**Java 9 Programming Language Enhancements**

Oracle has released Java 9 with rich set of new features. It includes various upgrades to the Java programming, JVM, Tools and libraries. In this tutorial, we will discuss all the main features that are given below.

* Platform Module System (Project Jigsaw)
* Interface Private Methods
* Try-With Resources
* Anonymous Classes
* @SafeVarargs Annotation
* Collection Factory Methods
* Process API Improvement
* New Version-String Scheme
* JShell: The Java Shell (REPL)
* Process API Improvement
* Control Panel
* Stream API Improvement
* Installer Enhancement for Microsoft windows and many more

**Java 1.9**

**Java 9 Private Interface Methods**

In Java 9, we can create private methods inside an interface. Interface allows us to declare private methods that help to **share** common code between **non-abstract** methods.

Before Java 9, creating private methods inside an interface cause a compile time error. The following example is compiled using Java 8 compiler and throws a compile time error.

Java 9 Private Interface Methods Example

**interface** Sayable{

**default** **void** say() {

        saySomething();

    }

    // Private method inside interface

**private** **void** saySomething() {

        System.out.println("Hello... I'm private method");

    }

}

**public** **class** PrivateInterface **implements** Sayable {

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

        Sayable s = **new** PrivateInterface();

        s.say();

    }

}

**Example:2**

**interface** Sayable{

**default** **void** say() {

        saySomething(); // Calling private method

        sayPolitely(); //  Calling private static method

    }

    // Private method inside interface

**private** **void** saySomething() {

        System.out.println("Hello... I'm private method");

    }

    // Private static method inside interface

**private** **static** **void** sayPolitely() {

        System.out.println("I'm private static method");

    }

}

**public** **class** PrivateInterface **implements** Sayable {

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

        Sayable s = **new** PrivateInterface();

        s.say();

    }

}

# ================================================================

# Java 9 Try With Resource:

Java introduced **try-with-resource** feature in Java 7 that helps to close resource automatically after being used.

In other words, we can say that we don't need to close resources (file, connection, network etc) explicitly, try-with-resource close that automatically by using AutoClosable interface.

In Java 7, try-with-resources has a limitation that requires resource to declare locally within its block.

**Example Java 7 Resource Declared within resource block**

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**public** **class** FinalVariable {

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

**try**(FileOutputStream fileStream=**new** FileOutputStream("javatpoint.txt");){

             String greeting = "Welcome to javaTpoint.";

**byte** b[] = greeting.getBytes();

                fileStream.write(b);

                System.out.println("File written");

        }**catch**(Exception e) {

            System.out.println(e);

        }

    }

}

This code executes fine with Java 7 and even with Java 9 because Java maintains it's legacy.

But the below program would not work with Java 7 because **we can't put resource declared outside the try-with-resource.**

Java 7 Resource declared outside the resource block

If we do like the following code in Java 7, compiler generates an error message.

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**public** **class** FinalVariable {

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

        FileOutputStream fileStream=**new** FileOutputStream("javatpoint.txt");

**try**(fileStream){

             String greeting = "Welcome to javaTpoint.";

**byte** b[] = greeting.getBytes();

                fileStream.write(b);

                System.out.println("File written");

        }**catch**(Exception e) {

            System.out.println(e);

        }

    }

}

Output:

error: <identifier> expected

try(fileStream){

To deal with this error, try-with-resource is improved in Java 9 and now we can use reference of the resource that is not declared locally.

In this case, **if we execute the above program using Java 9 compiler, it will execute nicely without any compile error.**

**Java 9 try-with-resource Example**

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**public** **class** FinalVariable {

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

        FileOutputStream fileStream=**new** FileOutputStream("javatpoint.txt");

**try**(fileStream){

             String greeting = "Welcome to javaTpoint.";

**byte** b[] = greeting.getBytes();

                fileStream.write(b);

                System.out.println("File written");

        }**catch**(Exception e) {

            System.out.println(e);

        }

    }

}

**========================================================================**

**Java 9 Anonymous Inner Classes Improvement**

Java 9 introduced a new feature that allows us to use diamond operator with anonymous classes. Using the diamond with anonymous classes was not allowed in Java 7.

In Java 9, as long as the inferred type is denotable, we can use the diamond operator when we create an anonymous inner class.

Data types that can be written in Java program like int, String etc are called denotable types. Java 9 compiler is enough smart and now can infer type.

**Java 9 Anonymous Inner Classes Example**

abstract class ABCD<T>{

    abstract T show(T a, T b);

}

public class TypeInferExample {

    public static void main(String[] args) {

        ABCD<String> a = new ABCD<>() { // diamond operator is empty, compiler infer type

            String show(String a, String b) {

                return a+b;

            }

        };

        String result = a.show("Java","9");

        System.out.println(result);

    }

}

Output:

Java9

Although we can specifying type in diamond operator explicitly and compiler does not produce any error message. See, the following example, type is specified explicitly.

**Java 9 Anonymous Inner Classes Example**

abstract class ABCD<T>{

    abstract T show(T a, T b);

}

public class TypeInferExample {

    public static void main(String[] args) {

        ABCD<String> a = new ABCD<String>() { // diamond operator is not empty

            String show(String a, String b) {

                return a+b;

            }

        };

        String result = a.show("Java","9");

        System.out.println(result);

    }

}

And we get the same result.

Output:

Java9

What happens? If we compile the following code using Java 8.

Anonymous Inner Class Example

abstract class ABCD<T>{

    abstract T show(T a, T b);

}

public class TypeInferExample {

    public static void main(String[] args) {

        ABCD<String> a = new ABCD<>() { // diamond operator is empty

            String show(String a, String b) {

                return a+b;

            }

        };

        String result = a.show("Java","9");

        System.out.println(result);

    }

}

Java 8 compiler throws compile time error because it can't infer type. The error message looks like the below.

Output:

TypeInferExample.java:7: error: cannot infer type arguments for ABCD<T>

ABCD<String> a = new ABCD<>() {

^

reason: cannot use '<>' with anonymous inner classes

where T is a type-variable:

T extends Object declared in class ABCD

1 error

==========================================================

[**SafeVarargs Annotation**](https://www.javatpoint.com/java-9-safevarargs-annotation)

It is an annotation which applies on a method or constructor that takes **varargs parameters**. It is used to ensure that the method does not perform unsafe operations on its varargs parameters.

It was included in Java7 and can only be applied on

* Final methods
* Static methods
* Constructors

**From Java 9**, it can also be used with **private instance methods**.

**Note:** The @SafeVarargs annotation can be applied only to methods that cannot be overridden. Applying to the other methods will throw a compile time error.



Java 9 @SafeVarargs Annotation Example

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** SafeVar{

**private** **void** display(List<String>... products) { // Not using @SaveVarargs

**for** (List<String> product : products) {

            System.out.println(product);

        }

    }

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

        SafeVar p = **new** SafeVar();

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

        list.add("Laptop");

        list.add("Tablet");

        p.display(list);

    }

}

It produces **warning messages** at compile time, but compiles without errors.

Output:

At compile time:

Note: SafeVar.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

At runtime:[Laptop, Tablet]

**This is a compiler generated warning regarding unsafe varargs type.**

**To avoid it, we should use @SaveVarargs notation to the method, as we did in the following example**.

Java 9 @SafeVarargs Annotation Example

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** SafeVar{

    // Applying @SaveVarargs annotation

    @SafeVarargs

**private** **void** display(List<String>... products) { // Not using @SaveVarargs

**for** (List<String> product : products) {

            System.out.println(product);

        }

    }

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

        SafeVar p = **new** SafeVar();

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

        list.add("Laptop");

        list.add("Tablet");

        p.display(list);

    }

}

Now, compiler does not produce warning message, code compiles and runs successfully.

Output:

[Laptop, Tablet]

Note: To apply @SaveVarargs annotation on private instance methods, compile code using Java 9 or higher versions only.

**Q: What happens? If we compile the following code by using older versions of Java.**

Output: SafeVar.java:6: error: Invalid SafeVarargs annotation. Instance method display(List<String>...) is not final.

private void display(List<String>... products) {

^

Note: SafeVar.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

1 error

**====================================**

**Java 9 Factory Methods**

Java 9 Collection library includes static factory methods for List, Set and Map interface. These methods are useful to create small number of collection.

Suppose, if we want to create a list of 5 elements, we need to write the following code.

Java List Example

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** FactoryMethodsExample {

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

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

        list.add("Java");

        list.add("JavaFX");

        list.add("Spring");

        list.add("Hibernate");

        list.add("JSP");

**for**(String l : list){

            System.out.println(l);

        }

    }

}

Output:

Java

JavaFX

Spring

Hibernate

JSP

**Factory Methods for Collection**

Factory methods are special type of static methods that are used to create **unmodifiable instances** of collections. It means we can use these methods to create list, set and map of small number of elements.

It is unmodifiable, so adding new element will throw **java.lang.UnsupportedOperationException**

Each interface has it's own factory methods, we are listing all the methods in the following tables.

Factory Methods of List Interface

|  |  |  |
| --- | --- | --- |
| Modifiers | Methods | Description |
| static <E> List<E> | Of() | It It returns an immutable list containing zero elements. |
| static <E> List<E> | of(E e1) | It It returns an immutable list containing one element. |
| static <E> List<E> | of(E... elements) | It It returns an immutable list containing an arbitrary number of elements. |
| static <E> List<E> | of(E e1, E e2) | It It returns an immutable list containing two elements. |
| static <E> List<E> | of(E e1, E e2, E e3) | It It returns an immutable list containing three elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4) | It It returns an immutable list containing four elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5) | It It returns an immutable list containing five elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6) | It It returns an immutable list containing six elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7) | It It returns an immutable list containing seven elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8) | It It returns an immutable list containing eight elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9) | It It returns an immutable list containing nine elements. |
| static <E> List<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10) | It It returns an immutable list containing ten elements. |

Java 9 List Factory Method Example

In Java 9, we can write this code in vary simple manner with the help of **List.of() factory method.**

**import** java.util.List;

**public** **class** FactoryMethodsExample {

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

        List<String> list = List.of("Java","JavaFX","Spring","Hibernate","JSP");

**for**(String l:list) {

            System.out.println(l);

        }

    }

}

Output:

Java

JavaFX

Spring

Hibernate

JSP

**Java 9 Set Interface**

Java Set interface provides a **Set.of() static factory method** which is used to create immutable set. The set instance created by this method has the following characteristcis.

* It is immutable
* No null elements
* It is serializable if all elements are serializable.
* No duplicate elements.
* The iteration order of set elements is unspecified and is subject to change.

**Java 9 Set Interface Factory Methods**

The following table contains the factory methods for Set interface.

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| static <E> Set<E> | of() | It It returns an immutable set containing zero elements. |
| static <E> Set<E> | of(E e1) | It It returns an immutable set containing one element. |
| static <E> Set<E> | of(E... elements) | It It returns an immutable set containing an arbitrary number of elements. |
| static <E> Set<E> | of(E e1, E e2) | It It returns an immutable set containing two elements. |
| static <E> Set<E> | of(E e1, E e2, E e3) | It It returns an immutable set containing three elements. |
| static <E> Set<E> | of(E e1, E e2, E e3, E e4) | It It returns an immutable set containing four elements. |
| static <E> Set<E> | of(E e1, E e2, E e3, E e4, E e5) | It It returns an immutable set containing five elements. |
| static <E> Set<E> | It It returns an immutable set containing six elements. |  |
| static <E> Set<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7) | It It returns an immutable set containing seven elements. |
| static <E> Set<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8) | It It returns an immutable set containing eight elements. |
| static <E> Set<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9) | It It returns an immutable set containing nine elements. |
| static <E> Set<E> | of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10) | It It returns an immutable set containing ten elements. |

**Java 9 Set Interface Factory Methods Example**

**import** java.util.Set;

**public** **class** FactoryMethodsExample {

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

        Set<String> set = Set.of("Java","JavaFX","Spring","Hibernate","JSP");

**for**(String l:set) {

            System.out.println(l);

        }

    }

}

Output:

Spring

JavaFX

JSP

Java

Hibernate

Java 9 Map Interface Factory Methods

In Java 9, Map includes Map.of() and Map.ofEntries() static factory methods that provide a convenient way to creae immutable maps.

Map created by these methods has the following characteristics.

It is immutable

It does not allow null keys and values

It is serializable if all keys and values are serializable

It rejects duplicate keys at creation time

The iteration order of mappings is unspecified and is subject to change.

Java 9 Map Interface Factory Methods

The following table contains the factory methods for Map interface.

|  |  |  |
| --- | --- | --- |
| Modifier and Type | Method | Description |
| static <K,V> Map<K,V> | of() | It returns an immutable map containing zero mappings. |
| static <K,V> Map<K,V> | of(K k1, V v1) | It returns an immutable map containing a single mapping. |
| static <K,V> Map<K,V> | of(K k1, V v1, K k2, V v2) | It returns an immutable map containing two mappings. |
| static <K,V> Map<K,V> | of(K k1, V v1, K k2, V v2, K k3, V v3) | It returns an immutable map containing three mappings. |
| static <K,V> Map<K,V> | of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4) | It returns an immutable map containing four mappings. |
| static <K,V> Map<K,V> | of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5) | It returns an immutable map containing five mappings. |
| static <K,V> Map<K,V> | of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6) | It returns an immutable map containing six mappings. |
| static <K,V> Map<K,V> | of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7) | It returns an immutable map containing seven mappings. |
| static <K,V> Map<K,V> | of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8) | It returns an immutable map containing eight mappings. |
| static <K,V> Map<K,V> | of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9) | It returns an immutable map containing nine mappings. |
| static <K,V> Map<K,V> | of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4, K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8, K k9, V v9, K k10, V v10) | It returns an immutable map containing ten mappings. |
| static <K,V> Map<K,V> | ofEntries(Map.Entry<? extends K,? extends V>... entries) | It returns an immutable map containing keys and values extracted from the given entries. |
|  |  |  |

Java 9 Map Interface Factory Methods Example

**import** java.util.Map;

**public** **class** FactoryMethodsExample {

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

        Map<Integer,String> map = Map.of(101,"JavaFX",102,"Hibernate",103,"Spring MVC");

**for**(Map.Entry<Integer, String> m : map.entrySet()){

            System.out.println(m.getKey()+" "+m.getValue());

        }

    }

}

Output:

102 Hibernate

103 Spring MVC

101 JavaFX

Java 9 Map Interface ofEntries() Method Example

In Java 9, apart from static **Map.of()** methods, Map interface includes one more static method **Map.ofEntries()**. This method is used to create a map of **Map.Entry**instances.

In the following example, we are creating map instance with the help of multiple map.entry instances.

**import** java.util.Map;

**public** **class** FactoryMethodsExample {

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

        // Creating Map Entry

        Map.Entry<Integer, String> e1 = Map.entry(101, "Java");

        Map.Entry<Integer, String> e2 = Map.entry(102, "Spring");

        // Creating Map using map entries

        Map<Integer, String> map = Map.ofEntries(e1,e2);

        // Iterating Map

**for**(Map.Entry<Integer, String> m : map.entrySet()){

            System.out.println(m.getKey()+" "+m.getValue());

        }

    }

}

Output:

102 Spring

101 Java