**- Java 8 -**

Java 8 was a massive release and you can find a list of all features at [the Oracle website](https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html). There’s two main feature sets I’d like to mention here, though:

**Language Features: Lambdas etc.**

Before Java 8, whenever you wanted to instantiate, for example, a new Runnable, you had to write an anonymous inner class like so:

|  |
| --- |
| Runnable runnable = new Runnable(){  @Override  public void run(){  System.out.println(*"Hello world !"*);  }  }; |

With lambdas, the same code looks like this:

|  |
| --- |
| Runnable runnable = () -> System.out.println(*"Hello world two!"*); |

You also got method references, repeating annotations, default methods for interfaces and a few other language features.

**Collections & Streams**

In Java 8 you also got functional-style operations for collections, also known as the Stream API. A quick example:

|  |
| --- |
| List<String> list = Arrays.asList(*"franz"*, *"ferdinand"*, *"fiel"*, *"vom"*, *"pferd"*); |

Now pre-Java 8 you basically had to write for-loops to do something with that list.

With the Streams API, you can do the following:

|  |
| --- |
| list.stream()  .filter(name -> name.startsWith(*"f"*))  .map(String::toUpperCase)  .sorted()  .forEach(System.out::println); |

**If you want more Java 8 practice**

Obviously, I can only give a quick overview of each newly added Stream, Lambda or Optional method in Java 8 in the scope of this guide.

If you want a more detailed, thorough overview - including exercises - you can have a look at my [Java 8 core features](https://www.marcobehler.com/courses/32-core-java-features-version-8-12?utm_campaign=java_features_guide&utm_medium=java_features_guide&utm_source=java_features_guide) course.

**- Java 9 -**

Java 9 also was a fairly big release, with a couple of additions:

**Collections**

Collections got a couple of new helper methods, to easily construct Lists, Sets and Maps.

|  |
| --- |
| List<String> list = List.of(*"one"*, *"two"*, *"three"*);  Set<String> set = Set.of(*"one"*, *"two"*, *"three"*);  Map<String, String> map = Map.of(*"foo"*, *"one"*, *"bar"*, *"two"*); |

**Streams**

Streams got a couple of additions, in the form of takeWhile,dropWhile,iterate methods.

|  |
| --- |
| Stream<String> stream = Stream.iterate(*""*, s -> s + *"s"*)  .takeWhile(s -> s.length() < 10); |

**Optionals**

Optionals got the sorely missed ifPresentOrElse method.

|  |
| --- |
| user.ifPresentOrElse(this::displayAccount, this::displayLogin); |

**Interfaces**

Interfaces got private methods:

|  |
| --- |
| public interface MyInterface {  private static void myPrivateMethod(){  System.out.println(*"Yay, I am private!"*);  }  } |

**Other Language Features**

And a couple of other improvements, like an improved try-with-resources statement or diamond operator extensions.

**JShell**

Finally, Java got a shell where you can try out simple commands and get immediate results.

|  |
| --- |
| % jshell  | Welcome to JShell -- Version 9  | For an introduction type: /help intro  jshell> int x = 10  x ==> 10 |

**HTTPClient**

Java 9 brought the initial preview version of a new HttpClient. Up until then, Java’s built-in Http support was rather low-level, and you had to fall back on using third-party libraries like Apache HttpClient or OkHttp (which are great libraries, btw!).

With Java 9, Java got its own, modern client - although in preview mode, which means subject to change in later Java versions.

**Project Jigsaw: Java Modules and Multi-Release Jar Files**

Java 9 got the [Jigsaw Module System](https://www.oracle.com/corporate/features/understanding-java-9-modules.html), which somewhat resembles the good old [OSGI specification](https://en.wikipedia.org/wiki/OSGi). It is not in the scope of this guide to go into full detail on Jigsaw, but have a look at the previous links to learn more.

Multi-Release .jar files made it possible to have one .jar file which contains different classes for different JVM versions. So your program can behave differently/have different classes used when run on Java 8 vs. Java 10, for example.

**If you want more Java 9 practice**

Again, this is just a quick overview of Java 9 features and if you want more thorough explanations and exercises, have a look at the [Java 9 core features](https://www.marcobehler.com/courses/32-core-java-features-version-8-12?utm_campaign=java_features_guide&utm_medium=java_features_guide&utm_source=java_features_guide) course.

**- Java 10 -**

There have been a few changes to Java 10, like Garbage Collection etc. But the only real change you as a developer will likely see is the introduction of the "var"-keyword, also called local-variable type inference.

**Local-Variable Type Inference: var-keyword**

|  |
| --- |
| *// Pre-Java 10*  String myName = *"Marco"*;  *// With Java 10*  var myName = *"Marco"* |

Feels Javascript-y, doesn’t it? It is still strongly typed, though, and only applies to variables *inside methods* (thanks, [dpash](https://www.reddit.com/user/dpash), for pointing that out again).

**- Java 11 -**

Java 11 was also a somewhat smaller release, from a developer perspective.

**Strings & Files**

Strings and Files got a couple new methods (not all listed here):

|  |
| --- |
| *"Marco"*.isBlank();  *"Mar\nco"*.lines();  *"Marco "*.strip();  Path path = Files.writeString(Files.createTempFile(*"helloworld"*, *".txt"*), *"Hi, my name is!"*);  String s = Files.readString(path); |

**Run Source Files**

Starting with Java 10, you can run Java source files *without* having to compile them first. A step towards scripting.

|  |
| --- |
| ubuntu@DESKTOP-168M0IF:~$ java MyScript.java |

**Local-Variable Type Inference (var) for lambda parameters**

The header says it all:

|  |
| --- |
| (var firstName, var lastName) -> firstName + lastName |

**HttpClient**

The HttpClient from Java 9 in its final, non-preview version.

**Other stuff**

Flight Recorder, No-Op Garbage Collector, Nashorn-Javascript-Engine deprecated etc.

**- Java 12 -**

Java 12 got a couple [new features and clean-ups](https://www.oracle.com/technetwork/java/javase/12-relnote-issues-5211422.html), but the only ones worth mentioning here are Unicode 11 support and a preview of the new switch expression, which you will see covered in the next section.

**- Java 13 -**

You can find a complete feature list [here](https://www.oracle.com/technetwork/java/13-relnote-issues-5460548.html), but essentially you are getting Unicode 12.1 support, as well as two new or improved preview features (subject to change in the future):

**Switch Expression (Preview)**

Switch expressions can now return a value. And you can use a lambda-style syntax for your expressions, without the fall-through/break issues:

Old switch statements looked like this:

|  |
| --- |
| switch(status) {  case SUBSCRIBER:  *// code block*  break;  case FREE\_TRIAL:  *// code block*  break;  default:  *// code block*  } |

Whereas with Java 13, switch statements can look like this:

|  |
| --- |
| boolean result = switch (status) {  case SUBSCRIBER -> true;  case FREE\_TRIAL -> false;  default -> throw new IllegalArgumentException(*"something is murky!"*);  }; |

**Multiline Strings (Preview)**

You can *finally* do this in Java:

|  |
| --- |
| String htmlBeforeJava13 = *"<html>\n"* +  *" <body>\n"* +  *" <p>Hello, world</p>\n"* +  *" </body>\n"* +  *"</html>\n"*;  String htmlWithJava13 = *"""*  *<html>*  *<body>*  *<p>Hello, world</p>*  *</body>*  *</html>*  *"""*; |

**- Java 14 -**

**Switch Expression (Standard)**

The switch expressions that were *preview* in versions 12 and 13, are now standardized.

|  |
| --- |
| int numLetters = switch (day) {  case MONDAY, FRIDAY, SUNDAY -> 6;  case TUESDAY -> 7;  default -> {  String s = day.toString();  int result = s.length();  yield result;  }  }; |

**Records (Preview)**

There are now record classes, which help alleviate the pain of writing a lot of boilerplate with Java.

Have a look at this pre Java 14 class, which only contains data, (potentially) getters/setters, equals/hashcode, toString.

|  |
| --- |
| final class Point {  public final int x;  public final int y;  public Point(int x, int y) {  this.x = x;  this.y = y;  }  }  *// state-based implementations of equals, hashCode, toString*  *// nothing else* |

With records, it can now be written like this:

|  |
| --- |
| record Point(int x, int y) { } |

Again, this is a preview feature and subject to change in future releases.

**Helpful NullPointerExceptions**

Finally NullPointerExceptions describe *exactly* which variable was null.

|  |
| --- |
| author.age = 35;  ---  Exception in thread *"main"* java.lang.NullPointerException:  Cannot assign field *"age"* because *"author"* is null |

**Pattern Matching For InstanceOf (Preview)**

Whereas previously you had to (cast) your objects inside an instanceof like this:

|  |
| --- |
| if (obj instanceof String) {  String s = (String) obj;  *// use s*  } |

You can now do this, effectively dropping the cast.

|  |
| --- |
| if (obj instanceof String s) {  System.out.println(s.contains(*"hello"*));  } |

**Packaging Tool (Incubator)**

There’s an incubating *jpackage* tool, which allows to package your Java application into platform-specific packages, including all necessary dependencies.

* Linux: deb and rpm
* macOS: pkg and dmg
* Windows: msi and exe

**Garbage Collectors**

The Concurrent Mark Sweep (CMS) Garbage Collector has been removed, and the experimental Z Garbage Collector has been added.

**- Java 15 -**

**Text-Blocks / Multiline Strings**

Introduced as an experimental feature in Java 13 (see above), multiline strings are now production-ready.

|  |
| --- |
| String text = *"""*  *Lorem ipsum dolor sit amet, consectetur adipiscing \*  *elit, sed do eiusmod tempor incididunt ut labore \*  *et do lore magna aliqua.\*  *"""*; |

**Sealed Classes - Preview**

If you ever wanted to have an even closer grip on who is allowed to subclass your classes, there’s now the *sealed* feature.

|  |
| --- |
| public abstract sealed class Shape  permits Circle, Rectangle, Square {...} |

This means that while the class is *public*, the only classes allowed to subclass *Shape* are *Circle*, *Rectangle* and *Square*.

**Records & Pattern Matching**

The *Records* and *Pattern Matching* features from Java 14 (see above), are still in preview and not yet finalized.

**Nashorn JavaScript Engine**

After having been deprecated in Java 11, the Nashorn Javascript Engine was now finally removed in JDK 15.

**ZGC: Production Ready**

The [Z Garbage Collector](https://wiki.openjdk.java.net/display/zgc/Main) is not marked experimental anymore. It’s now production-ready.

**- Java 16 -**

**Pattern Matching for instanceof**

Instead of:

|  |
| --- |
| if (obj instanceof String) {  String s = (String) obj;  *// e.g. s.substring(1)*  } |

You can now do this:

|  |
| --- |
| if (obj instanceof String s) {  *// Let pattern matching do the work!*  *// ... s.substring(1)*  } |

**Unix-Domain Socket Channels**

You can now connect to Unix domain sockets (also supported by macOS and Windows (10+).

|  |
| --- |
| socket.connect(UnixDomainSocketAddress.of(  *"/var/run/postgresql/.s.PGSQL.5432"*)); |

**Foreign Linker API - Preview**

A planned replacement for JNI (Java Native Interface), allowing you to bind to native libraries (think C).

**Records & Pattern Matching**

Both features are now production-ready, and not marked *in preview* anymore.

**Sealed Classes**

Sealed Classes (from Java 15, see above) are still in preview.

**- Java 17 -**

Java 17 is the new long-term support (LTS) release of Java, after Java 11.

**Pattern Matching for switch (Preview)**

Already available in many other languages:

|  |
| --- |
| public String test(Object obj) {  return switch(obj) {  case Integer i -> *"An integer"*;  case String s -> *"A string"*;  case Cat c -> *"A Cat"*;  default -> *"I don't know what it is"*;  };  } |

Now you can pass *Objects* to switch functions and check for a particular type.

**Sealed Classes (Finalized)**

A feature that was delivered in Java 15 as a preview is now finalized.

Recap: If you ever wanted to have an even closer grip on who is allowed to subclass your classes, there’s now the *sealed* feature.

|  |
| --- |
| public abstract sealed class Shape  permits Circle, Rectangle, Square {...} |

This means that while the class is *public*, the only classes allowed to subclass *Shape* are *Circle*, *Rectangle* and *Square*.

**Foreign Function & Memory API (Incubator)**

A replacement for the Java Native Interface (JNI). Allows you to call native functions and access memory *outside* the JVM. Think C calls for now, but with plans for supporting additional languages (like C++, Fortran) over time.

**Deprecating the Security Manager**

Since Java 1.0, there had been a Security Manager. It’s now deprecated and will be removed in a future version.

**- Java 18 -**

**UTF-8 by Default**

If you tried, e.g. reading in files without specifying an explicit character ending, the operating system encoding was used in previous Java versions (e.g. UTF-8 on Linux and macOS, and Windows-1252 on Windows). With Java 18 this changed to UTF-8 by default.

**Simple Web Server**

Java 18 now comes with a rudimentary HTTP server, that you can start with:

|  |
| --- |
| jwebserver |

Learn more about its features [here](https://openjdk.org/jeps/408).

**Other Not-So-Exciting-Stuff / Incubating Features**

For a full list and overview, check out [this article](https://www.happycoders.eu/java/java-18-features/).

**- Java 19 -**

Java 19 added a couple of exciting features, like *Virtual Threads* and the new *Foreign Function & Memory API*, as well as *Structured Concurrency* and the *Vector API* - but they are all in preview mode, thus subject to change across the next releases.

If you want to read up on these features and what’s to come, check out [this article](https://www.happycoders.eu/java/java-19-features/).

**- Java 20 -**

Apart from some smaller features, Java 20 mainly iterated on previously introduced preview features: Scope Values, Record Patterns, Pattern Matching for switch, Foreign Function & Memory API, Virtual Threads & Structured Concurrency. All of them are work-in-progress, i.e. haven’t been finalized yet.

If you want to read up on these features and what’s to come, check out [this article](https://www.happycoders.eu/java/java-20-features/).

public class MethodReference {

List<String> withoutMethodReference =

cars.stream().map(car -> car.toString())

.collect(Collectors.toList());

}

public class MethodReference {

List<String> methodReference = cars.stream().map(Car::toString)

.collect(Collectors.toList());

}

*EXAMPLE FOR DEFAULT METHOD  
  
interface Vehicle* {  
 *// Abstract method  
 void* move();  
  
 *// Default method  
 default void* stop() {  
 System.*out*.println("Vehicle is stopping...");  
 }  
}  
  
*// Implement the interface  
class* Car *implements Vehicle* {  
 *// Implement the abstract method* @Override  
 *public void* move() {  
 System.*out*.println("Car is moving...");  
 }  
  
 *// No need to implement the default method if not required*}  
  
*// Implement another class with different behavior for the default method  
class* Bicycle *implements Vehicle* {  
 *// Implement the abstract method* @Override  
 *public void* move() {  
 System.*out*.println("Bicycle is moving...");  
 }  
  
 *// Override the default method* @Override  
 *public void* stop() {  
 System.*out*.println("Bicycle is stopping...");  
 }  
}  
  
*public class* DefaultMethod {  
 *public static void* main(String[] args) {  
 *// Create instances of Car and Bicycle* Car car = *new* Car();  
 Bicycle bicycle = *new* Bicycle();  
  
 *// Call methods* car.move(); *// Output: Car is moving...* car.stop(); *// Output: Vehicle is stopping...* bicycle.move(); *// Output: Bicycle is moving...* bicycle.stop(); *// Output: Bicycle is stopping...* }  
}