







## **Java Features from Java 8 to Java 17**

Trainers, A lot has changed in Java from 1995 until today. Java 8 was a revolutionary release that put Java back on the pedestal of the best programming languages.

Let's go through most of the changes in the Java language that happened from Java 8 in 2014 until Java 17. I will share the main features between Java 8 and Java 17. The intention is to have a reference for all features between Java 8 and Java 17 inclusively. I am also sharing examples of each new feature in Java 8 for reference.





### **Java 8**

The main changes of the Java 8 release were these:

-  Lambda Expression and Stream API
-  Method Reference
-  Default Methods
-  Type Annotations
-  Repeating Annotations
-  Method Parameter Reflection

### **Java 9**


Java 9 introduced these main features:

-  Java Module System
-  Try-with-resources
-  Diamond Syntax with Inner Anonymous Classes
-  Private Interface Methods



### **Java 10**

-  Local Variable Type Inference

### **Java 11**

-  Local Variable Type in Lambda Expressions



### **Java 14**

-  Switch Expressions
-  The yield Keyword

### **Java 15**

-  Text Blocks

### **Java 16**

-  Pattern Matching of instanceof
-  Records

### **Java 17**

-  Sealed Classes

## Lambda Expressions and Stream API

Java was always known for having a lot of **boilerplate code**. With the release of Java 8, this statement became less valid. The stream API and lambda expressions are the new features that move us closer to functional programming.

The below examples will show how we use lambdas and streams in different scenarios.

### Scenario:

We own a car dealership business. To discard all the paperwork, we want to create a piece of software that finds all currently available cars that have run less than 50,000 km.

Let us take a look at how we would implement a function for something like this in a naive way:

**To implement the above scenario, we will create a static function that accepts a List of cars. It should return a filtered list according to a specified condition.**

### Example: [Before Java 8]

```
public class LambdaExpressions {  
    public static List<Car> findCarsOldWay(List<Car> cars) {  
        List<Car> selectedCars = new ArrayList<>();  
        for (Car car : cars) {  
            if (car.kilometers < 50000) {  
                selectedCars.add(car);  
            }  
        }  
        return selectedCars;  
    }  
}
```

**Let us look at how we would implement the above scenario using a Stream and a Lambda Expression.**

### Example:

```
public class LambdaExpressions {  
    public static List<Car> findCarsUsingLambda(List<Car> cars) {  
        return cars.stream().filter(car -> car.kilometers < 50000)  
            .collect(Collectors.toList());  
    }  
}
```





```
}  
}
```

### **Explanation:**

We need to transfer the list of cars into a stream by calling the `stream()` method. Inside the `filter()` method, we are setting our condition. We are evaluating every entry against the desired condition. We are keeping only those entries that have less than 50,000 kilometers. The last thing we must do is wrap it up into a list.

### **Method Reference**

A method reference allows us to call functions in classes using a special syntax `::`. There are four kinds of method references:

-  Reference to a static method
-  Reference to an instance method on an object
-  Reference to an instance method on a type
-  Reference to a constructor

**We will use the same scenario of owning a car dealership shop and want to print out all the cars in the shop. For that, we will use a method reference.**

### **Example using the standard method call:**

```
public class MethodReference {  
    List<String> withoutMethodReference =  
        cars.stream().map(car -> car.toString())  
            .collect(Collectors.toList());  
}
```

**We are using a lambda expression to call the `toString()` method on each car.**

### **Example Using a Method Reference for the above scenario**

```
public class MethodReference {  
    List<String> methodReference = cars.stream().map(Car::toString)  
        .collect(Collectors.toList());  
}
```

We are, again, using a lambda expression, but now we call the `toString()` method by method reference. We can see how it is more concise and easier to read.

## **Default Methods**

Suppose we have a simple method `log(String message)` that prints log messages on invocation. We wanted to provide message timestamps so that logs are easily searchable. We don't want our clients to break after we introduce this change. We will do this using a default method implementation on an interface.

Default method implementation is the feature that allows us to create a fallback implementation of an interface method.

In the below example, we have created a simple interface with just one method and implemented it in `LoggingImplementation` class.

### **Example:**

```
public class DefaultMethods {

    public interface Logging {

        void log(String message);

    }

    public class LoggingImplementation implements Logging {

        @Override

        public void log(String message) {

            System.out.println(message);

        }

    }

}
```

Next, we will add a new method inside the interface. The method accepts the second argument, called `date`, which represents the timestamp.

```
public class DefaultMethods {

    public interface Logging {

        void log(String message);

        void log(String message, Date date);

    }

}
```

```
}  
}
```

We have added a new method but have yet to implement it inside all client classes. The compiler will fail with the exception:

```
Class 'LoggingImplementation' must either be declared abstract  
or implement abstract method 'log(String, Date)' in 'Logging'.
```

After adding a new method inside the interface, our compiler threw exceptions. We will solve this using the default method implementation for the new method.






Let us look at how to create a default method implementation:

```
public class DefaultMethods {  
  
    public interface Logging {  
        void log(String message);  
  
        default void log(String message, Date date) {  
            System.out.println(date.toString() + ": " + message);  
        }  
    }  
}
```

Putting the default keyword allows us to add the implementation of the method inside the interface. Now, our LoggingImplementation class does not fail with a compiler error even though we didn't implement this new method inside of it.

### **Type Annotations**

Type annotations are one more feature introduced in Java 8. Even though we had annotations available before, now we can use them wherever we use a type. This means that we can use them on the following:

-  a local variable definition
-  constructor calls
-  type casting
-  generics
-  throw clauses

Tools like IDEs can then read these annotations and show warnings or errors based on the annotations.

### **Local Variable Definition**

Example to ensure that our local variable doesn't end up as a null value:

```
public class TypeAnnotations {  
  
    public static void main(String[] args) {  
        @NotNull String userName = args[0];  
    }  
}
```

We are using annotation on the local variable definition here. A compile-time annotation processor could now read the `@NotNull` annotation and throw an error when the string is null.

### **Constructor Call**

Example to make sure that we cannot create an empty ArrayList:

```
public class TypeAnnotations {  
  
    public static void main(String[] args) {  
        List<String> request =  
            new @NotEmpty ArrayList<>(Arrays.stream(args).collect(  
                Collectors.toList()));  
    }  
}
```

This is the perfect example of how to use type annotations on a constructor. Again, an annotation processor can evaluate the annotation and check if the array list is not empty.

### **Generic Type**

One of our requirements is that each email has to be in the format `<name>@<company>.com`. If we use type annotations, we can do it easily:

Example:

```
public class TypeAnnotations {  
  
    public static void main(String[] args) {  
        List<@Email String> emails;  
    }  
}
```

}

This is a definition of a list of email addresses. We use @Email annotation that ensures that every record inside this list is in the desired format.

A tool could use reflection to evaluate the annotation and check that each element in the list is a valid email address.