbit.com/blog/java-11-features/#5-java-flight-recorder)
  + [6. APis Improvements](https://www.interviewbit.com/blog/java-11-features/#6-apis-improvements)
  + [7. Deprecations and Deletions](https://www.interviewbit.com/blog/java-11-features/#7-deprecations-and-deletions)
  + [8. Other Enhancements](https://www.interviewbit.com/blog/java-11-features/#8-other-enhancements-)
  + [9. Long-Term Support (LTS)](https://www.interviewbit.com/blog/java-11-features/#9-long-term-support-lts)
* [Conclusions](https://www.interviewbit.com/blog/java-11-features/#conclusions)
* [FAQs](https://www.interviewbit.com/blog/java-11-features/#faqs)
  + [Q.1: What is special about Java 11?](https://www.interviewbit.com/blog/java-11-features/#q1-what-is-special-about-java-11)
  + [Q.2: What are the Advantages of Java 11?](https://www.interviewbit.com/blog/java-11-features/#q2-what-are-the-advantages-of-java-11)
  + [Q.3: What is the difference between Java 8 and Java 11?](https://www.interviewbit.com/blog/java-11-features/#q3-what-is-the-difference-between-java-8-and-java-11)
  + [Q.4: Does spring boot support Java 11?](https://www.interviewbit.com/blog/java-11-features/#q4-does-spring-boot-support-java-11)
* [Additional Resources](https://www.interviewbit.com/blog/java-11-features/#additional-resources)

**Introduction**

One of the world’s most commonly utilised and in-demand programming languages, Java, since its arrival in 1995, has kept evolving. In recent years, with its periodic release cycle, it needs a bit more effort to stay on par with the new releases. Java introduces a new version of the release, at an interval every six months.

Oracle made JDK (Java Development Kit) 11 on September 25, 2018, available. The latest update of the reference implementation of the Java SE platform, JDK 11, is the most efficient version in the Java version history. Java SE 11 is also the first Long Term Support (LTS) update in the new six-month release schedule of Java.



Many tech enthusiasts were waiting eagerly for the new release and working with the latest features of Java 11. After Java 8, Java 11 appears to be the second LTS release, speaking enough for its usefulness. With the advent of Java 11, Oracle JDK is not available anymore for commercial purposes with zero costs. Additionally, you should note that the new releases do not influence the effectiveness of a Java programming certification.

**Confused about your next job?**

In 4 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)

Now, let us look into the Java 11 features for a better understanding of what Java 11 brings to the table!

**Top Java 11 Features You Must Know**



**1. New HTTP (and WebSocket) Client**

Java 11 comprises a new HTTP client. (Since Java 9, this had been available as an ‘incubator’ module, but it has now graduated following the finalisation of its API). It contained the client in the java.net.http module/package and comprises the following important interfaces and classes:

* HttpClient is the entry point for utilising the API.
* HttpRequest is a request to be sent through the HttpClient.
* HttpResponse is the result of an HttpRequest call.
* WebSocket is the entry point for building up a WebSocket client.

The client is a considerable enhancement over Java’s continuing HTTP client support –  java.net.HttpURLConnection – which, owing to the various limitations, wasn’t a feasible choice for production usage. The enhancements include:

* **Easier API and Better Code Readability** – A fluent API ensures constructing requests and processing responses are easier to code, simpler to understand, and less verbose. Usage of the Builder pattern rather than setter techniques also enhances the client’s thread safety and its requests/responses.
* **Compatible with other contemporary Java APIs** – The client backs using contemporary Java (8+) language features and APIs, such as Lambda expressions, Optional, and the standard (java.time) Date and Time API, and so on.
* **Support for Non-blocking I/O** – The client supports executing requests without blocking the current thread through the usage of an async API/callbacks, which utilise java.util.concurrent.CompletableFuture
* **Protocol support** – Besides HTTP/1.1, the latest HTTP client also offers support for HTTP/2 (the default, with automatic negotiation and fallback to HTTP/1.1) along with WebSocket.

**2. Nested classes attributes’ visibility management**

Java supports several classes to be declared in a single source file, like nested classes. However, from the user’s perspective, they are regarded to belong to the “same class”. Thus, users assume them to have a shared access regime relative to their attributes. To sustain these assumptions, compilers should expand private attribute access to classes in the same package by incorporating access bridges. They assembled an invocation of a private member into a call of a compiler-generated method (getter) in the target class, which accesses the proposed private member.

For instance, with a NestedClass class, nested within a NestingClass class, which requires accessing one of the private attributes of the host class:

The compiler splits the two classes and sets up a public access method to nestingInt adopted by the NestedClass class:

These bridges undermine the encapsulation (private no longer has the same meaning) and can mislead users and tools. A formal idea of a group of nesting class files (or nest), where nest partners share a common access control technique, makes it plausible to get the anticipated outcome directly in a simpler, more safe, and more transparent way. To readily connect nested classes and hosts in JDK 11, it has incorporated two new attributes to the classes: NestMembers (nest members) and NestHost (new host)

Also, they have added 3 methods to java.lang.Class

* Class getNestHost ()
* Class [] getNestMembers ()
* boolean isNestmateof (Class)

**3. Epsilon: A No-Op Garbage Collector**

We got a new garbage collector, Epsilon GC with JDK 11. This GC manages memory allocation but doesn’t work on the mechanism of actual memory reclamation. Thus, when the overall java heap runs out, throwing an OutOfMemoryError, it will shut down the JVM.

The idea of building this GC is to offer the lowest possible latency overhead with a bounded memory allocation limit, but at the cost of memory throughout. Basically, it implies that learning about the heap will be sufficient for our application, so we don’t need the JVM to utilise resources to execute GC tasks(short pauses). It eliminates the impact GC has on execution and the code runs in absolute isolation.

* It can mainly utilise epsilon GC for testing the performance, like how garbage collection influences your application’s performance and what is the threshold of your application memory. If you assume that your application will require 4GB of memory then with “-Xmx4g” argument, we can execute it and if the JVM collapses, with “-XX:HeapDumpOnOutOfMemoryError” enabled, we can rerun it. Then we can observe the heap dump to learn where or which process has spent more memory.
* Also, Epsilon GC is beneficial for applications where we have to perform a last-drop latency enhancements, mostly for applications that are sensitive to ultra latency, where we are certain about memory allotments and know the application memory footprint and where receiving the GC cycle can be a concern. In those applications, long GC cycle can be an issue as that stretches the failure detection and conclusively recovery gets delayed. In such scenarios, allowing load balancers to figure out failure is at times a better recovery strategy than accepting a GC cycle.

**4. New Characters, Scripts, and Blocks**

The striking difference between various versions of Java and Java 11 is the addition of 16,018 new characters. Java 11 has incorporated 19 symbols, and 128 emoji characters, perfect for the new 4K TV standard, and Bitcoin signs. Additionally, you can come across 18 new blocks and 10 new scripts. The 10 latest scripts in Java 11 are Newa, Soyombo, Adlam, Osage, Marchen, Masaram Gondi, Tangut, Bhaiksuki, Zanabazar Square, and Nushu. Among the latest 18 blocks, ten are especially ideal for the ten new scripts. The rest of the eight new blocks are the right fit for existing scripts. The existing scripts are Tangut Components, Cyrillic Extended-C, Mongolian Supplement, Ideographic Symbols and Punctuation, Kana Extended-A, CJK Extension F, Syriac Supplement, and Glagolitic Supplement.

**5. Java Flight Recorder**

Many tools assist us in analysing and resolving errors during the process of development. However, specific issues only appear at the application’s runtime. Analysing them is often laborious or impossible, as we often cannot reproduce such mistakes. Java Flight Recorder (JFR) can help us by recording JVM data at runtime and ensuring it is available in a file for successive evaluation. For many years, Flight Recorder has already been there as a commercial feature in Oracle’s JDK. With JDK Enhancement Proposal 328, it becomes part of the OpenJDK and can be utilised freely.

**How to Initiate a Flight Recorder?**

You can initiate Flight Recorder in two manners. First, with the help of the following option on the java command line, you can activate Flight Recorder at the beginning of an application:

**-XX:StartFlightRecording=filename=<file name>**

Second, in a running Java application, you can utilise jcmd to activate Flight Recorder:

**jcmd JFR.start filename=<file name>**

You can mention many options; for instance, you can make use of “duration” to specify how long the recorder should operate.

**6. APis Improvements**

JDK 11 encloses several new classes and methods built into existing modules. The list mentioned below is a non-comprehensive overview, emphasising the enhancements that seem to appear to be the most important.

**java.lang.String:**

* boolean isBlank (): If the String is blank or made of just whitespaces, returns true, else false.
* stream lines (): Returns a stream of lines derived from the String.
* String strip (): returns a String clear of leading and trailing whitespaces. In simple terms, strip () is the “Unicode-aware” version of trim () whose definition of whitespace goes back from Java’s first versions.
* string repeat (int): returns a string that is the original String concatenated with itself n times.
* String stripLeading (): Returns a String free of its leading whitespaces.
* String stripTrailing (): returns a String devoid of trailing white spaces.
* Java.util.function.: It represents a function which takes in one argument and produces a result

**java.util.function.Predicate**

* Predicate not (Predicate). Returns a Predicate which is the repudiation of the Predicate passed as an argument. For instance, to filter a list:

**java.lang.CharSequence**

* int compare (CharSequence, CharSequence): juxtaposes two occurrences of CharSequence in lexicographic order. Returns a positive, null or negative value.

**java.lang.StrinBuilder / java.lang.StringBuffer**

* Now, both classes have access to a new compareTo () approach that receives a StringBuffer / StringBuilder argument and returns an int. The logic of comparison follows the exact lexicographic order as for the CharSequence class’s new method.

**7. Deprecations and Deletions**

This lists talks about all the “deprecated” features or features that are removed from the JDK.

**Eliminate the Java EE and CORBA Modules**

* JDK Enhancement Proposal 320 gets rid of the below mentioned modules from the JDK:
* java.xml.ws (JAX-WS)
* java.xml.bind (JAXB)
* java.activation (JAF)
* java.xml.ws.annotation (Common Annotations)
* java.corba (CORBA)
* java.transaction (JTA)
* java.se.ee (aggregator module for the 6 formerly discussed modules)
* jdk.xml.ws (tools for JAX-WS)
* jdk.xml.bind (tools for JAXB)

Initially, the listed technologies were designed for the Java EE platform and were incorporated into the standard edition “Java SE” when Java 6 was released. They were marked “deprecated” in Java 9 and finally got rid of with Java 11. After upgrading to Java 11, should you miss these libraries, you can bring them back to your project, example through Maven dependencies.

**Discontinue the Nashorn JavaScript Engine**

In JDK 8, the JavaScript engine, “Rhino” was introduced and with JEP 335 in Java 11, it was tagged as “deprecated for removal” and is to be eliminated altogether in one of the subsequent versions. The reason for this is the quick advancement of ECMAScript (the standard behind JavaScript) and the node.js engine, which has made further improvement of Rhino too pricey.

**Discontinue the Pack200 Tools and API**

In Java 5, it introduced a special compression method Pack200 that reaches greater compression rates than standard methods, specifically for .jar and .class files. In the early 2000s, Pack200 saved as much bandwidth as possible. Nonetheless, the algorithm is complicated, and the further enhancement costs are no longer in line with the usefulness in times of 100-Mbit Internet lines. Thus, the tool has been tagged as “deprecated” with JDK Enhancement Proposal 336 and should be erased in one of the following Java releases.

**JavaFX Goes Its Own Way**

With Java 11, JavaFX and the related javapackager tool are no longer shipped with the JDK. Rather, you can a separate SDK can be downloaded from the JavaFX homepage.

**8. Other Enhancements**

**Incorporated support for Unicode 11**

Java 11 has added Unicode 11 support to prevailing platform APIs. Mentioned below are the following Java classes that are mostly supported with Unicode 10:

* In the java.awt.font package: NumericShaper
* In the java.lang package: String and Character
* In the java.text package: Bidi, Normalizer and BreakIterator

This upgrade comprises Unicode 9 changes and incorporates ten new scripts and 16,018 characters.

**Enhanced KeyStore Mechanisms**

Java 11 can inspire the security precedents for Java 12 features. The enhanced KeyStore mechanisms in Java 11 can certainly give valid proof of that. In Java 11, you can look for a new security property with the name ‘jceks.key.serialFilter’. Users of JCEKS KeyStore can use this security filter during the time of deserialization of encrypted key object placed in a SecretKeyEntry. With no configuration, the result of the filter renders an UNDECIDED value and gets default configuration by ‘jdk.serialFilte

**Low-overhead Heap Profiling**

Java 11 offers the feature of a low-overhead method of sampling of Java heap allocations via the JVMTI. The idea of this method is perfect for focusing on specific goals. First, low-overhead is at a satisfactory level for default activation. The next usefulness is to get data about dead and live Java objects. The JVMTI is a well-defined and programmatic interface that offers easy access. You would also learn about low-overhead heap profiling as one of the probable enhancements to Java 12 features.

**Enhanced Aarch64 Intrinsics**

The compiler handles another function, an Intrinsic in some special way. They take benefit from the CPU architecture-specific assembly code to enhance the performance. Java 11 optimised and enhanced the prevailing string and array intrinsic on AArch64(or ARM64) processors. Also, Java 11 incorporated new Intrinsics for cos, log and sin methods of the java.lang.

**Launching Single-File Java Program**

Java 11 backs the execution of a single file Java program with a single command.

**Details**

Facilitates high speed prototyping and beginner friendly Java development.  
It ought to be a single-file program only, having one or more classes.  
We can’t utilise external dependencies./

**How to use?**

Let’s assume the file name is MyProgram.java, to RUN it, we only require to utilise java MyProgram.java.

**9. Long-Term Support (LTS)**

You can utilise Java 11 in various phases of development. Nonetheless, commercial usage of the latest Java 11 features would call for a license. After the release date of Java 11, many essential details came to the forefront. For example, Oracle would not lend support to Java 8 post January 2019. You can keep using Java 8 by paying for support, though you would not get any patches or security updates. Since you cannot utilise Oracle JDK any longer with no costs, you can download Open JDK builds. Many providers offer flexible options, and you can utilise Open JDK builds from these providers and also upgrade them when needed. With Java 11, on September 25, 2018, the first LTS release of the JDK since the shift to the six-month release cycle was published. “Long-Term Support (LTS)” implies that Oracle will give this version with security patches for many years. The last LTS release was Java 8, Java 9 and 10 were not LTS releases, which implies that support for these versions was discontinued with every successive release.

In this article, we have sorted the changes in Java 11 as per the relevance for daily programmer work. First comes the modifications to the language itself, accompanied by improvements to the JDK class library, tools, and experimental features. And lastly, deletions, deprecations, and other slight adjustments. It is also necessary to learn that from version 11 onwards, programmers can only utilise freely the Oracle JDK. Businesses call for a paid support contract with Oracle. OpenJDK 11 is for everyone and free to use

**Conclusions**

A study of the different new Java 11 features reveals the path for more enhancements in future Java releases. Also, Java 11 incorporated different recent developments in a place, like other new releases.

The most striking feature of Java 11 is that it is a steady Long-Term Support (LTS) release of Java. Post Java 8, this is the 2nd LTS release of Java. You may not recognise any developments regarding the casting in Java. This means that Java 12 is another Long-Term Support release. As we looked into the new features in Java 11, we discovered that it misses many old features. For instance, Java Deployment Technologies and SNMP Agent are not available in Java 11. Java 11 has surely brought some radical modifications to the Java landscape. Thus, if you wish to try out the latest features in Java 11, then buy your license and get started!

**FAQs**

**Q.1: What is special about Java 11?**

**Answer:** Java 11 incorporated a few more modifications that are necessary to highlight like New ChaCha20 and ChaCha20-Poly1305 cipher implementations change the insecure RC4 stream cipher. Support for cryptographic key agreement with Curve25519 and Curve448 replaces the prevailing ECDH scheme.

It is plausible in Java 11 to develop a jar file that comprises several Java-release-specific versions of class files. Multi-release jar files make it feasible for library programmers to back different versions of Java without having to ship different versions of jar files.

**Q.2: What are the Advantages of Java 11?**

**Answer:**Java 11 incorporates new Java Enhancement Proposals. The advantages are as follows:

* Nest-based access control
* Dynamic class-file constraints
* Enhance Aarch64 Intrinsics
* Epsilon: A No-Op Garbage Collector
* Eliminate the Java EE and CORBA Modules
* HTTP Client (Standard)

**Q.3: What is the difference between Java 8 and Java 11?**

**Answer:** In Java 8, the appletviewer tool is available whereas, in Java 11, the tool is not available. Java 8 has fewer string methods whereas in Java 11, different new methods of String like repeat(n), lines(), stripLeading(), stripTrailing(), and strip() are introduced.

**Q.4: Does spring boot support Java 11?**

**Answer**: Spring Boot 2.4 backs Java 15 while also remaining compatible with Java 11 and 8.

**OR**

# Java 11 Features: A Brief Overview

## Introduction

Java, one of the world’s most widely used and in-demand programming languages, has continued to develop since its introduction in 1995. Because of the periodic release cycle, it takes a little more work these days to keep up with the latest releases of Java.

Every six months, Java releases a new version of its software. **Java JDK 11**became accessible on September 25, 2018, thanks to Oracle. The first Long Term Support (LTS) upgrade in Java’s new six-month release cycle is Java SE 11. Many computer specialists eagerly awaited the new release and worked with Java 11’s most recent features. Java 11 is the second LTS release after Java 8, which speaks volumes about its utility.

Oracle JDK is no longer free to use for commercial purposes with the release of **Java 11**. You should also be aware that the effectiveness of a Java programming certification is unaffected by the latest releases.

## Why Is Java 11 Important?

Let’s look at the importance of**Java 11** before we go into the new features.

* Running applets and web apps requires a deployment stack.
* An entire category of supported browsers is dropped from the list of permitted configurations since the deployment stack is unavailable.
* Java 11 offers separate downloads for Java Mission Control and JavaFX.
* For JRE installs on macOS and Windows, there is no auto-update feature.
* JRE or Server JRE is not included in Java 11, and users only receive JDK.
* Some languages’ Java language translations are not accessible in Java 11. French, Italian, German, Swedish, Spanish, Brazilian, Portuguese, and Korean are the languages not supported.
* Java 11 features also include updates to the Windows and macOS packaging formats.
* Instead of tar.gz, Java 11’s revised package format for Windows is.zip. Instead of .app, the modified packaging type for macOS is.dmg.

Now let’s take a closer look at the **new Java features**.

## What Are the Features of Java 11?

Here is a list of **Java 11 features with examples**:

### New String Methods

Several useful methods have been added to the String class in Java 11:

**StripLeading(), StripTailing(), and String.strip ():**

A String’s leading and trailing whitespace is eliminated via the String.strip() function. The trim() function eliminates any characters with a code point of U 0020 or less. For instance, “space,” “tab,” “newline,” and “carriage return” are included in this.

The strip() removes characters that are classified as whitespaces by Character.isWhitespace(). On the one hand, some (but not all) of these characters have a code point of U 0020 or less.

The letters designated in the Unicode Standard as spaces, line breaks, and paragraph separators, such as U 2002, a space as wide as the letter “n,” are the opposite.The technique has two variations: stripLeading() only removes leading whitespaces, and stripTailing() only removes trailing ones.

**String.isBlank():**

Only characters that the Character.isWhitespace() method indicated in the preceding sentence defines as whitespaces are allowed in a String for the new String.isBlank() function to return true.

**String.repeat():**

String.repeat() can be used to concatenate a String repeatedly:

           System.out.println(“:-) “.repeated(20));

### New File Methods:

One of the new Java release’s main attractions is adding new file methods to the list of Java 11 capabilities. There are three new file methods: “writeString(),” “readString,” and “isSameFile.”

While “readString()” is best for reading file contents, “writeString()” is best for writing content to files. When determining whether two paths refer to the same file, the ‘isSameFile()’ method is useful.

### Collection to an Array:

           The new default toArray() method makes it simple to change a collection into the right kind of array.

public class JavaDemo

{

    public static void main(Strings[] args) {

        ArrayList<Integers> list = new ArrayList<>();

        list.add(8);

        list.add(16);

        list.add(24);

        Integer[] intArr = list.toArray(Integers[]::Allnew);

        System.out.print(Arrays.toString(intArr));

    }

}

### The Not Predicate Method:

The Predicate.not() static method introduced in Java 11 is used to reject an existing predicate. The java.util.function package contains the Predicate class.

negate = Predicate.not( positivesPredicate );

### Local-Variable Syntax for Lambda:

Var can be used in lambda expressions in JDK 11. This was added to be compatible with Java 10’s local “var” syntax.

To avoid combining the type name with the variable name in Java 11, we can use the var keyword with lambda expression parameters. The var keyword was previously forbidden when working with a lambda expression in Java, but it is now permitted.

interface Calc{

    int sum(int x, int y);

}

public class Main {

    public static void main(String[] args){

         Calc calc = (x, y)-> x y;

         int result = calc.sum(20, 40);

         System.out.println(result);

    }

}

### HTTP Client:

The new HTTP Client API, part of the java.net.http package, was first made available in Java 9 and later enhanced in Java 10 before becoming a standard feature in Java 11. A simple GET request can be sent using a Java 11 HttpClient.

HttpsClient httpsClient = HttpsClient.newBuilder()       .versionNumber(HttpsClient.Version.HTTP\_2)        .followRedirect(HttpsClient.Redirect.NORMAL)  
.connectionTimeout(Duration.ofSecond(40))  
.proxy(ProxySelectors.of(new InetSocketAddress(“proxy.yourcompanyid.com”, 90)))  
.authenticators(Authenticators.getDefault())  
.build();

### Nest-Based Access Control:

Nest is an access-control framework that complies with the Java programming language’s current nested type concept. The use of nests allows classes compiled into distinct class files but components of the same code structure to communicate with each other’s private members without the need for compilers to provide accessibility-enlarging bridge methods.

public class JavaDemo {

    private void privateMethod() {

        System.out.print(“Private Methods”);

    }

    class NestedClass {

        public void callPrivateMethods() {

            privateMethod();

        }

    }

### Running Java Files:

We must first compile a Java file using the javac command before we can run it. But with Java 11, we may use a single java command to start a Java file instantly.

java MyFirstJavaBasedProgram.java

## JAVA 11 features for Performance Enhancement

 The following are the Java 11 features for performance enhancement:

### Dynamic Class-File Constants:

With Java 11, Class-File format supports a brand-new constant pool form known as CONSTANT Dynamic. In this situation, a bootstrap method will be created. The expense of creating new forms of materializable class-file constants was reduced by establishing a single new constant-pool form that would be parameterized with the appropriate user-defined behavior. With this feature, the performance has significantly improved.

### Improved Aarch64 Intrinsics:

A function that is treated uniquely by the compiler is known as an intrinsic. Performance is increased by utilizing assembly-level language code tailored to the CPU architecture.The string and array intrinsics on AArch64 (or ARM64) processors were enhanced and optimized in Java 11. For the java.lang.Math sin, cos, and log methods, Java 11 also included new intrinsics.

### A No-Op Garbage Collector:

With JDK 11, we received a new garbage collector called Epsilon GC. Although it doesn’t work on the process of actual memory reclamation, this GC manages memory allocation. As a result, the JVM will shut down when the entire Java heap runs out of memory and throws an OutOfMemoryError.

The goal of developing this GC is to provide limited memory allocation limits with the least amount of delay overhead possible. In essence, it says that learning about the heap will be enough for our program. Therefore we don’t need to borrow resources from the JVM to carry out GC chores (short pauses). The GC’s influence on execution is eliminated, and the code executes in complete isolation.

Epsilon GC can be used primarily for performance testing, including determining how garbage collection affects your application’s speed and memory threshold for your application. If you anticipate that your program will need 4GB of memory, you can run it with the “-Xmx4g” parameter, and if the JVM crashes, you can rerun it with the “-XX:HeapDumpOnOutOfMemoryError” option enabled. The heap dump is then examined to see which process or where extra memory has been used.

          -XX: UnlockExperimentalsVMOption -XX: UseEpsilonGC

### Flight Recorder:

The Java Flight Recorder, sometimes called JFR, collects applications’ profiling information. It was formerly exclusively accessible for commercial purposes, but OpenJDK 11 has made it open-source. Due to its low overhead (around 1%), we can use it for production applications. The information is stored in a JFR file, and we can use the JDK Launch Control tool to examine it. Start a 180-second JFR recording with the following command, and the data will be saved in the Javademo.jfr file.

 -XX:StartFlightsRecording=duration=180s,setting=profiles,filename=Javademo.jfr

**Other Changes:**

The following are some of the other changes in Java 11:

* **Included Assistance for Unicode 11:**

Java 11 has added Unicode 11 compatibility to existing platform APIs. The following Java classes are listed below, and most of them are compatible with Unicode 10:

* NumericShaper is a typeface in the java.awt.font package.
* String and Character are included in java.lang package.
* Bidi, Normalizer, and BreakIterator are included in java.text package.

This update adds 16,018 characters and ten new scripts to the Unicode 9 standard.

* **Start Single-File Source-Code Applications:**

JDK Enhancement Proposal 330 becomes interesting for small Java programs with just one class.

This allows you to use the java command to compile and run a .java file. Furthermore, a so-called “shebang” can make a .java file executable.

#### Conclusion

Examining the various new Java 11 features highlights the direction for additional improvements in the next Java editions. Additionally, Java 11 included other recent advancements in a location, similar to other new releases. The fact that Java 11 is a consistent Long-Term Support (LTS) release of Java is its most notable characteristic. The second Java Long Term Support release follows Java 8. You might not notice any changes in Java’s casting system. This indicates that Java 12 will also be covered by long-term support.

We found that Java 11 lacked a lot of the older functionality when we investigated its new features. For instance, Java 11 does not support SNMP Agent or Java Deployment Technologies. The Java ecosystem has undoubtedly seen some significant changes thanks to Java 11. You can learn more about this language via **Java 11 tutorials**available for free over the web. We recommend you check out UNext to acquire cutting-edge skills to land a job.