Java 8 Features

Java 9 Features

Java 10 Features

Java 11 Features

Some of the important Java 11 features are:

1. Running Java File with single command
2. New utility methods in String class
3. Local-Variable Syntax for Lambda Parameters
4. Nested Based Access Control
5. Flight Recorder
6. Reading/Writing Strings to and from the Files
7. Not Predicate
8. HTTP Client
9. File APIs
10. Optional Class

Java 12 Features

Some of the important Java 12 features are;

1. Switch Expressions
2. File mismatch() Method
3. Compact Number Formatting
4. Teeing Collectors in Stream API
5. Java Strings New Methods - indent(), transform(), describeConstable(), and resolveConstantDesc().
6. Pattern Matching for instanceof

Java 13 Features

1. Text Blocks
2. New Methods in String Class for Text Blocks
3. Switch Expressions Enhancements
4. Reimplement the Legacy Socket API

1. Lambda Expressions

Lambda expression helps us to write our code in functional style. It provides a clear and concise way to implement SAM interface (Single Abstract Method) by using an expression. It is very useful in collection library in which it helps to iterate, filter and extract data.

Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

Java Lambda Expression Syntax

1. (Argument-list) -> {body}

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Examples:

```
Runnable r1 = () -> {
                        System.out.println("My Runnable");
                };
```

Without Lambda Expression

```
interface Drawable{
    public void draw();
}
public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){
            public void draw(){System.out.println("Drawing "+width);}
        };
        d.draw();
    }
}
```

Java Lambda Expression Example

Now, we are going to implement the above example with the help of Java lambda expression.

```
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

2.Method References
A method reference is the shorthand syntax for a lambda expression that executes just **ONE** method. Here's the general syntax of a method reference:
Object :: methodName

Types of Method References

There are following types of method references in java:

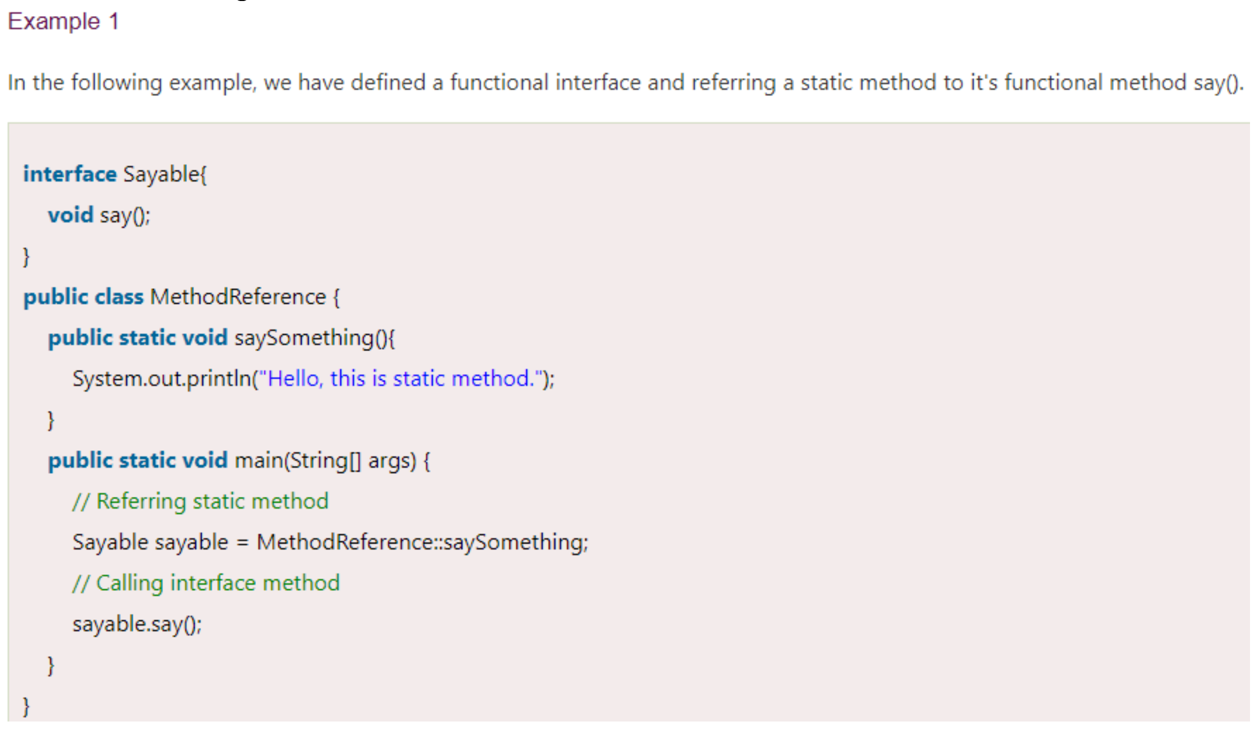
1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

2.1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

1. ContainingClass::staticMethodName


Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

```
interface Sayable{
    void say();
}
public class MethodReference {
    public static void saySomething(){
        System.out.println("Hello, this is static method.");
    }
    public static void main(String[] args) {
        // Referring static method
        Sayable sayable = MethodReference::saySomething;
        // Calling interface method
        sayable.say();
    }
}
```

Example 2

In the following example, we are using predefined functional interface Runnable to refer static method.

```
public class MethodReference2 {
    public static void ThreadStatus(){
        System.out.println("Thread is running...");
    }
    public static void main(String[] args) {
        Thread t2=new Thread(MethodReference2::ThreadStatus);
        t2.start();
    }
}
```

Example 3

You can also use predefined functional interface to refer methods. In the following example, we are using BiFunction interface and using it's apply() method.

```
import java.util.function.BiFunction;
class Arithmetic{
public static int add(int a, int b){
return a+b;
}
}
public class MethodReference3 {
public static void main(String[] args) {
BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;
int result = adder.apply(10, 20);
System.out.println(result);
}
}
```

Example 4

You can also override static methods by referring methods. In the following example, we have defined and overloaded three add methods.

```
import java.util.function.BiFunction;
class Arithmetic{
public static int add(int a, int b){
return a+b;
}
public static float add(int a, float b){
return a+b;
}
public static float add(float a, float b){
return a+b;
}
}
public class MethodReference4 {
public static void main(String[] args) {
BiFunction<Integer, Integer, Integer>adder1 = Arithmetic::add;
BiFunction<Integer, Float, Float>adder2 = Arithmetic::add;
BiFunction<Float, Float, Float>adder3 = Arithmetic::add;
int result1 = adder1.apply(10, 20);
float result2 = adder2.apply(10, 20.0f);
float result3 = adder3.apply(10.0f, 20.0f);
System.out.println(result1);
System.out.println(result2);
System.out.println(result3);
}
}
```

2.2) Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

1. containingObject::instanceMethodName

Example 1

In the following example, we are referring instance (non-static) method. Runnable interface contains only one abstract method. So, we can use it as functional interface.

```
public class InstanceMethodReference2 {
    public void printnMsg(){
        System.out.println("Hello, this is instance method");
    }
    public static void main(String[] args) {
    Thread t2=new Thread(new InstanceMethodReference2()::printnMsg);
        t2.start();
    }
}
```

Example 2

In the following example, we are using BiFunction interface. It is a predefined interface and contains a functional method apply(). Here, we are referring add method to apply method.

```
import java.util.function.BiFunction;
class Arithmetic{
public int add(int a, int b){
return a+b;
}
}
public class InstanceMethodReference3 {
public static void main(String[] args) {
BiFunction<Integer, Integer, Integer>adder = new Arithmetic()::add;
int result = adder.apply(10, 20);
System.out.println(result);
}
}
```

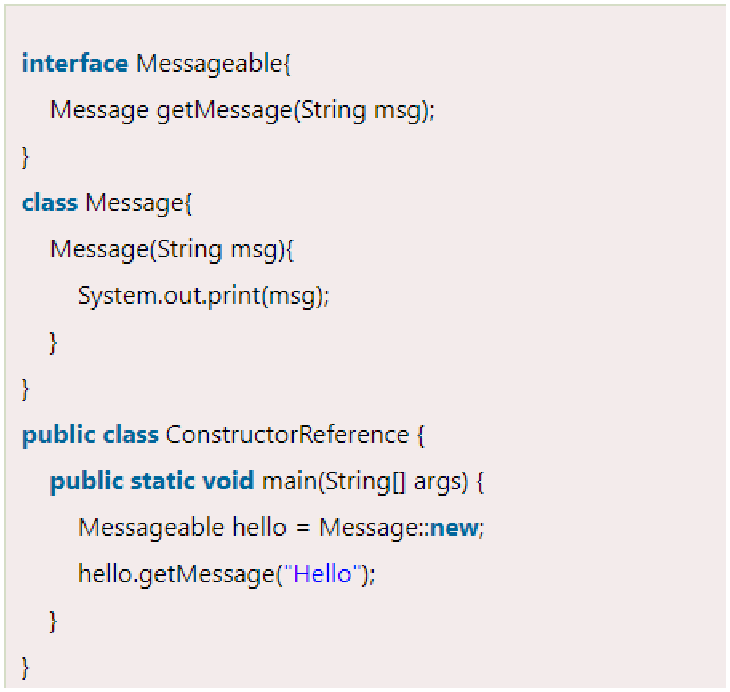
2.3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

1. ClassName::**new**

Example



```
interface Messageable{
    Message getMessage(String msg);
}
class Message{
    Message(String msg){
        System.out.print(msg);
    }
}
public class ConstructorReference {
    public static void main(String[] args) {
        Messageable hello = Message::new;
        hello.getMessage("Hello");
    }
}
```

Functional Interface

An Interface that contains only one abstract method is known as functional interface. It can have any number of default and static methods. It can also declare methods of object class.
Functional interfaces are also known as Single Abstract Method Interfaces (SAM Interfaces). We don’t need to use @FunctionalInterface annotation to mark an interface as a Functional Interface.

@FunctionalInterface annotation is a facility to avoid the accidental addition of abstract methods in the functional interfaces. You can think of it like **@Override** annotation and its best practice to use it. **java.lang.Runnable** with a single abstract method **run()** is a great example of a functional interface. One of the major benefits of the functional interface is the possibility to use lambda expressions to instantiate them.

A functional interface can have methods of object class. See in the following example.

Example 2

```
@FunctionalInterface
interface sayable{
    void say(String msg);   // abstract method
    // It can contain any number of Object class methods.
    int hashCode();
    String toString();
    boolean equals(Object obj);
}
public class FunctionalInterfaceExample2 implements sayable{
    public void say(String msg){
        System.out.println(msg);
    }
    public static void main(String[] args) {
        FunctionalInterfaceExample2 fie = new FunctionalInterfaceExample2();
        fie.say("Hello there");
    }
}
```

Invalid Functional Interface

A functional interface can extends another interface only when it does not have any abstract method.

```
interface sayable{
    void say(String msg);   // abstract method
}
@FunctionalInterface
interface Doable extends sayable{
    // Invalid '@FunctionalInterface' annotation; Doable is not a functional interface
    void doIt();
}
```

Output:

```
compile-time error
```

Java Predefined-Functional Interfaces

Java provides predefined functional interfaces to deal with functional programming by using lambda and method references.

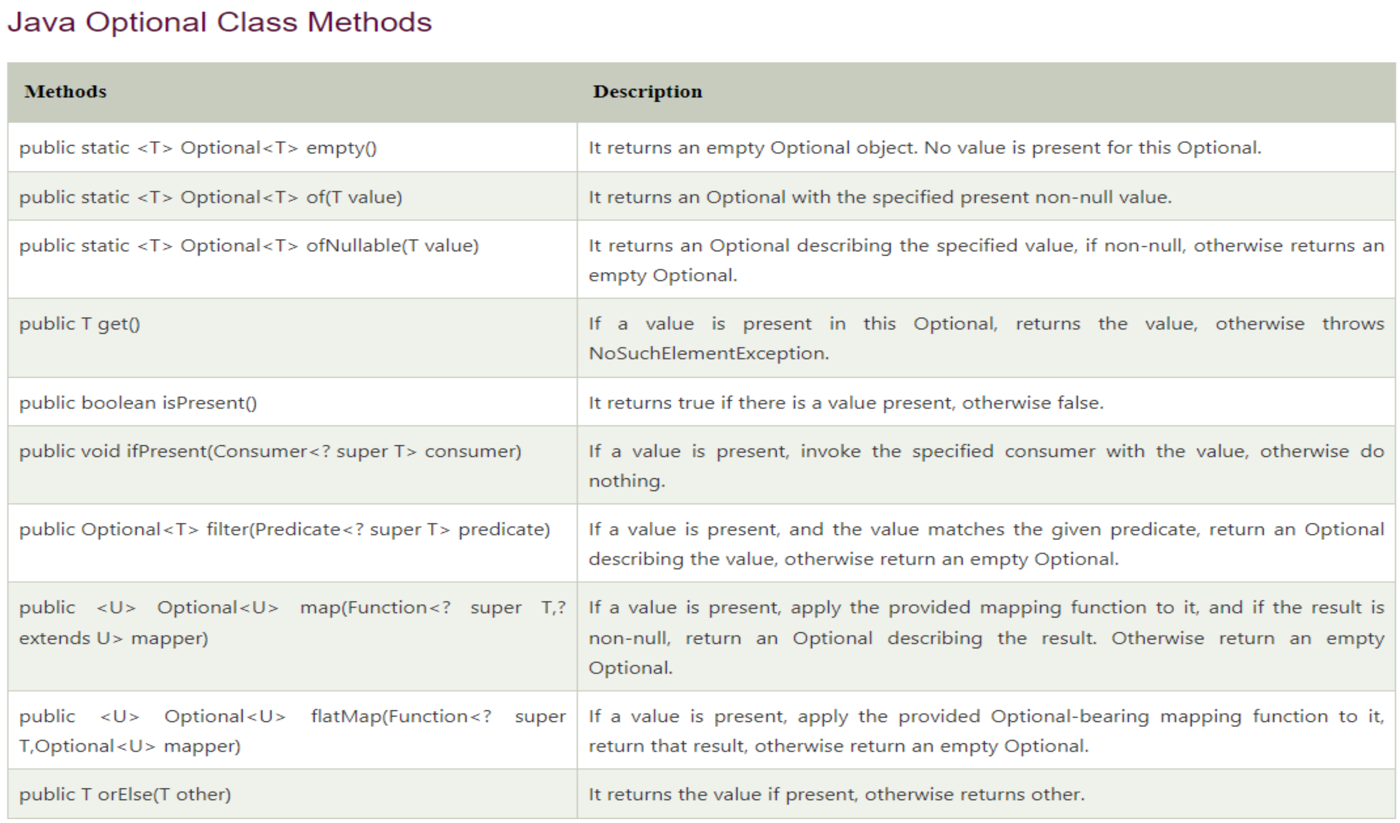
You can also define your own custom functional interface. Following is the list of functional interface which are placed in java.util.function package.

Interface	Description
BiConsumer<T,U>	It represents an operation that accepts two input arguments and returns no result.
Consumer<T>	It represents an operation that accepts a single argument and returns no result.
Function<T,R>	It represents a function that accepts one argument and returns a result.
Predicate<T>	It represents a predicate (boolean-valued function) of one argument.
BiFunction<T,U,R>	It represents a function that accepts two arguments and returns a a result.
BinaryOperator<T>	It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands.
BiPredicate<T,U>	It represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	It represents a supplier of boolean-valued results.
DoubleBinaryOperator	It represents an operation upon two double type operands and returns a double type value.
DoubleConsumer	It represents an operation that accepts a single double type argument and returns no result.
DoubleFunction<R>	It represents a function that accepts a double type argument and produces a result.
DoublePredicate	It represents a predicate (boolean-valued function) of one double type argument.
DoubleSupplier	It represents a supplier of double type results.
DoubleToIntFunction	It represents a function that accepts a double type argument and produces an int type result.

DoubleToLongFunction	It represents a function that accepts a double type argument and produces a long type result.
DoubleUnaryOperator	It represents an operation on a single double type operand that produces a double type result.
IntBinaryOperator	It represents an operation upon two int type operands and returns an int type result.
IntConsumer	It represents an operation that accepts a single integer argument and returns no result.
IntFunction<R>	It represents a function that accepts an integer argument and returns a result.
IntPredicate	It represents a predicate (boolean-valued function) of one integer argument.
IntSupplier	It represents a supplier of integer type.
IntToDoubleFunction	It represents a function that accepts an integer argument and returns a double.
IntToLongFunction	It represents a function that accepts an integer argument and returns a long.
IntUnaryOperator	It represents an operation on a single integer operand that produces an integer result.
LongBinaryOperator	It represents an operation upon two long type operands and returns a long type result.
LongConsumer	It represents an operation that accepts a single long type argument and returns no result.
LongFunction<R>	It represents a function that accepts a long type argument and returns a result.
LongPredicate	It represents a predicate (boolean-valued function) of one long type argument.
LongSupplier	It represents a supplier of long type results.
LongToDoubleFunction	It represents a function that accepts a long type argument and returns a result of double type.
LongToIntFunction	It represents a function that accepts a long type argument and returns an integer result.
LongUnaryOperator	It represents an operation on a single long type operand that returns a long type result.
ObjDoubleConsumer<T>	It represents an operation that accepts an object and a double argument, and returns no result.
ObjIntConsumer<T>	It represents an operation that accepts an object and an integer argument. It does not return result.
ObjLongConsumer<T>	It represents an operation that accepts an object and a long argument, it returns no result.
Supplier<T>	It represents a supplier of results.
ToDoubleBiFunction<T,U>	It represents a function that accepts two arguments and produces a double type result.
ToDoubleFunction<T>	It represents a function that returns a double type result.
ToIntBiFunction<T,U>	It represents a function that accepts two arguments and returns an integer.
ToIntFunction<T>	It represents a function that returns an integer.
ToLongBiFunction<T,U>	It represents a function that accepts two arguments and returns a result of long type.
ToLongFunction<T>	It represents a function that returns a result of long type.
UnaryOperator<T>	It represents an operation on a single operand that returnsa a result of the same type as its operand.

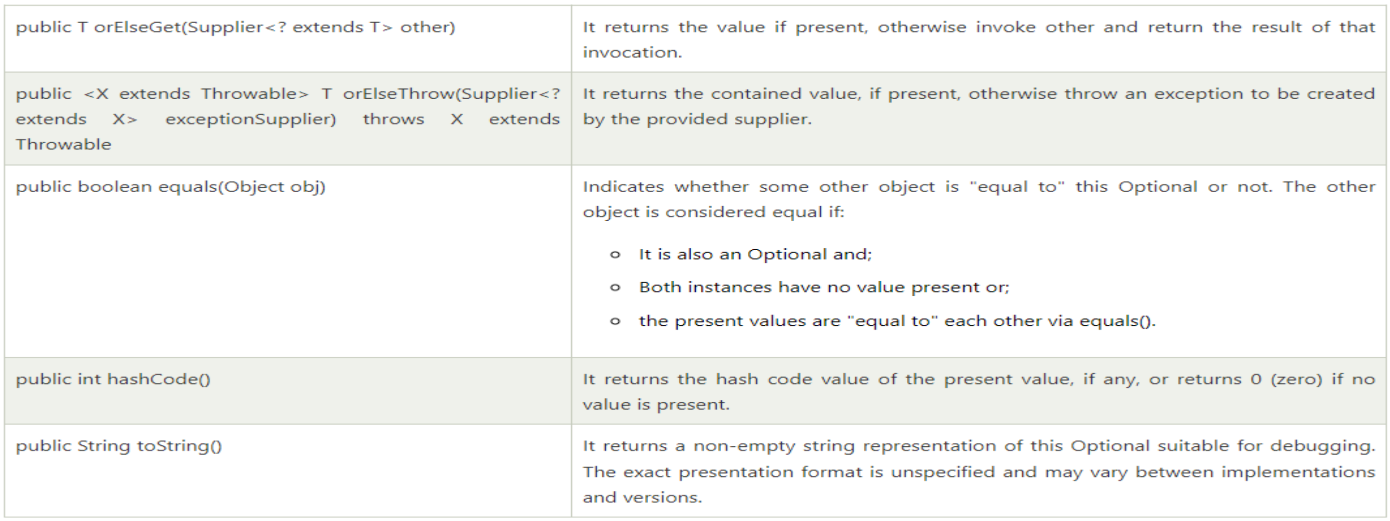
Optional

Java introduced a new class Optional in Java 8. It is a public final class which is used to deal with NullPointerException in Java application. We must import *java.util* package to use this class. It provides methods to check the presence of value for particular variable.



Java Optional Class Methods

Methods	Description
public static <T> Optional<T> empty()	It returns an empty Optional object. No value is present for this Optional.
public static <T> Optional<T> of(T value)	It returns an Optional with the specified present non-null value.
public static <T> Optional<T> ofNullable(T value)	It returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
public T get()	If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
public boolean isPresent()	It returns true if there is a value present, otherwise false.
public void ifPresent(Consumer<? super T> consumer)	If a value is present, invoke the specified consumer with the value, otherwise do nothing.
public Optional<T> filter(Predicate<? super T> predicate)	If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional.
public <U> Optional<U> map(Function<? super T,? extends U> mapper)	If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result. Otherwise return an empty Optional.
public <U> Optional<U> flatMap(Function<? super T,Optional<U> mapper)	If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional.
public T orElse(T other)	It returns the value if present, otherwise returns other.



public T orElseGet(Supplier<? extends T> other)	It returns the value if present, otherwise invoke other and return the result of that invocation.
public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X extends Throwable	It returns the contained value, if present, otherwise throw an exception to be created by the provided supplier.
public boolean equals(Object obj)	Indicates whether some other object is "equal to" this Optional or not. The other object is considered equal if: ◦ It is also an Optional and; ◦ Both instances have no value present or; ◦ the present values are "equal to" each other via equals().
public int hashCode()	It returns the hash code value of the present value, if any, or returns 0 (zero) if no value is present.
public String toString()	It returns a non-empty string representation of this Optional suitable for debugging. The exact presentation format is unspecified and may vary between implementations and versions.

Example: Java Program without using Optional

In the following example, we are not using Optional class. This program terminates abnormally and throws a nullPointerException.

```
public class OptionalExample {
    public static void main(String[] args) {
        String[] str = new String[10];
        String lowercaseString = str[5].toLowerCase();
        System.out.print(lowercaseString);
    }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException
        at lambdaExample.OptionalExample.main(OptionalExample.java:6)
```

To avoid the abnormal termination, we use Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

Java Optional Example: If Value is not Present

```
import java.util.Optional;
public class OptionalExample {
    public static void main(String[] args) {
        String[] str = new String[10];
        Optional<String> checkNull = Optional.ofNullable(str[5]);
        if(checkNull.isPresent()){  // check for value is present or not
            String lowercaseString = str[5].toLowerCase();
            System.out.print(lowercaseString);
        }else
            System.out.println("string value is not present");
    }
}
```

Output:

```
string value is not present
```

Java Optional Example: If Value is Present

```java
import java.util.Optional;
public class OptionalExample {
    public static void main(String[] args) {
        String[] str = new String[10];
        str[5] = "JAVA OPTIONAL CLASS EXAMPLE";// Setting value for 5th index
        Optional<String> checkNull = Optional.ofNullable(str[5]);
        if(checkNull.isPresent()){  // It Checks, value is present or not
            String lowercaseString = str[5].toLowerCase();
            System.out.print(lowercaseString);
        }else
            System.out.println("String value is not present");
    }
}
```

Output:

```
java optional class example
```

Another Java Optional Example

```java
import java.util.Optional;
public class OptionalExample {
    public static void main(String[] args) {
        String[] str = new String[10];
        str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
        Optional<String> checkNull = Optional.ofNullable(str[5]);
        checkNull.ifPresent(System.out::println);   // printing value by using method reference
        System.out.println(checkNull.get());    // printing value by using get method
        System.out.println(str[5].toLowerCase());
    }
}
```

Output:

```
JAVA OPTIONAL CLASS EXAMPLE
JAVA OPTIONAL CLASS EXAMPLE
java optional class example
```

Java Optional Methods Example

```java
import java.util.Optional;
public class OptionalExample {
    public static void main(String[] args) {
        String[] str = new String[10];
        str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
        // It returns an empty instance of Optional class
        Optional<String> empty = Optional.empty();
        System.out.println(empty);
        // It returns a non-empty Optional
        Optional<String> value = Optional.of(str[5]);
        // If value is present, it returns an Optional otherwise returns an empty Optional
        System.out.println("Filtered value: "+value.filter((s)->s.equals("Abc")));
        System.out.println("Filtered value: "+value.filter((s)->s.equals("JAVA OPTIONAL CLASS EXAMPLE")));
        // It returns value of an Optional. if value is not present, it throws an NoSuchElementException
        System.out.println("Getting value: "+value.get());
        // It returns hashCode of the value
        System.out.println("Getting hashCode: "+value.hashCode());
        // It returns true if value is present, otherwise false
        System.out.println("Is value present: "+value.isPresent());
        // It returns non-empty Optional if value is present, otherwise returns an empty Optional
        System.out.println("Nullable Optional: "+Optional.ofNullable(str[5]));
        // It returns value if available, otherwise returns specified value,
        System.out.println("orElse: "+value.orElse("Value is not present"));
        System.out.println("orElse: "+empty.orElse("Value is not present"));
        value.ifPresent(System.out::println);   // printing value by using method reference
    }
}
```

Output:

```
Optional.empty
Filtered value: Optional.empty
Filtered value: Optional[JAVA OPTIONAL CLASS EXAMPLE]
Getting value: JAVA OPTIONAL CLASS EXAMPLE
Getting hashCode: -619947648
Is value present: true
Nullable Optional: Optional[JAVA OPTIONAL CLASS EXAMPLE]
orElse: JAVA OPTIONAL CLASS EXAMPLE
orElse: Value is not present
JAVA OPTIONAL CLASS EXAMPLE
```

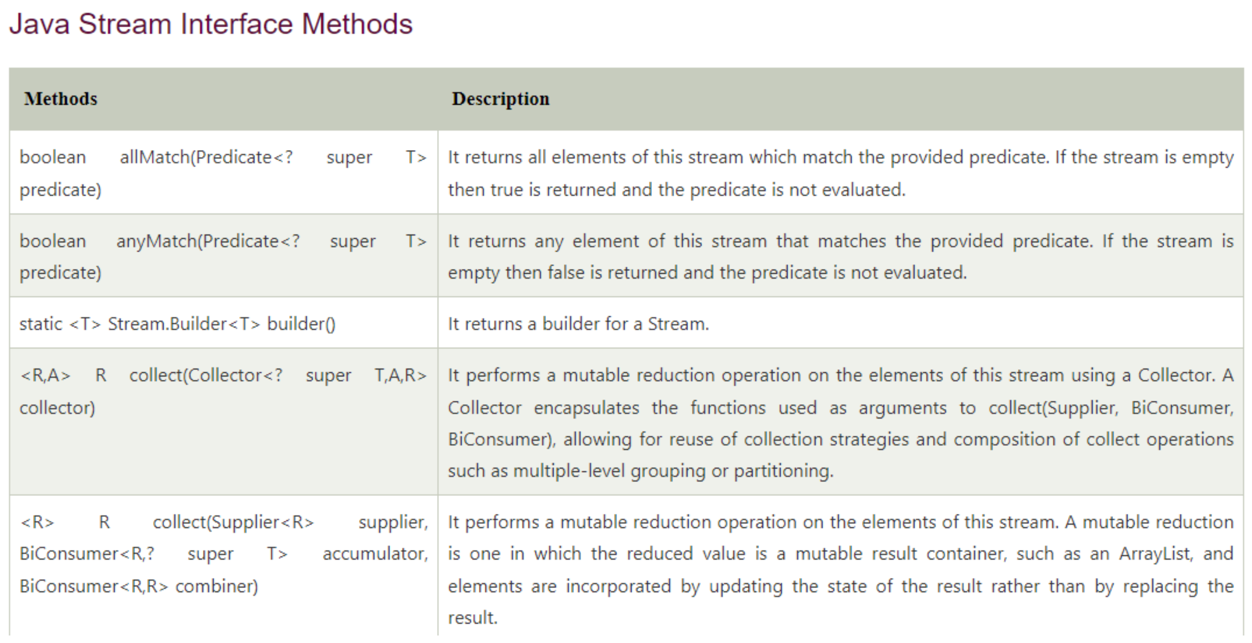
Stream API

Java 8 java.util.stream package consists of classes, interfaces and an **enum** to allow functional-style operations on the elements. It performs lazy computation. So, it executes only when it requires.

Stream provides following features:

- Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
- Stream is functional in nature. Operations performed on a stream does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- Stream is lazy and evaluates code only when required.
- The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have applied various operations with the help of stream.

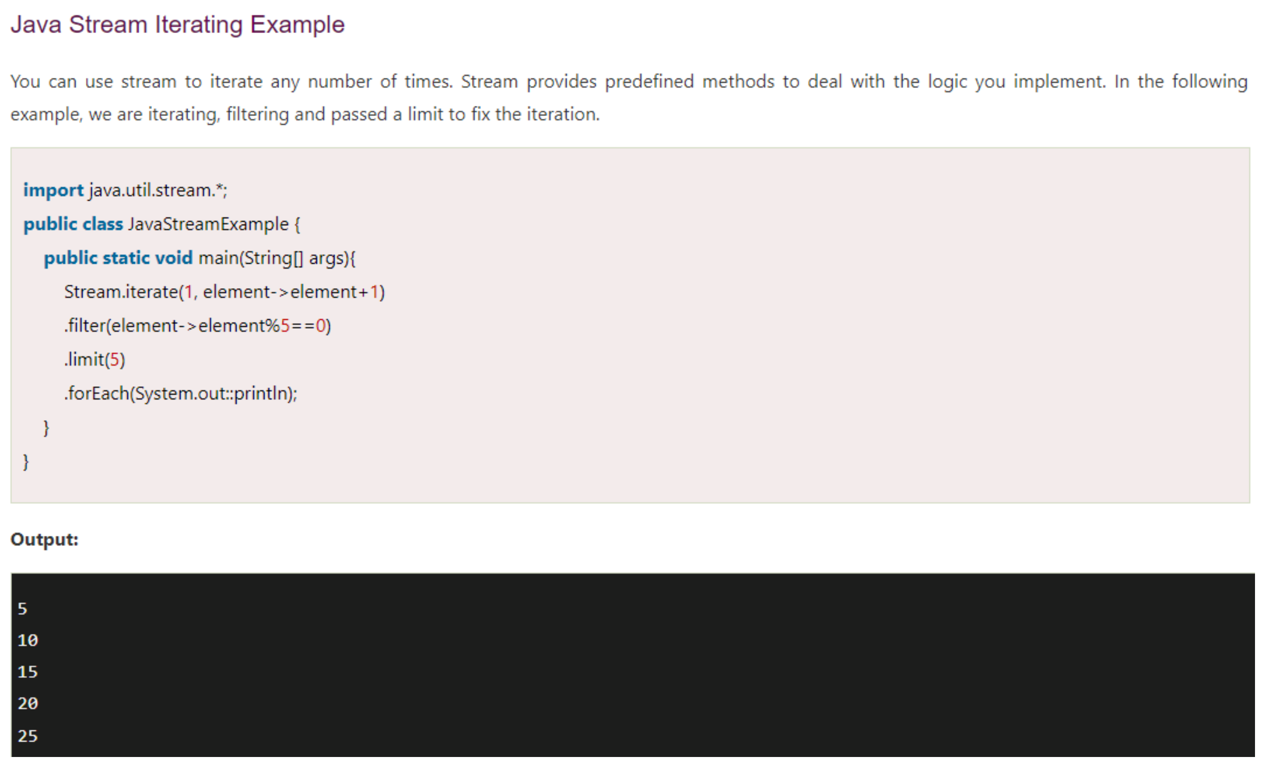


Java Stream Interface Methods

Methods	Description
boolean allMatch(Predicate<? super T> predicate)	It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
boolean anyMatch(Predicate<? super T> predicate)	It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated.
static <T> Stream.Builder<T> builder()	It returns a builder for a Stream.
<R,A> R collect(Collector<? super T,A,R> collector)	It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.
<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)	It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result.

static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)	It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.
long count()	It returns the count of elements in this stream. This is a special case of a reduction.
Stream<T> distinct()	It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
static <T> Stream<T> empty()	It returns an empty sequential Stream.
Stream<T> filter(Predicate<? super T> predicate)	It returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T> findAny()	It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T> findFirst()	It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned.
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)	It returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)	It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.)
IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)	It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)	It returns a LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.)
void forEach(Consumer<? super T> action)	It performs an action for each element of this stream.
void forEachOrdered(Consumer<? super T> action)	It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
static <T> Stream<T> generate(Supplier<T> s)	It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc.
static <T> Stream<T> iterate(T seed,UnaryOperator<T> f)	It returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
Stream<T> limit(long maxSize)	It returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
<R> Stream<R> map(Function<? super T,? extends R> mapper)	It returns a stream consisting of the results of applying the given function to the elements of this stream.

DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)	It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
IntStream mapToInt(ToIntFunction<? super T> mapper)	It returns an IntStream consisting of the results of applying the given function to the elements of this stream.
LongStream mapToLong(ToLongFunction<? super T> mapper)	It returns a LongStream consisting of the results of applying the given function to the elements of this stream.
Optional<T> max(Comparator<? super T> comparator)	It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction.
Optional<T> min(Comparator<? super T> comparator)	It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction.
boolean noneMatch(Predicate<? super T> predicate)	It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated.
@SafeVarargs static <T> Stream<T> of(T... values)	It returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T> of(T t)	It returns a sequential Stream containing a single element.
Stream<T> peek(Consumer<? super T> action)	It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
Optional<T> reduce(BinaryOperator<T> accumulator)	It performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
T reduce(T identity, BinaryOperator<T> accumulator)	It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)	It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.
Stream<T> skip(long n)	It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned.
Stream<T> sorted()	It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.
Stream<T> sorted(Comparator<? super T> comparator)	It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
Object[] toArray()	It returns an array containing the elements of this stream.
<A> A[] toArray(IntFunction<A[]> generator)	It returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.


Java Stream Iterating Example

You can use stream to iterate any number of times. Stream provides predefined methods to deal with the logic you implement. In the following example, we are iterating, filtering and passed a limit to fix the iteration.

```
import java.util.stream.*;
public class JavaStreamExample {
   public static void main(String[] args){
      Stream.iterate(1, element->element+1)
      .filter(element->element%5==0)
      .limit(5)
      .forEach(System.out::println);
   }
}
```

Output:

```
5
10
15
20
25
```

For more examples, visit below provided link

https://www.javatpoint.com/java-8-stream

Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default keyword are known as default methods. These methods are non-abstract methods and can have method body.

Java Default Method Example

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

Let's see a simple

```java
interface Sayable{
    // Default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
}
public class DefaultMethods implements Sayable{
    public void sayMore(String msg){        // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();   // calling default method
        dm.sayMore("Work is worship");  // calling abstract method

    }
}
```

Output:

```
Hello, this is default method
Work is worship
```

Static Methods inside Java 8 Interface

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

```
interface Sayable{
    // default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
    // static method
    static void sayLouder(String msg){
        System.out.println(msg);
    }
}
public class DefaultMethods implements Sayable{
    public void sayMore(String msg){     // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();                    // calling default method
        dm.sayMore("Work is worship");      // calling abstract method
        Sayable.sayLouder("Helloooo...");   // calling static method
    }
}
```

Output:

```
Hello there
Work is worship
Helloooo...
```

Abstract Class vs Java 8 Interface

After having default and static methods inside the interface, we think about the need of abstract class in Java. An interface and an abstract class are almost similar except that you can create constructor in the abstract class whereas you can't do this in interface.

Java Base64 Encoding and Decoding

Java provides a class Base64 to deal with encryption and decryption. You need to import java.util.Base64 class in your source file to use its methods.

This class provides three different encoders and decoders to encrypt information at each level.

Basic Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 and RFC 2045 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

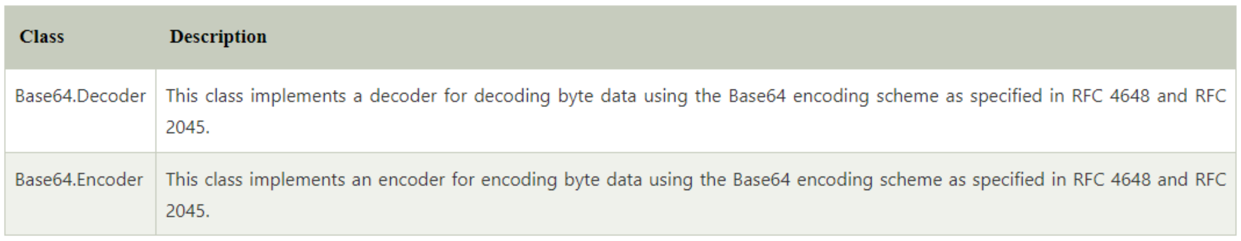
URL and Filename Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

MIME

It uses the Base64 alphabet as specified in RFC 2045 for encoding and decoding operations. The encoded output must be represented in lines of no more than 76 characters each and uses a carriage return '\r' followed immediately by a linefeed '\n' as the line separator. No line separator is added to the end of the encoded output. All line separators or other characters not found in the base64 alphabet table are ignored in decoding operation.

Nested Classes of Base64



Class	Description
Base64.Decoder	This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.
Base64.Encoder	This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Base64 Methods

Methods	Description
public static Base64.Decoder getDecoder()	It returns a Base64.Decoder that decodes using the Basic type base64 encoding scheme.
public static Base64.Encoder getEncoder()	It returns a Base64.Encoder that encodes using the Basic type base64 encoding scheme.
public static Base64.Decoder getUrlDecoder()	It returns a Base64.Decoder that decodes using the URL and Filename safe type base64 encoding scheme.
public static Base64.Decoder getMimeDecoder()	It returns a Base64.Decoder that decodes using the MIME type base64 decoding scheme.
public static Base64.Encoder getMimeEncoder()	It Returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme.
public static Base64.Encoder getMimeEncoder(int lineLength, byte[] lineSeparator)	It returns a Base64.Encoder that encodes using the MIME type base64 encoding scheme with specified line length and line separators.
public static Base64.Encoder getUrlEncoder()	It returns a Base64.Encoder that encodes using the URL and Filename safe type base64 encoding scheme.

Base64.Decoder Methods

Methods	Description
public byte[] decode(byte[] src)	It decodes all bytes from the input byte array using the Base64 encoding scheme, writing the results into a newly-allocated output byte array. The returned byte array is of the length of the resulting bytes.
public byte[] decode(String src)	It decodes a Base64 encoded String into a newly-allocated byte array using the Base64 encoding scheme.
public int decode(byte[] src, byte[] dst)	It decodes all bytes from the input byte array using the Base64 encoding scheme, writing the results into the given output byte array, starting at offset 0.
public ByteBuffer decode(ByteBuffer buffer)	It decodes all bytes from the input byte buffer using the Base64 encoding scheme, writing the results into a newly-allocated ByteBuffer.
public InputStream wrap(InputStream is)	It returns an input stream for decoding Base64 encoded byte stream.

Base64.Encoder Methods

Methods	Description
public byte[] encode(byte[] src)	It encodes all bytes from the specified byte array into a newly-allocated byte array using the Base64 encoding scheme. The returned byte array is of the length of the resulting bytes.
public int encode(byte[] src, byte[] dst)	It encodes all bytes from the specified byte array using the Base64 encoding scheme, writing the resulting bytes to the given output byte array, starting at offset 0.
public String encodeToString(byte[] src)	It encodes the specified byte array into a String using the Base64 encoding scheme.
public ByteBuffer encode(ByteBuffer buffer)	It encodes all remaining bytes from the specified byte buffer into a newly-allocated ByteBuffer using the Base64 encoding scheme. Upon return, the source buffer's position will be updated to its limit; its limit will not have been changed. The returned output buffer's position will be zero and its limit will be the number of resulting encoded bytes.
public OutputStream wrap(OutputStream os)	It wraps an output stream for encoding byte data using the Base64 encoding scheme.
public Base64.Encoder withoutPadding()	It returns an encoder instance that encodes equivalently to this one, but without adding any padding character at the end of the encoded byte data.

Java Base64 Example: URL Encoding and Decoding

```
import java.util.Base64;
publicclass Base64BasicEncryptionExample {
    publicstaticvoid main(String[] args) {
        // Getting encoder
        Base64.Encoder encoder = Base64.getUrlEncoder();
        // Encoding URL
        String eStr = encoder.encodeToString("http://www.javatpoint.com/java-tutorial/".getBytes());
        System.out.println("Encoded URL: "+eStr);
        // Getting decoder
        Base64.Decoder decoder = Base64.getUrlDecoder();
        // Decoding URl
        String dStr = new String(decoder.decode(eStr));
        System.out.println("Decoded URL: "+dStr);
    }
}
```

Output:

```
Encoded URL: aHR0cDovL3d3dy5qYXZhdHBvaW50LmNvbS9qYXZhLXR1dG9yaWFsLw==
Decoded URL: http://www.javatpoint.com/java-tutorial/
```

Java Base64 Example: MIME Encoding and Decoding

```
package Base64Encryption;
import java.util.Base64;
publicclass Base64BasicEncryptionExample {
    publicstaticvoid main(String[] args) {
        // Getting MIME encoder
        Base64.Encoder encoder = Base64.getMimeEncoder();
        String message = "Hello, \nYou are informed regarding your inconsistency of work";
        String eStr = encoder.encodeToString(message.getBytes());
        System.out.println("Encoded MIME message: "+eStr);

        // Getting MIME decoder
        Base64.Decoder decoder = Base64.getMimeDecoder();
        // Decoding MIME encoded message
        String dStr = new String(decoder.decode(eStr));
        System.out.println("Decoded message: "+dStr);
    }
}
```

Output:

```
Encoded MIME message: SGVsbG8sIApZb3UgYXJlIGluZm9ybWVkIHJlZ2FyZGluZyB5b3VyIGluY29uc2lzdGVuY3kgb2Yg
d29yaw==
Decoded message: Hello,
You are informed regarding your inconsistency of work
```

Java Collectors

Collectors is a final class that extends Object class. It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

Java Collectors class provides various methods to deal with elements

Methods	**Description**
public static <T> Collector<T,?,Double> averagingDouble(ToDoubleFunction<? super T> mapper)	It returns a Collector that produces the arithmetic mean of a double-valued function applied to the input elements. If no elements are present, the result is 0.
public static <T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> op)	It returns a Collector which performs a reduction of its input elements under a specified BinaryOperator using the provided identity.
public static <T> Collector<T,?,Optional<T>> reducing(BinaryOperator<T> op)	It returns a Collector which performs a reduction of its input elements under a specified BinaryOperator. The result is described as an Optional<T>.
public static <T,U> Collector<T,?,U> reducing(U identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op)	It returns a Collector which performs a reduction of its input elements under a specified mapping function and BinaryOperator. This is a generalization of reducing(Object, BinaryOperator) which allows a transformation of the elements before reduction.
public static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)	It returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map.
public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? Super T,A,D> downstream)	It returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.
public static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)	It returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. The Map produced by the Collector is created with the supplied factory function.
public static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)	It returns a concurrent Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function.
public static <T,K,A,D> Collector<T,?,ConcurrentMap<K,D>> groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)	It returns a concurrent Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.
public static <T,K,A,D,M extends ConcurrentMap<K,D>> Collector<T,?,M> groupingByConcurrent(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)	It returns a concurrent Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector. The ConcurrentMap produced by the Collector is created with the supplied factory function.
public static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super T> predicate)	It returns a Collector which partitions the input elements according to a Predicate, and organizes them into a Map<Boolean, List<T>>. There are no guarantees on the type, mutability, serializability, or thread-safety of the Map returned.

public static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(Predicate<? super T> predicate, Collector<? Super T,A,D> downstream)	It returns a Collector which partitions the input elements according to a Predicate, reduces the values in each partition according to another Collector, and organizes them into a Map<Boolean, D> whose values are the result of the downstream reduction.
public static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)	It returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements.
public static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)	It returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements.
public static <T,K,U,M extends Map<K,U>> Collector<T,?,M> toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)	It returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements.
public static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)	It returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements.
public static <T,K,U> Collector<T,?,ConcurrentMap<K,U>> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)	It returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements.
public static <T,K,U,M extends ConcurrentMap<K,U>> Collector<T,?,M> toConcurrentMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction, Supplier<M> mapSupplier)	It returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements.

public static <T> Collector<T,?,IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper)	It returns a Collector which applies an int-producing mapping function to each input element, and returns summary statistics for the resulting values.
public static <T> Collector<T,?,LongSummaryStatistics> summarizingLong(ToLongFunction<? super T> mapper)	It returns a Collector which applies an long-producing mapping function to each input element, and returns summary statistics for the resulting values.
public static <T> Collector<T,?,DoubleSummaryStatistics> summarizingDouble(ToDoubleFunction<? super T> mapper)	It returns a Collector which applies an double-producing mapping function to each input element, and returns summary statistics for the resulting values.

```
List<Float> productPriceList =
        productsList.stream()
                    .map(x->x.price)        // fetching price
                    .collect(Collectors.toList());  // collecting as list
```

```
Set<Float>productPriceList =
        productsList.stream()
                    .map(x->x.price)        // fetching price
                    .collect(Collectors.toSet());   // collecting as list
```

forEach

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interfaces.

It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach() method to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

Java 8 forEach() example 1

```
import java.util.ArrayList;
import java.util.List;
public class ForEachExample {
    public static void main(String[] args) {
        List<String> gamesList = new ArrayList<String>();
        gamesList.add("Football");
        gamesList.add("Cricket");
        gamesList.add("Chess");
        gamesList.add("Hocky");
        System.out.println("------------Iterating by passing lambda expression--------------");
        gamesList.forEach(games -> System.out.println(games));

    }
}
```

Output:

```
------------Iterating by passing lambda expression--------------
Football
Cricket
Chess
Hocky
```

Java 8 forEach() example 2

```java
import java.util.ArrayList;
import java.util.List;
public class ForEachExample {
    public static void main(String[] args) {
        List<String> gamesList = new ArrayList<String>();
        gamesList.add("Football");
        gamesList.add("Cricket");
        gamesList.add("Chess");
        gamesList.add("Hocky");
        System.out.println("------------Iterating by passing method reference---------------");
        gamesList.forEach(System.out::println);
    }
}
```

Output:

```
------------Iterating by passing method reference---------------
Football
Cricket
Chess
Hocky
```

Java Stream forEachOrdered() Method

Along with forEach() method, Java provides one more method forEachOrdered(). It is used to iterate elements in the order specified by the stream.

Singnature:

```
void forEachOrdered(Consumer<? super T> action)
```

Java Stream forEachOrdered() Method Example

```
import java.util.ArrayList;
import java.util.List;
public class ForEachOrderedExample {
   public static void main(String[] args) {
      List<String> gamesList = new ArrayList<String>();
      gamesList.add("Football");
      gamesList.add("Cricket");
      gamesList.add("Chess");
      gamesList.add("Hocky");
      System.out.println("------------Iterating by passing lambda expression---------------");
      gamesList.stream().forEachOrdered(games -> System.out.println(games));
      System.out.println("------------Iterating by passing method reference---------------");
      gamesList.stream().forEachOrdered(System.out::println);
   }

}
```

Output:

```
------------Iterating by passing lambda expression---------------
Football
Cricket
Chess
Hocky
------------Iterating by passing method reference---------------
Football
Cricket
Chess
Hocky
```

Java Parallel Array Sorting

Java provides a new additional feature in Array class which is used to sort array elements parallel. New methods has added to java.util.Arrays package that use the JSR 166 Fork/Join parallelism common pool to provide sorting of arrays in parallel. The methods are called parallelSort() and are overloaded for all the primitive data types and Comparable objects.

The following table contains Arrays overloaded sorting methods.

Methods	Description
public static void parallelSort(byte[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(byte[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(char[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(char[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(double[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(double[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(float[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(float[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(int[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(int[] a,int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(long[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(long[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.

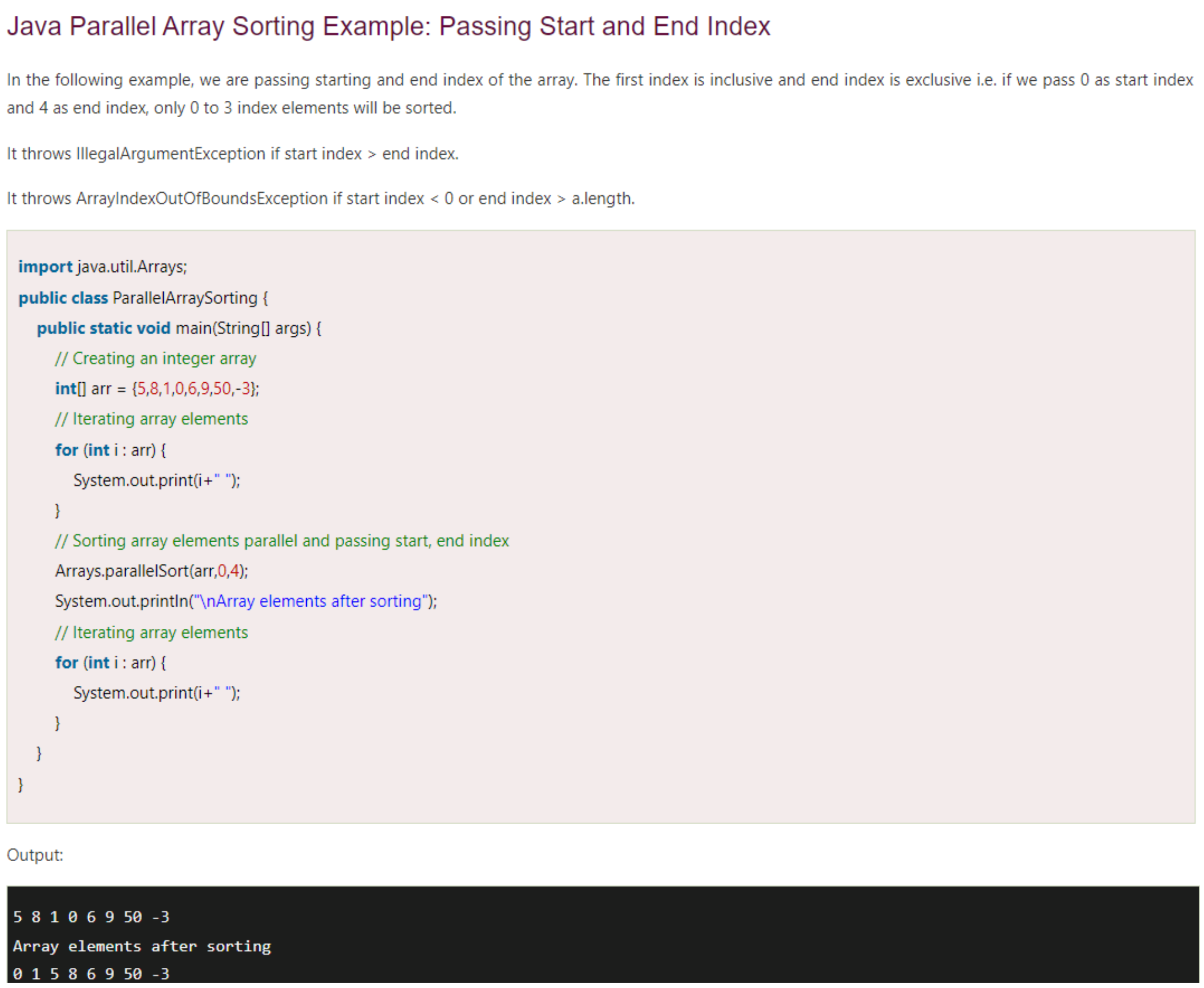
public static void parallelSort(short[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(short[] a,int fromIndex,int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static <T extends Comparable<? super T>> void parallelSort(T[] a)	Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array).
public static <T7gt; void parallelSort(T[] a,Comparator<? super T> cmp)	It sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the array).
public static <T extends Comparable<? super T>> void parallelSort(T[] a,int fromIndex, int toIndex)	It sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be sorted is empty.) All elements in this range must implement the Comparable interface. Furthermore, all elements in this range must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array).
public static <T> void parallelSort(T[] a, int fromIndex, int toIndex, Comparator<? super T> cmp)	It sorts the specified range of the specified array of objects according to the order induced by the specified comparator. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive. (If fromIndex==toIndex, the range to be sorted is empty.) All elements in the range must be mutually comparable by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the range).

Java Parallel Array Sorting Example

```java
import java.util.Arrays;
public class ParallelArraySorting {
    public static void main(String[] args) {
        // Creating an integer array
        int[] arr = {5,8,1,0,6,9};
        // Iterating array elements
        for (int i : arr) {
            System.out.print(i+" ");
        }
        // Sorting array elements parallel
        Arrays.parallelSort(arr);
        System.out.println("\nArray elements after sorting");
        // Iterating array elements
        for (int i : arr) {
            System.out.print(i+" ");
        }
    }
}
```

Output:

```
5 8 1 0 6 9
Array elements after sorting
0 1 5 6 8 9
```



Java Parallel Array Sorting Example: Passing Start and End Index

In the following example, we are passing starting and end index of the array. The first index is inclusive and end index is exclusive i.e. if we pass 0 as start index and 4 as end index, only 0 to 3 index elements will be sorted.

It throws IllegalArgumentException if start index > end index.

It throws ArrayIndexOutOfBoundsException if start index < 0 or end index > a.length.

```java
import java.util.Arrays;
public class ParallelArraySorting {
  public static void main(String[] args) {
    // Creating an integer array
    int[] arr = {5,8,1,0,6,9,50,-3};
    // Iterating array elements
    for (int i : arr) {
      System.out.print(i+" ");
    }
    // Sorting array elements parallel and passing start, end index
    Arrays.parallelSort(arr,0,4);
    System.out.println("\nArray elements after sorting");
    // Iterating array elements
    for (int i : arr) {
      System.out.print(i+" ");
    }
  }
}
```

Output:

```
5 8 1 0 6 9 50 -3
Array elements after sorting
0 1 5 8 6 9 50 -3
```

Java 8 I/O Enhancements

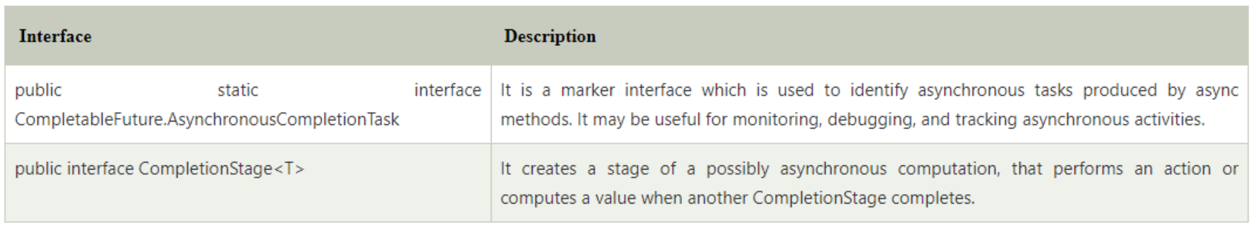
In Java 8, there are several improvements to the java.nio.charset.Charset and extended charset implementations. It includes the following:

- A New SelectorProvider which may improve performance or scalability for server. The /dev/poll SelectorProvider continues to be the default. To use the Solaris event port mechanism, run with the system property java.nio.channels.spi.Selector set to the value sun.nio.ch.EventPortSelectorProvider.
- The size of <JDK_HOME>/jre/lib/charsets.jar file is decreased.
- Performance has been improvement for the java.lang.String(byte[], ∗) constructor and the java.lang.String.getBytes() method.

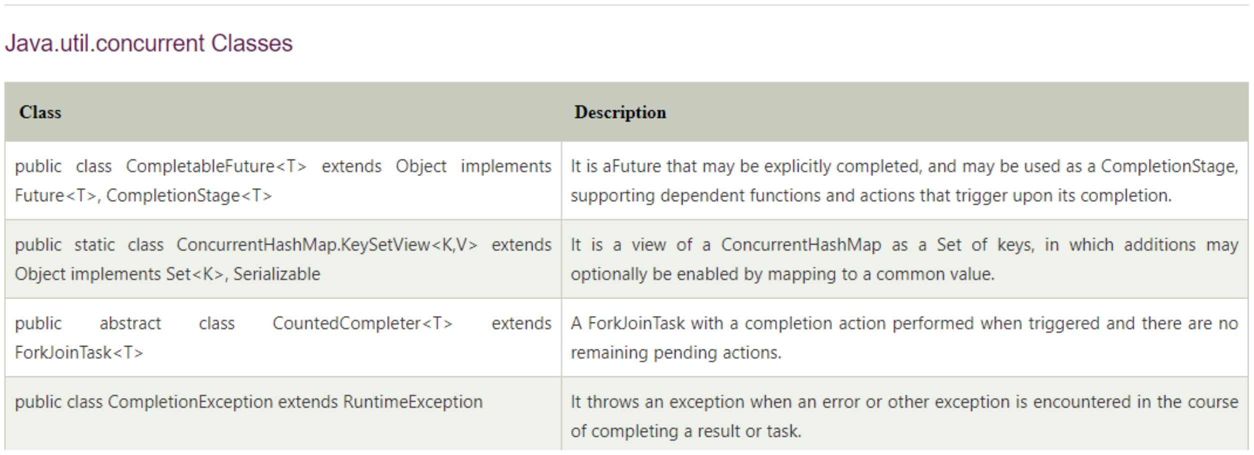
Java 8 Concurrency Enhancements

The java.util.concurrent package added two new interfaces and four new classes.

Java.util.concurrent Interfaces



Interface	Description
public static interface CompletableFuture.AsynchronousCompletionTask	It is a marker interface which is used to identify asynchronous tasks produced by async methods. It may be useful for monitoring, debugging, and tracking asynchronous activities.
public interface CompletionStage<T>	It creates a stage of a possibly asynchronous computation, that performs an action or computes a value when another CompletionStage completes.



Java.util.concurrent Classes

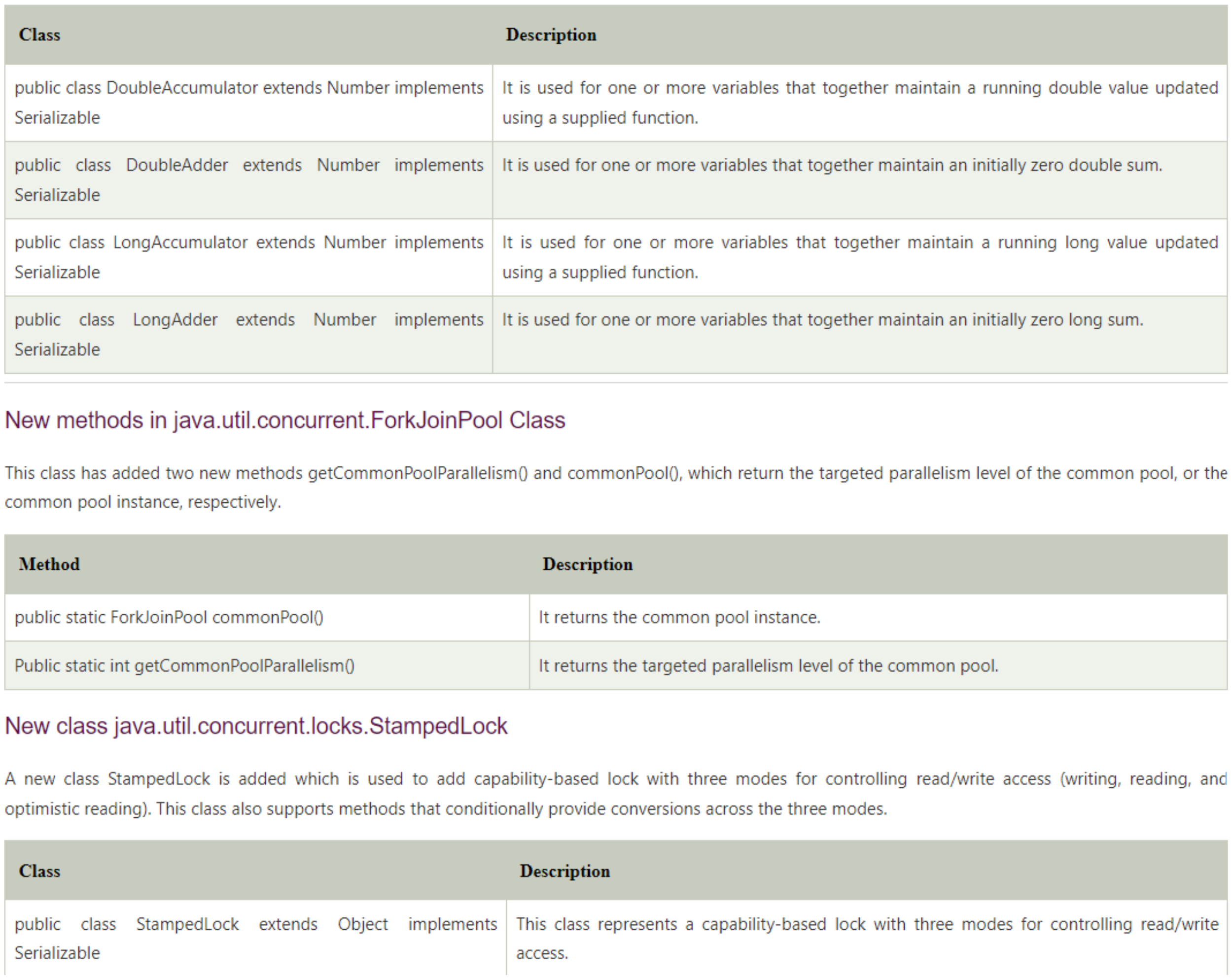
Class	Description
public class CompletableFuture<T> extends Object implements Future<T>, CompletionStage<T>	It is aFuture that may be explicitly completed, and may be used as a CompletionStage, supporting dependent functions and actions that trigger upon its completion.
public static class ConcurrentHashMap.KeySetView<K,V> extends Object implements Set<K>, Serializable	It is a view of a ConcurrentHashMap as a Set of keys, in which additions may optionally be enabled by mapping to a common value.
public abstract class CountedCompleter<T> extends ForkJoinTask<T>	A ForkJoinTask with a completion action performed when triggered and there are no remaining pending actions.
public class CompletionException extends RuntimeException	It throws an exception when an error or other exception is encountered in the course of completing a result or task.

New Methods in java.util.concurrent.ConcurrentHashMap class

ConcurrentHashMap class introduces several new methods in its latest release. It includes various forEach methods (forEach, forEachKey, forEachValue, and forEachEntry), search methods (search, searchKeys, searchValues, and searchEntries) and a large number of reduction methods (reduce, reduceToDouble, reduceToLong etc.). Other miscellaneous methods (mappingCount and newKeySet) have been added as well.

New classes in java.util.concurrent.atomic

Latest release introduces scalable, updatable, variable support through a small set of new classes DoubleAccumulator, DoubleAdder, LongAccumulator andLongAdder. It internally employ contention-reduction techniques that provide huge throughput improvements as compared to Atomic variables.



Class	Description
public class DoubleAccumulator extends Number implements Serializable	It is used for one or more variables that together maintain a running double value updated using a supplied function.
public class DoubleAdder extends Number implements Serializable	It is used for one or more variables that together maintain an initially zero double sum.
public class LongAccumulator extends Number implements Serializable	It is used for one or more variables that together maintain a running long value updated using a supplied function.
public class LongAdder extends Number implements Serializable	It is used for one or more variables that together maintain an initially zero long sum.

New methods in java.util.concurrent.ForkJoinPool Class

This class has added two new methods getCommonPoolParallelism() and commonPool(), which return the targeted parallelism level of the common pool, or the common pool instance, respectively.

Method	Description
public static ForkJoinPool commonPool()	It returns the common pool instance.
Public static int getCommonPoolParallelism()	It returns the targeted parallelism level of the common pool.

New class java.util.concurrent.locks.StampedLock

A new class StampedLock is added which is used to add capability-based lock with three modes for controlling read/write access (writing, reading, and optimistic reading). This class also supports methods that conditionally provide conversions across the three modes.

Class	Description
public class StampedLock extends Object implements Serializable	This class represents a capability-based lock with three modes for controlling read/write access.

Collection API improvements

We have already seen forEach() method and Stream API for collections. Some new methods added in Collection API are:

- Iterator default method forEachRemaining(Consumer action) to perform the given action for each remaining element until all elements have been processed or the action throws an exception.
- Collection default method removeIf(Predicate filter) to remove all of the elements of this collection that satisfy the given predicate.
- Collection spliterator() method returning Spliterator instance that can be used to traverse elements sequentially or parallel.
- Map replaceAll(), compute(), merge() methods.
- Performance Improvement for HashMap class with Key Collisions

Java 9 Module System

Java Module System is a major change in Java 9 version. Java added this feature to collect Java packages and code into a single unit called *module*.

In earlier versions of Java, there was no concept of module to create modular Java applications, that why size of application increased and difficult to move around. Even JDK itself was too heavy in size, in Java 8, **rt.jar** file size is around 64MB.

To deal with situation, **Java 9 restructured JDK into set of modules** so that we can use only required module for our project. Apart from JDK, Java also allows us to create our own modules so that we can develop module-based application.

The module system includes various tools and options that are given below.

- Includes various options to the Java tools **javac, jlink and java** where we can specify module paths that locates to the location of module.
- Modular JAR file is introduced. This JAR contains module-info.class file in its root folder.
- JMOD format is introduced, which is a packaging format similar to JAR except it can include native code and configuration files.
- The JDK and JRE both are reconstructed to accommodate modules. It improves performance, security and maintainability.
- Java defines a new URI scheme for naming modules, classes and resources.

Java 9 Module

Module is a collection of Java programs or software's. To describe a module, a Java file **module-info.java** is required. This file also known as module descriptor and defines the following

- Module name
- What does it export
- What does it require

Module Name

It is a name of module and should follow the reverse-domain-pattern. Like we name packages, e.g., com.javatpoint.

How to create Java module

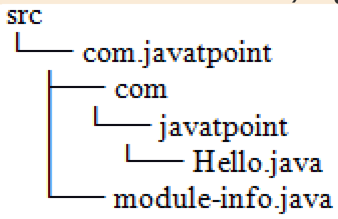
Creating Java module required the following steps.

- Create a directory structure
- Create a module declarator
- Java source code

Create a Directory Structure

To create module, it is recommended to follow given directory structure, it is same as reverse-domain-pattern, we do to create packages / project-structure in Java.

Note: The name of the directory containing a module's sources should be equal to the name of the module, e.g. com.javatpoint.



```
src
└── com.javatpoint
    ├── com
    │   └── javatpoint
    │       └── Hello.java
    └── module-info.java
```

Create a file **module-info.java**, inside this file, declare a module by using **module** identifier and provide module name same as the directory name that contains it. In our case, our directory name is com.javatpoint.

```
module com.javatpoint {
}
```

Leave module body empty, if it does not have any module dependency. Save this file inside **src/com.javatpoint** with **module-info.java** name.

Java Source Code

Now, create a Java file to compile and execute module. In our example, we have a **Hello.java** file that contains the following code.

```
class Hello {

    public static void main(String[] args) {

    System.out.println("Hello from the Java module");

        }

  }
```

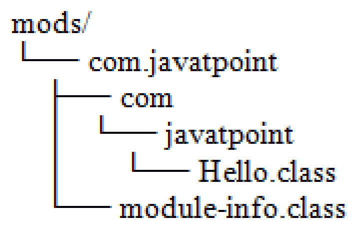
Save this file inside **src/com.javatpoint/com/javatpoint/** with **Hello.java** name.

Compile Java Module

To compile the module, use the following command.

```
1. javac -d mods --module-source-path src/ --module com.javatpoint
```

After compiling, it will create a new directory that contains the following structure.



```
mods/
└── com.javatpoint
    ├── com
    │   └── javatpoint
    │       └── Hello.class
    └── module-info.class
```

Now, we have a compiled module that can be just run.

Run Module

To run the compiled module, use the following command.

```
1. java --module-path mods/ --module com.javatpoint/com.javatpoint.Hello
```

Output:

```
Hello from the Java module
```

Well, we have successfully created, compiled and executed Java module.

Look inside compiled Module Descriptor

To see the compiled module descriptor, use the following command.

```
1. javap mods/com.javatpoint/module-info.class
```

This command will show the following code to the console.

```
Compiled from "module-info.java"
module com.javatpoint {
  requires java.base;
}
```

See, we created an empty module but it contains a **java.base** module. Why? Because all Java modules are linked to java.base module and it is default module.

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();
    }
}
```

Output:

```
PrivateInterface.java:6: error: modifier private not allowed here
```

Note: To implement private interface methods, compile source code using Java 9 or higher versions only.

Now, lets execute the above code by using Java 9. See the output, it executes fine.

Output:

```
Hello... I'm private method
```

Java 9 Try With Resource Enhancement

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 its 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.

```java
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.**

Output:

```
File written
```

Java 9 Anonymous Inner Classes Improvement

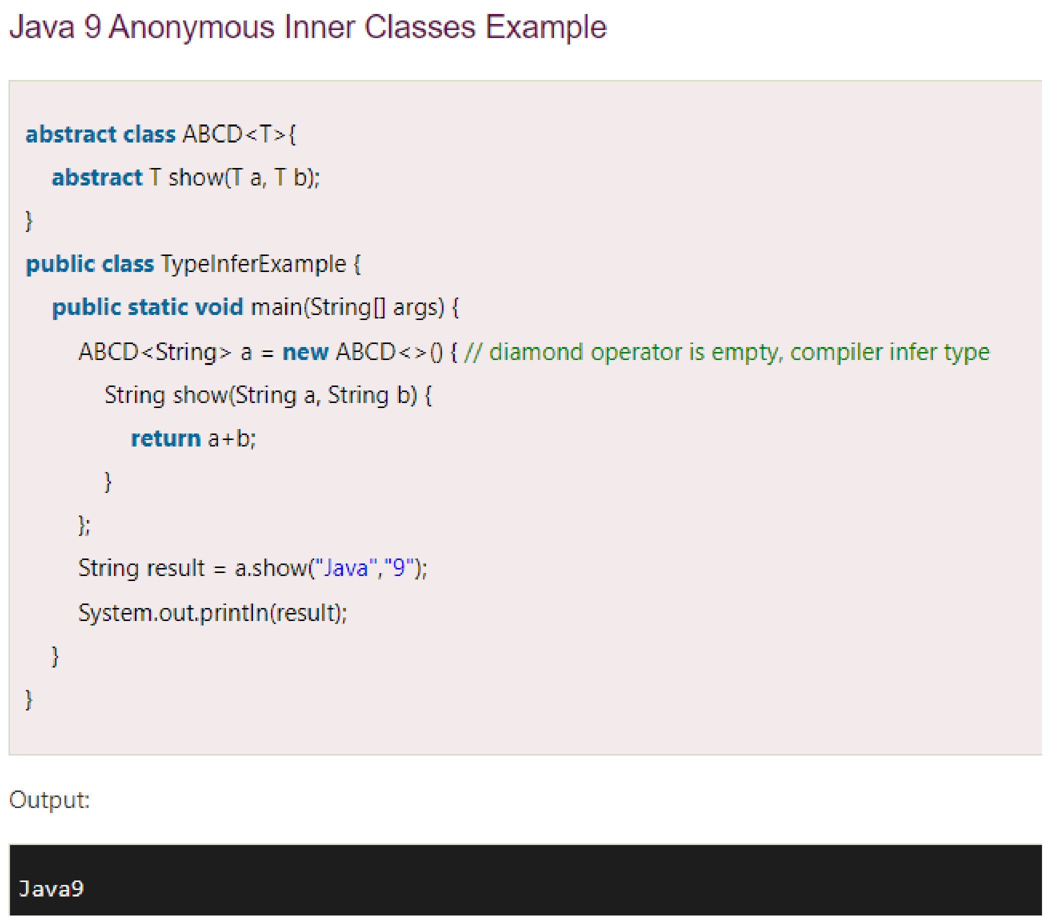
Java 9 introduced a new feature that allows us to use diamond operator with anonymous classes. Using the diamond with anonymous classes was not allowed in Java 7.

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.

Note: This feature is included to Java 9, to add type inference in anonymous inner classes.

Let's see an example, in which we are using diamond operator with inner class without specifying type.



Java 9 Anonymous Inner Classes Example

```java
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 specify type in diamond operator explicitly and compiler does not produce any error message. See, the following example, type is specified explicitly.

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

And we get the same result.

Output:

```
Java9
```

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.

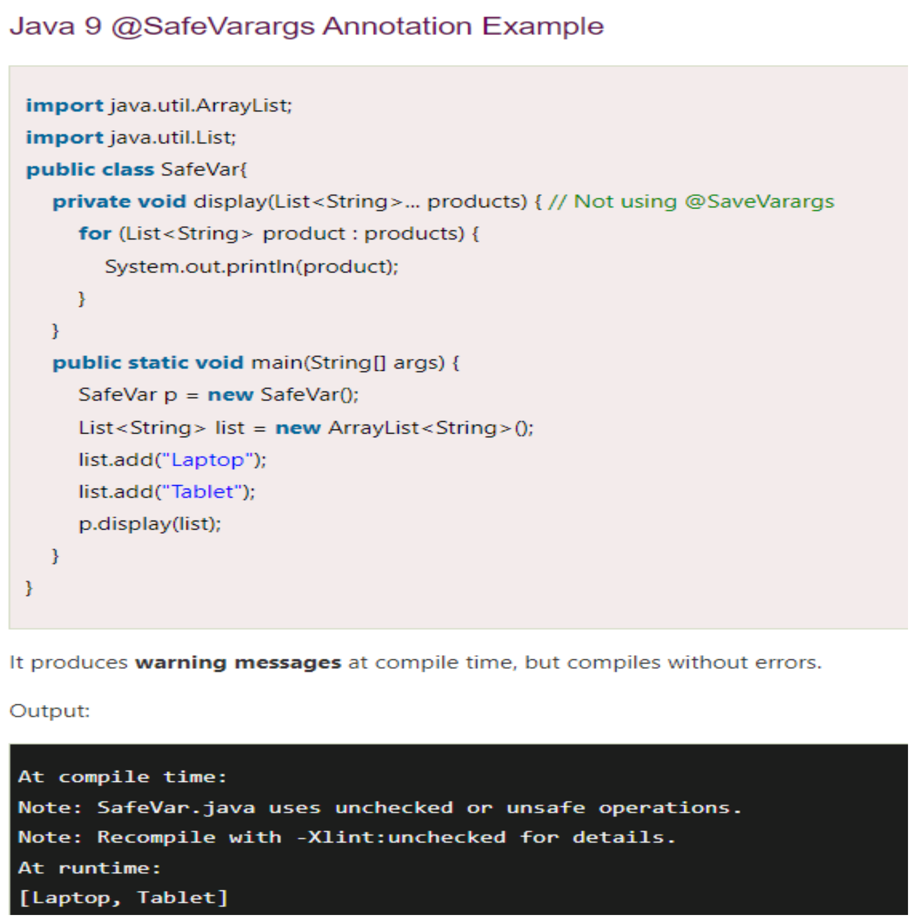
It was included in Java7 and can only be applied on

- 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.

Let's see some examples, in first example, we are not using @SafeVarargs annotation and compiling code



Java 9 @SafeVarargs Annotation Example

```java
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);
    }
}
```

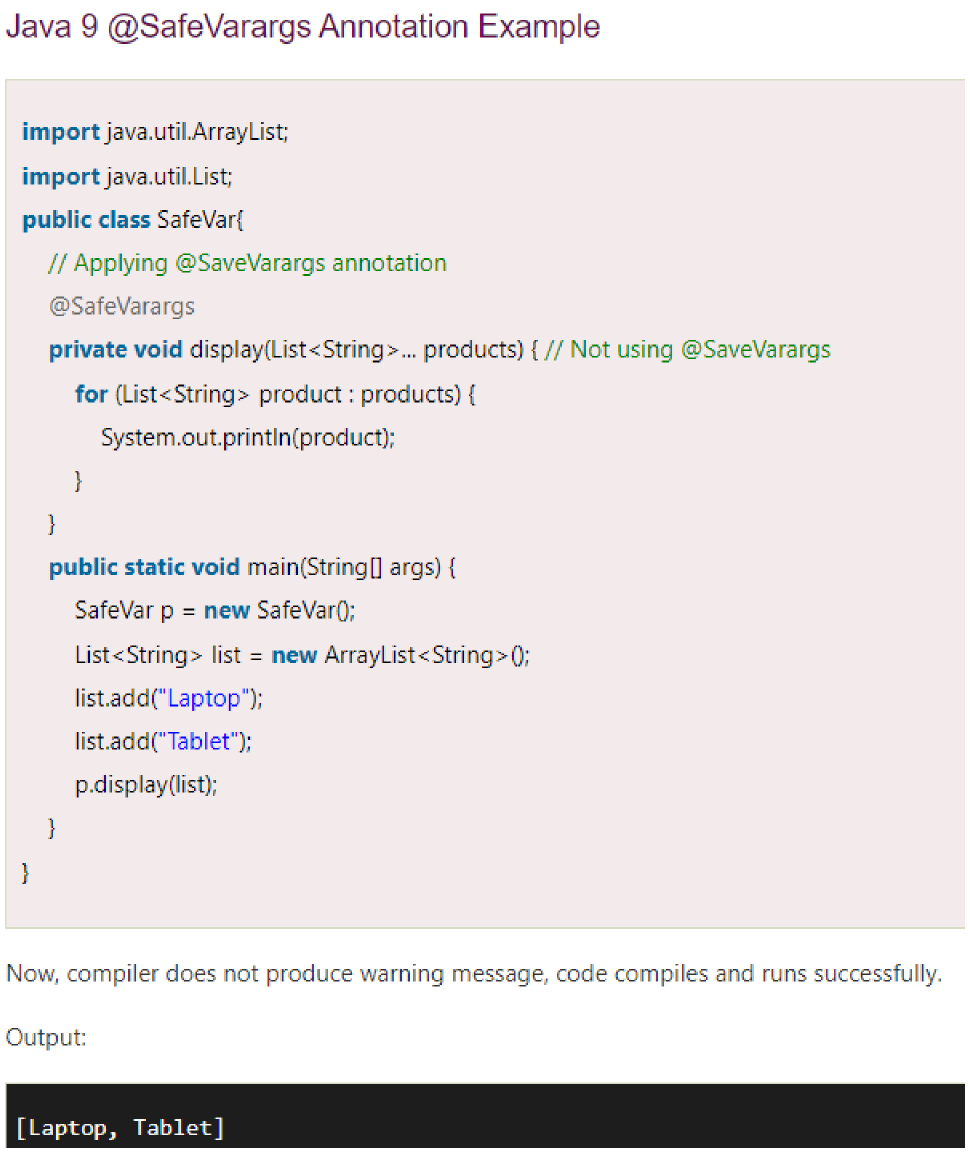
It produces **warning messages** at compile time, but compiles without errors.

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]
```

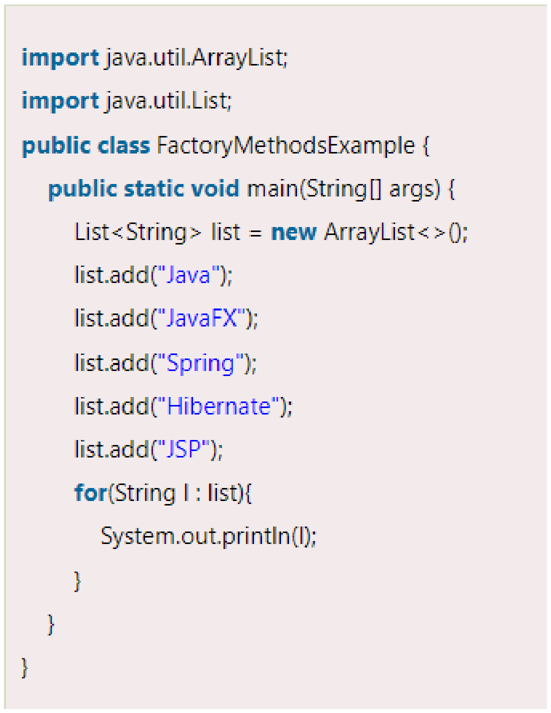
Note: To apply @SaveVarargs annotation on private instance methods, compile code using Java 9 or higher versions only.

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);
        }
    }
}
```

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.

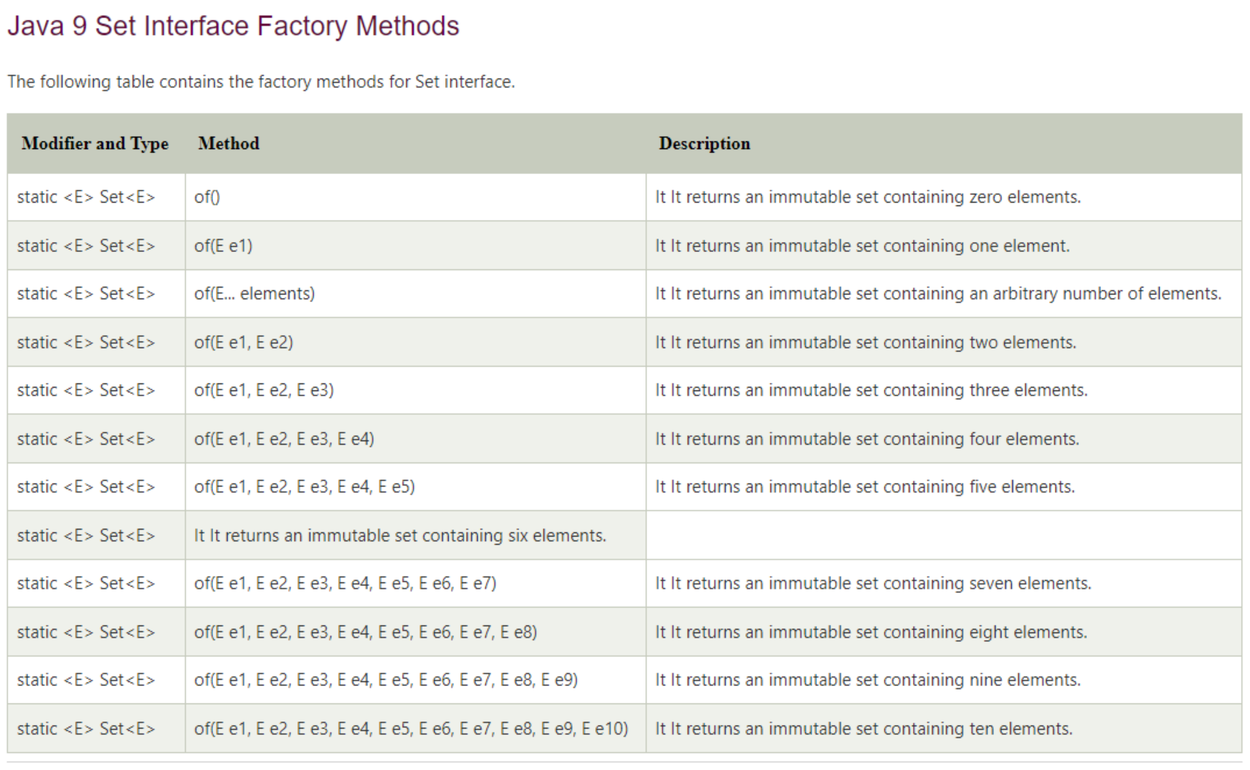
It is unmodifiable, so adding new element will throw **java.lang.UnsupportedOperationException**

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

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 characteristics.

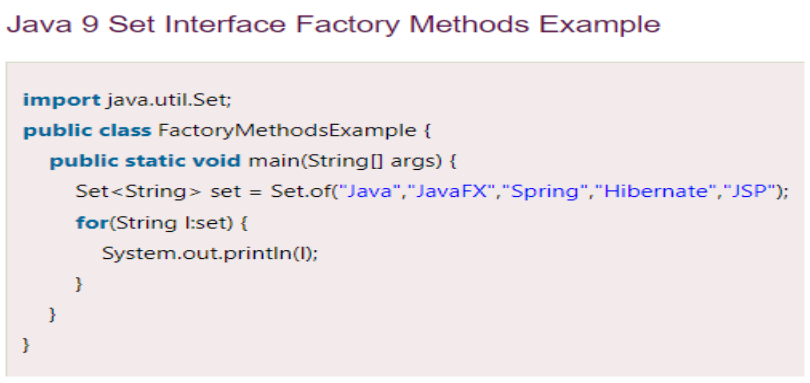
- It is immutable
- No null elements
- It is serializable if all elements are serializable.
- No duplicate elements.
- The iteration order of set elements is unspecified and is subject to change.



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

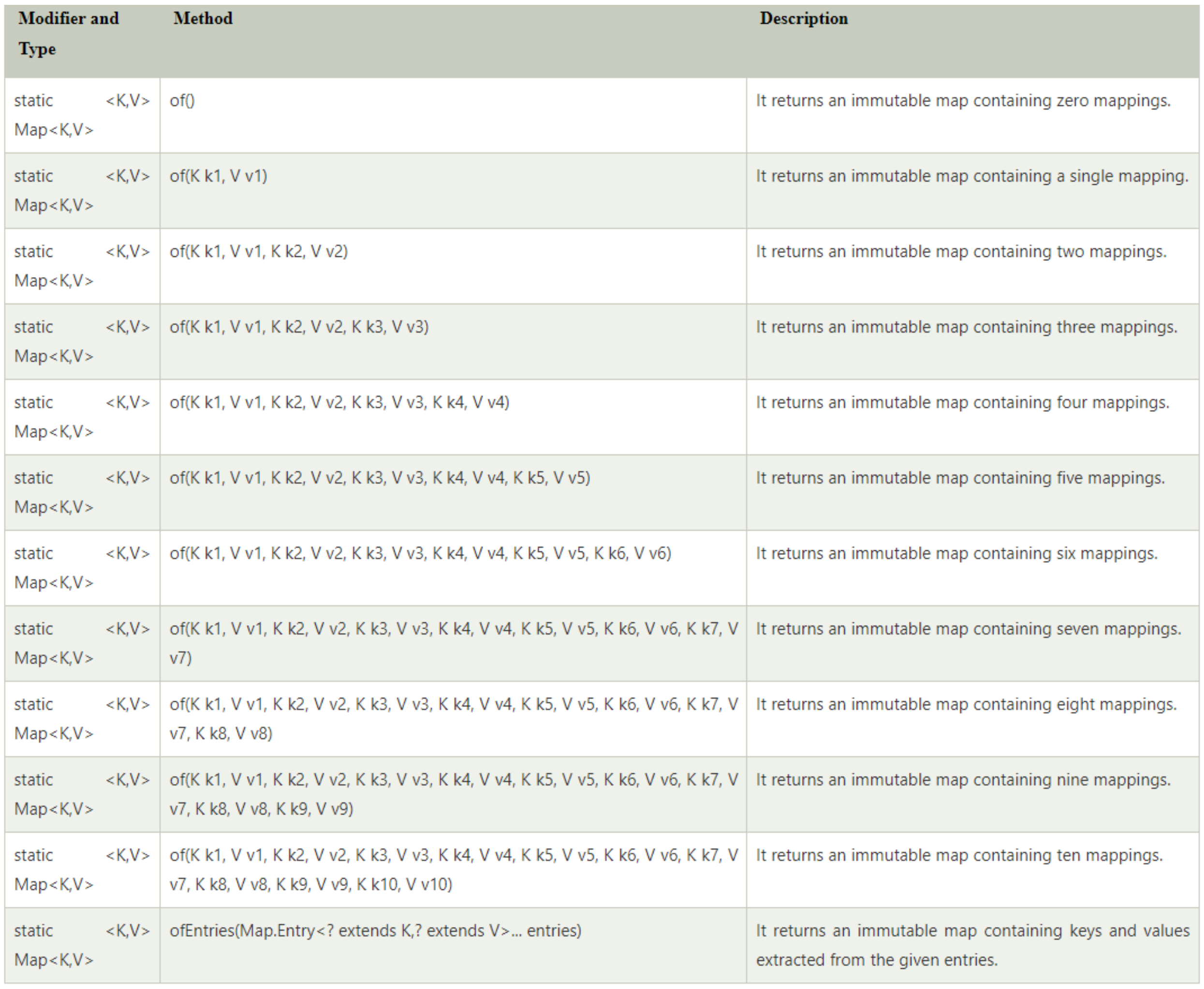
```java
import java.util.Set;
public class FactoryMethodsExample {
    public static void main(String[] args) {
        Set<String> set = Set.of("Java","JavaFX","Spring","Hibernate","JSP");
        for(String l:set) {
            System.out.println(l);
        }
    }
}
```

Java 9 Map Interface Factory Methods

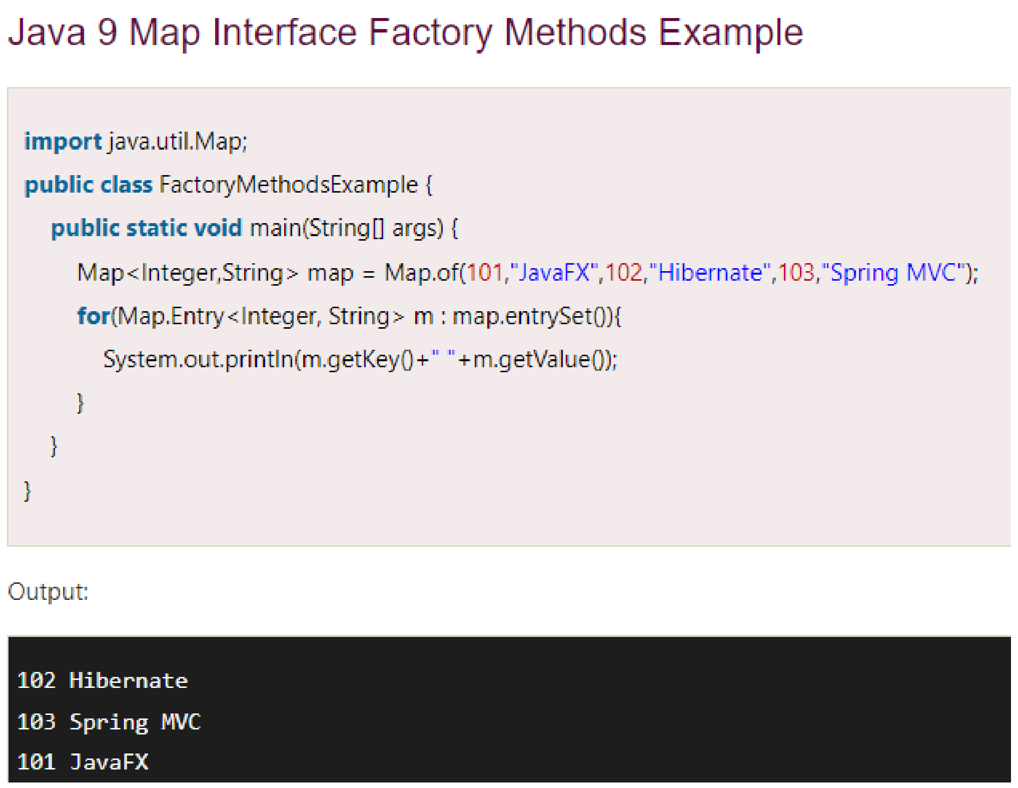
In Java 9, Map includes Map.of() and Map.ofEntries() static factory methods that provide a convenient way to create immutable maps.

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.



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

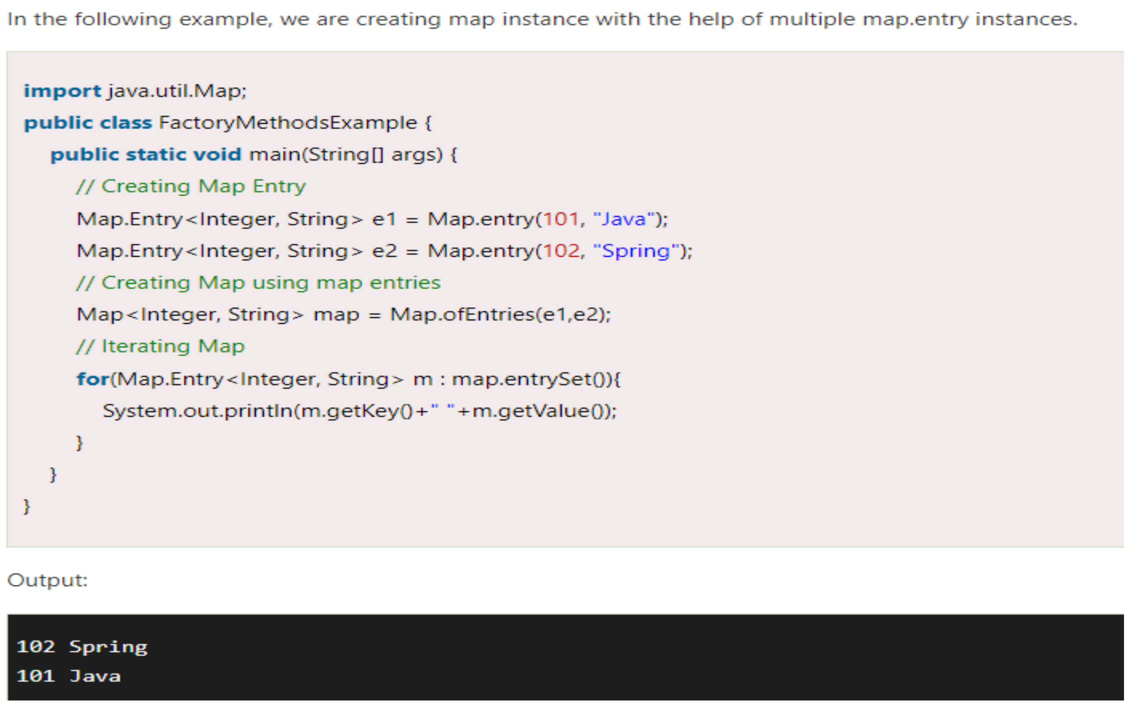
```java
import java.util.Map;
public class FactoryMethodsExample {
    public static void main(String[] args) {
        Map<Integer,String> map = Map.of(101,"JavaFX",102,"Hibernate",103,"Spring MVC");
        for(Map.Entry<Integer, String> m : map.entrySet()){
            System.out.println(m.getKey()+" "+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.

```java
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
```

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.**

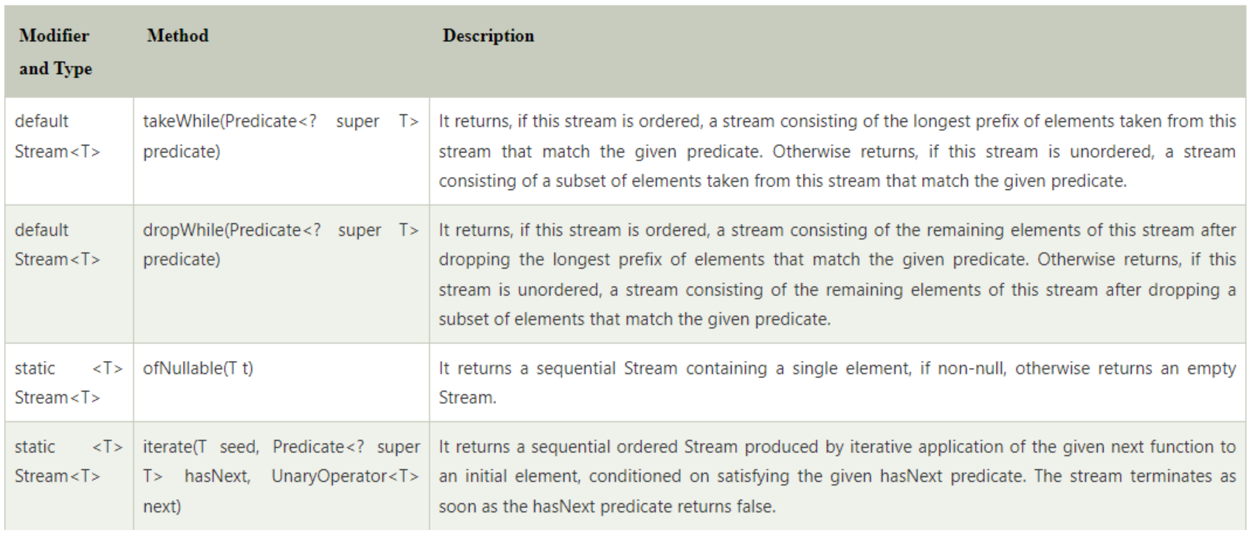
```java
import java.util.List;
public class FactoryMethodsExample {
    public static void main(String[] args) {
        List<String> list = List.of("Java","JavaFX","Spring","Hibernate","JSP");
        for(String l:list) {
            System.out.println(l);
        }
    }
}
```

Output:

```
Java
JavaFX
Spring
Hibernate
JSP
```

Java 9 Stream API Improvement

In Java 9, Stream API has improved and new methods are added to the Stream interface. These methods are tabled below.



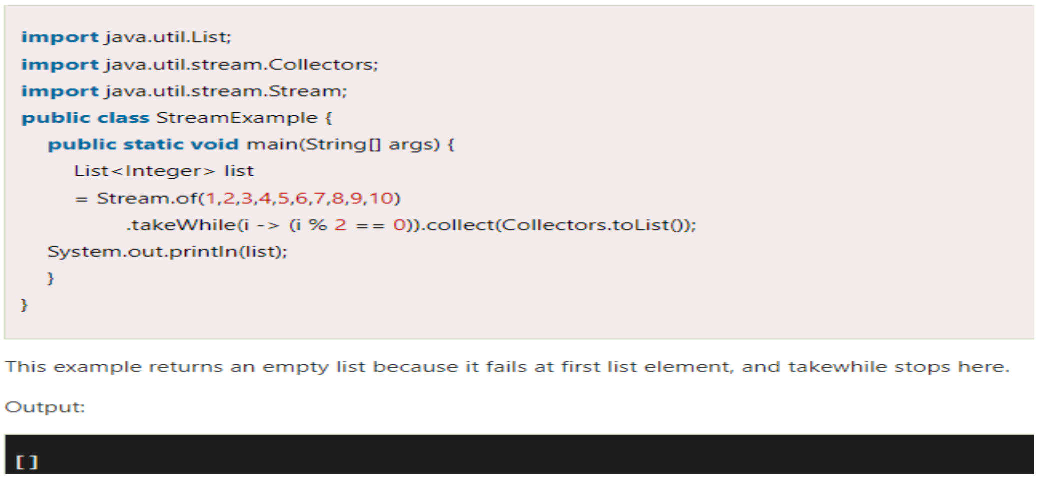
Modifier and Type	Method	Description
default Stream<T>	takeWhile(Predicate<? super T> predicate)	It returns, if this stream is ordered, a stream consisting of the longest prefix of elements taken from this stream that match the given predicate. Otherwise returns, if this stream is unordered, a stream consisting of a subset of elements taken from this stream that match the given predicate.
default Stream<T>	dropWhile(Predicate<? super T> predicate)	It returns, if this stream is ordered, a stream consisting of the remaining elements of this stream after dropping the longest prefix of elements that match the given predicate. Otherwise returns, if this stream is unordered, a stream consisting of the remaining elements of this stream after dropping a subset of elements that match the given predicate.
static <T> Stream<T>	ofNullable(T t)	It returns a sequential Stream containing a single element, if non-null, otherwise returns an empty Stream.
static <T> Stream<T>	iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)	It returns a sequential ordered Stream produced by iterative application of the given next function to an initial element, conditioned on satisfying the given hasNext predicate. The stream terminates as soon as the hasNext predicate returns false.

Java Stream takeWhile() Method

Stream takeWhile method takes each element that matches its predicate. It stops when it get unmatched element. It returns a subset of elements that contains all matched elements, other part of stream is discarded.

Java Stream takeWhile() Method Example 1

In this example, we have a list of integers and picks up even values by using takewhile method.



```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class StreamExample {
    public static void main(String[] args) {
        List<Integer> list
        = Stream.of(1,2,3,4,5,6,7,8,9,10)
                .takeWhile(i -> (i % 2 == 0)).collect(Collectors.toList());
    System.out.println(list);
    }
}
```

This example returns an empty list because it fails at first list element, and takewhile stops here.

Output:

```
[]
```

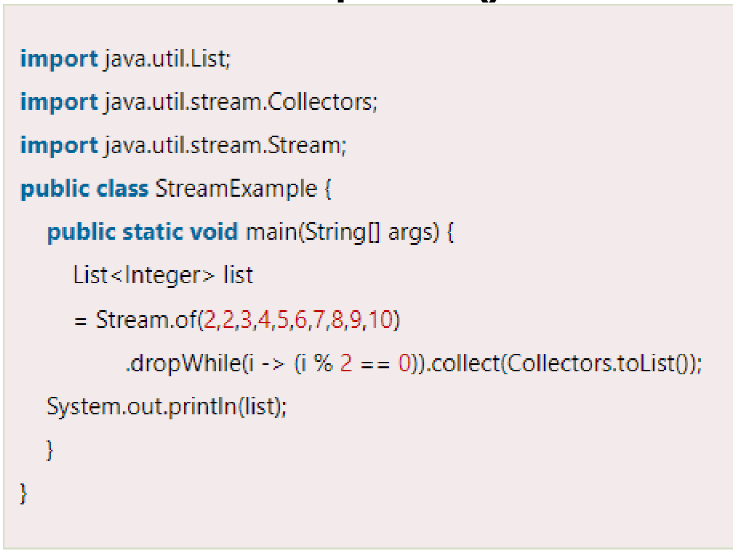
Java Stream dropWhile() Method

Stream dropWhile method returns result on the basis of order of stream elements.

Ordered stream: It returns a stream that contains elements after dropping the elements that match the given predicate.

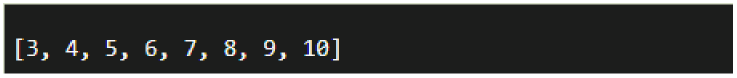
Unordered stream: It returns a stream that contains remaining elements of this stream after dropping a subset of elements that match the given predicate.

Java Stream dropWhile() Method Example



```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class StreamExample {
    public static void main(String[] args) {
        List<Integer> list
        = Stream.of(2,2,3,4,5,6,7,8,9,10)
                .dropWhile(i -> (i % 2 == 0)).collect(Collectors.toList());
    System.out.println(list);
    }
}
```

Output:

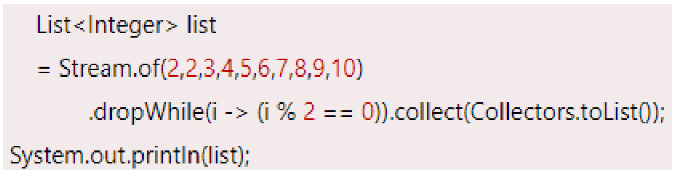


```
[3, 4, 5, 6, 7, 8, 9, 10]
```

Java 9 Stream ofNullable Method

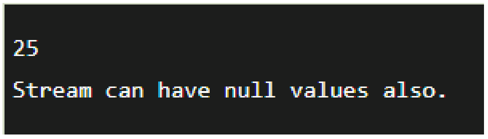
Stream ofNullable method returns a sequential stream that contains a single element, if non-null. Otherwise, it returns an empty stream.It helps to handle null stream and NullPointerException.

Java 9 Stream ofNullable Method Example 1



```java
    List<Integer> list
    = Stream.of(2,2,3,4,5,6,7,8,9,10)
            .dropWhile(i -> (i % 2 == 0)).collect(Collectors.toList());
System.out.println(list);
```

Output:



```
25
Stream can have null values also.
```

Java 9 Stream ofNullable Method Example 2

```
import java.util.stream.Stream;

public class StreamExample {
    public static void main(String[] args) {
        Stream<Integer> val
        = Stream.ofNullable(null);
    val.forEach(System.out::println);
    }
}
```

This program will not produce any output.

Java Stream Iterate Method

A new overloaded method **iterate** is added to the Java 9 stream interface. This method allows us to iterate stream elements till the specified condition.

It takes three arguments, seed, hasNext and next.

Java Stream Iterate Method Example

```
import java.util.stream.Stream;

public class StreamExample {
    public static void main(String[] args) {
        Stream.iterate(1, i -> i <= 10, i -> i*3)
        .forEach(System.out::println);
    }
}
```

Output:

```
1
3
9
```

Local Variable Type Inference

Local Variable Type Inference is one of the most evident change to language available from Java 10 onwards. It allows to define a variable using var and without specifying the type of it. The compiler infers the type of the variable using the value provided. This type inference is restricted to local variables.

Old way of declaring local variable.

```
String name = "Welcome to tutorialspoint.com";
```

New Way of declaring local variable.

```
var name = "Welcome to tutorialspoint.com";
```

Now compiler infers the type of name variable as String by inspecting the value provided.

Noteworthy points

- No type inference in case of member variable, method parameters, return values.
- Local variable should be initialized at time of declaration otherwise compiler will not be infer and will throw error.
- Local variable inference is available inside initialization block of loop statements.
- No runtime overhead. As compiler infers the type based on value provided, there is no performance loss.
- No dynamic type change. Once type of local variable is inferred it cannot be changed.
- Complex boilerplate code can be reduced using local variable type inference.

```
Map<Integer, String> mapNames = new HashMap<>();

var mapNames1 = new HashMap<Integer, String>();
```

Example

Following Program shows the use of Local Variable Type Inference in JAVA 10.

```
import java.util.List;

public class Tester {
  public static void main(String[] args) {
    var names = List.of("Julie", "Robert", "Chris", "Joseph");
    for (var name : names) {
      System.out.println(name);
    }
    System.out.println("");
    for (var i = 0; i < names.size(); i++) {
      System.out.println(names.get(i));
    }
  }
}
```

Output

It will print the following output.

```
Julie
Robert
Chris
Joseph

Julie
Robert
Chris
Joseph
```

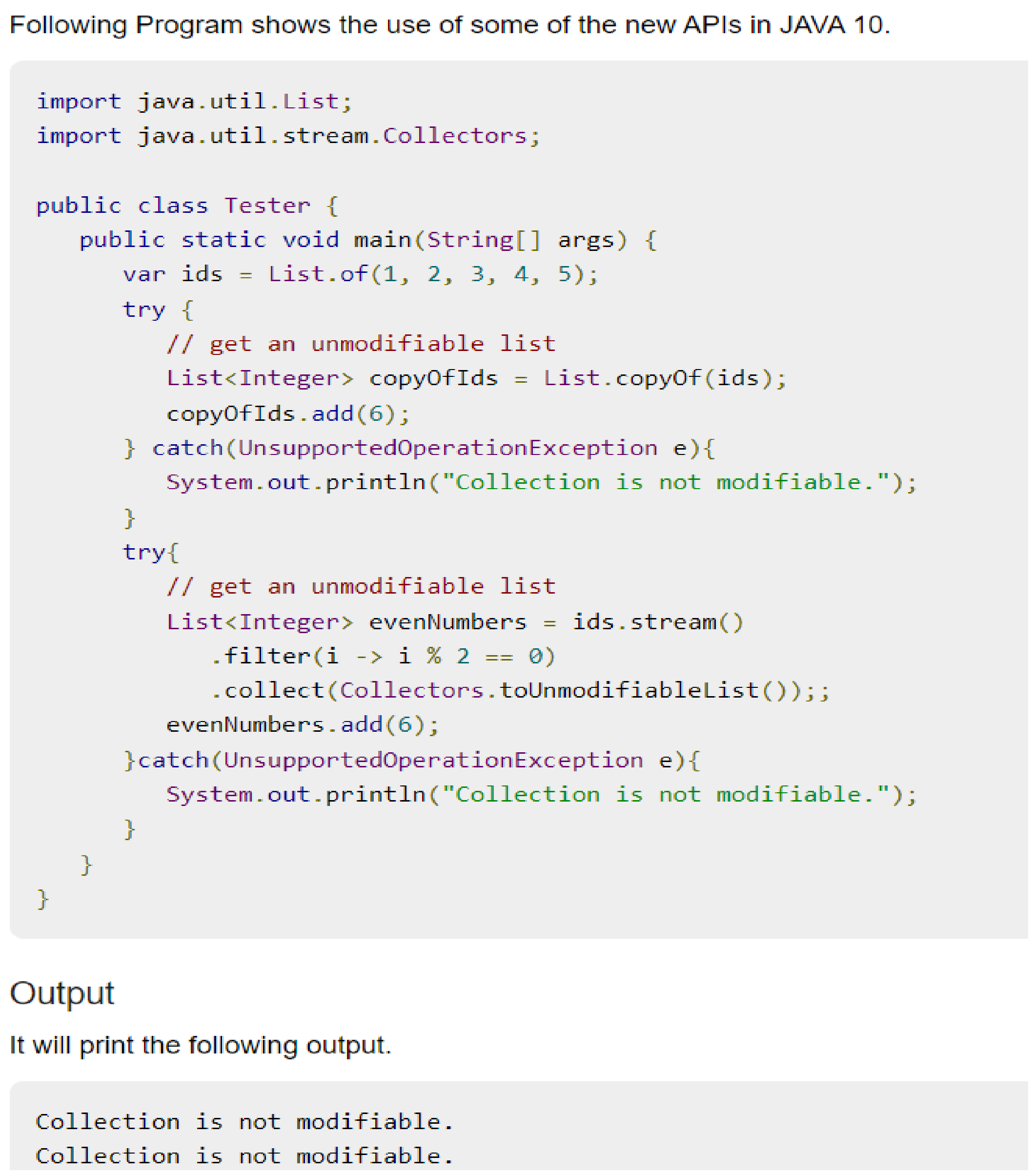
New APIs & Options

APIs to create Unmodifiable Collections

A new method **copyOf()** is available in List, Set and Map interfaces which can create new collection instances from existing one. Collector class has new methods **toUnmodifiableList(), toUnmodifiableSet(), and toUnmodifiableMap()** to get elements of a stream into an unmodifiable collection.

Hashed Password

The plain text passwords available in the jmxremote.password file are now being over-written with their SHA3-512 hash by the JMX agent.



Following Program shows the use of some of the new APIs in JAVA 10.

```java
import java.util.List;
import java.util.stream.Collectors;

public class Tester {
   public static void main(String[] args) {
      var ids = List.of(1, 2, 3, 4, 5);
      try {
         // get an unmodifiable list
         List<Integer> copyOfIds = List.copyOf(ids);
         copyOfIds.add(6);
      } catch(UnsupportedOperationException e){
         System.out.println("Collection is not modifiable.");
      }
      try{
         // get an unmodifiable list
         List<Integer> evenNumbers = ids.stream()
            .filter(i -> i % 2 == 0)
            .collect(Collectors.toUnmodifiableList());;
         evenNumbers.add(6);
      }catch(UnsupportedOperationException e){
         System.out.println("Collection is not modifiable.");
      }
   }
}
```

Output

It will print the following output.

```
Collection is not modifiable.
Collection is not modifiable.
```

Optional.orElseThrow() Method

A new method **orElseThrow()** is available in java.util.Optional class which is now a preferred alternative for **get()** method.

Java.util.Optional, java.util.OptionalDouble, java.util.OptionalInt and java.util.OptionalLong each got a new method orElseThrow() which doesn't take any argument and throws NoSuchElementException if no value is present.

```
@Test
public void whenListContainsInteger_OrElseThrowReturnsInteger() {
    Integer firstEven = someIntList.stream()
      .filter(i -> i % 2 == 0)
      .findFirst()
      .orElseThrow();
    is(firstEven).equals(Integer.valueOf(2));
}
```

It's synonymous with and is now the preferred alternative to the existing *get()*method.

Running Java File with single command

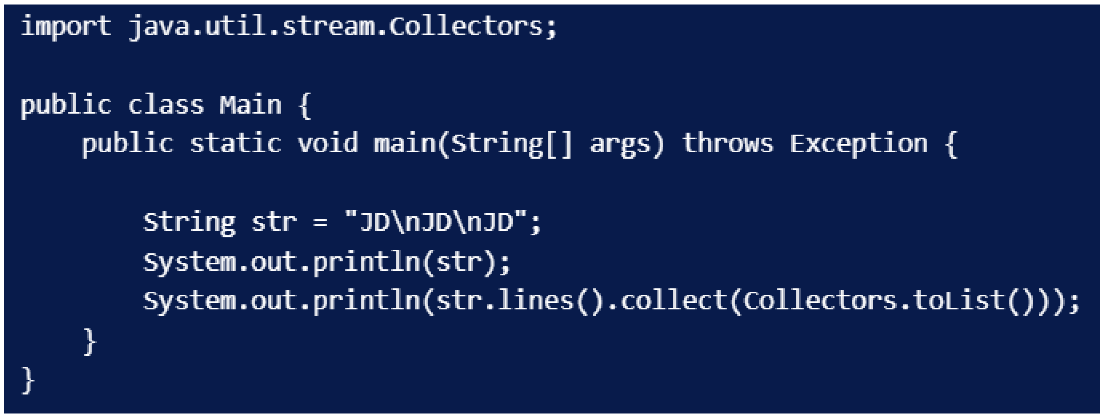
One major change is that you don’t need to compile the java source file with `javac` tool first. You can directly run the file with **java** command and it implicitly compiles.

Java String Methods

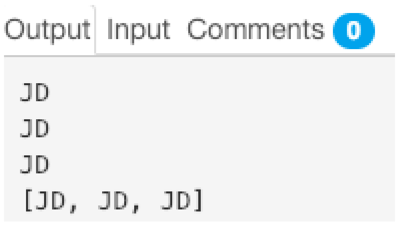
isBlank() - This instance method returns a boolean value. Empty Strings and Strings with only white spaces are treated as blank.

```
import java.util.*;

public class Main {
    public static void main(String[] args) throws Exception {
        // Your code here!

        System.out.println(" ".isBlank()); //true

        String s = "Anupam";
        System.out.println(s.isBlank()); //false
        String s1 = "";
        System.out.println(s1.isBlank()); //true
    }
}
```

lines() This method returns a stream of strings, which is a collection of all substrings split by lines.



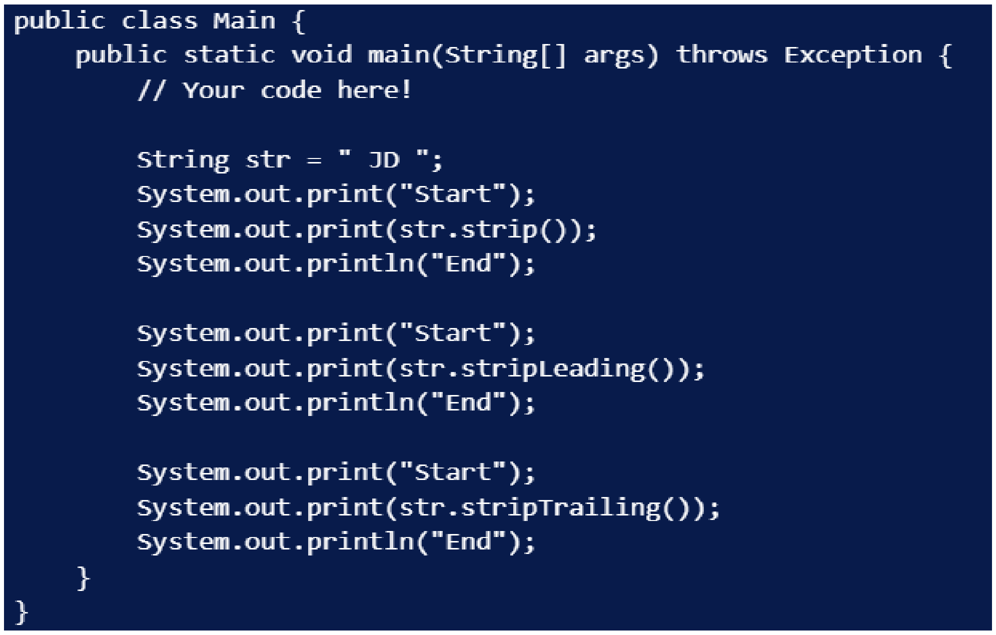
```
import java.util.stream.Collectors;

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

        String str = "JD\nJD\nJD";
        System.out.println(str);
        System.out.println(str.lines().collect(Collectors.toList()));
    }
}
```



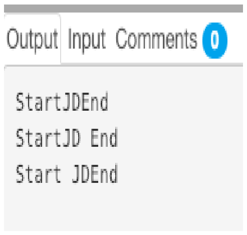
```
JD
JD
JD
[JD, JD, JD]
```

strip(), stripLeading(), stripTrailing() `strip()` - Removes the white space from both, beginning and the end of string.

But we already have trim(). Then what’s the need of strip()? `strip()` is “Unicode-aware” evolution of `trim()`. When `trim()` was introduced, Unicode wasn’t evolved. Now, the new strip() removes all kinds of whitespaces leading and trailing(check the method `Character.isWhitespace(c)` to know if a unicode is whitespace or not).

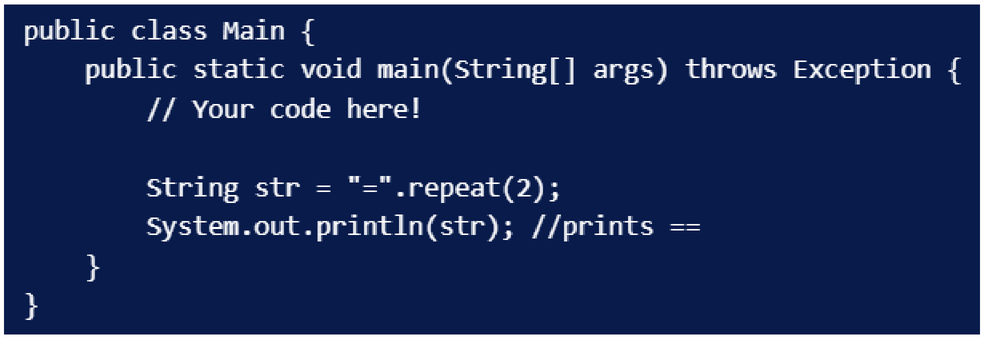


```
public class Main {
    public static void main(String[] args) throws Exception {
        // Your code here!

        String str = " JD ";
        System.out.print("Start");
        System.out.print(str.strip());
        System.out.println("End");

        System.out.print("Start");
        System.out.print(str.stripLeading());
        System.out.println("End");

        System.out.print("Start");
        System.out.print(str.stripTrailing());
        System.out.println("End");
    }
}
```



```
StartJDEnd
StartJD End
Start JDEnd
```

repeat(int) The repeat method simply repeats the string that many numbers of times as mentioned in the method in the form of an int.



```
public class Main {
    public static void main(String[] args) throws Exception {
        // Your code here!

        String str = "=".repeat(2);
        System.out.println(str); //prints ==
    }
}
```

Local-Variable Syntax for Lambda Parameters

Local-Variable Syntax for Lambda Parameters is the only language feature release in Java 11. In Java 10, Local Variable Type Inference was introduced. Thus we could infer the type of the variable from the RHS - `var list = new ArrayList<String>();` JEP 323 allows `var` to be used to declare the formal parameters of an implicitly typed lambda expression. We can now define :

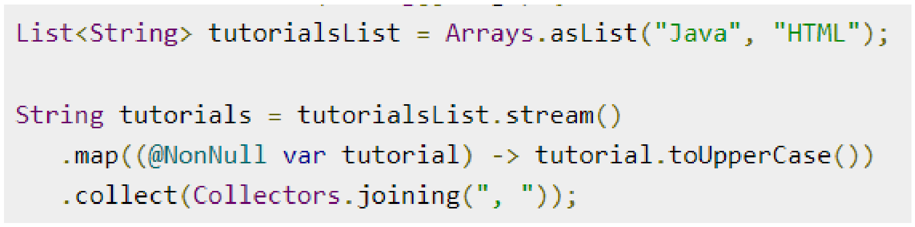
```
(var s1, var s2) -> s1 + s2
```

This was possible in Java 8 too but got removed in Java 10. Now it's back in Java 11 to keep things uniform.

But why is this needed when we can just skip the type in the lambda? If you need to apply an annotation just as @Nullable, you cannot do that without defining the type.

Limitation of this feature - You must specify the type var on all parameters or none. Things like the following are not possible:

```
(@NonNull var value1, @Nullable var value2) -> value1 + value2 // Correct one
(var s1, s2) -> s1 + s2 //no skipping allowed
(var s1, String y) -> s1 + y //no mixing allowed
var s1 -> s1 //not allowed. Need parentheses if you use var in lambda.
```



```
List<String> tutorialsList = Arrays.asList("Java", "HTML");

String tutorials = tutorialsList.stream()
   .map((@NonNull var tutorial) -> tutorial.toUpperCase())
   .collect(Collectors.joining(", "));
```

Nested Based Access Control

Java 11 introduced a concept of nested class where we can declare a class within a class. This nesting of classes allows to logically group the classes to be used in one place, making them more readable and maintainable. Nested class can be of four types −

- Static nested classes
- Non-static nested classes
- Local classes
- Anonymous classes

Java 11 also provide the concept of nestmate to allow communication and verification of nested classes.

```
import java.util.Arrays;
import java.util.Set;
import java.util.stream.Collectors;

public class APITester {
   public static void main(String[] args) {
      boolean isNestMate = APITester.class.isNestmateOf(APITester.Inner.class);
      boolean nestHost = APITester.Inner.class.getNestHost() == APITester.class;

      System.out.println(isNestMate);
      System.out.println(nestHost);

      Set<String> nestedMembers = Arrays.stream(APITester.Inner.class.getNestMembers())
         .map(Class::getName)
         .collect(Collectors.toSet());
      System.out.println(nestedMembers);
   }
   public class Inner{}
}
```

Output

```
true
true
[APITester$Inner, APITester]
```

Remove the Java EE and CORBA Modules

The modules were already deprecated in Java 9. They are now completely removed. Following packages are removed: `java.xml.ws`, `java.xml.bind`, `java.activation`, `java.xml.ws.annotation`, `java.corba`, `java.transaction`, `java.se.ee`, `jdk.xml.ws`, `jdk.xml.bind` .

Flight Recorder

Flight Recorder which earlier used to be a commercial add-on in Oracle JDK is now open-sourced since Oracle JDK is itself not free anymore. JFR is a profiling tool used to gather diagnostics and profiling data from a running Java application. Its performance overhead is negligible and that's usually below 1%. Hence it can be used in production applications.

Reading/Writing Strings to and from the Files

Java 11 strives to make reading and writing of String convenient. It has introduced the following methods for reading and writing to/from the files:

- readString()
- writeString()

Following code showcases an example of this

```
Path path = Files.writeString(Files.createTempFile("test", ".txt"), "This was posted on JD");
System.out.println(path);
String s = Files.readString(path);
System.out.println(s); //This was posted on JD
```

Not Predicate

Java 11 introduced new method to Predicate interface as not() to negate an existing predicate similar to negate method.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class APITester {
   public static void main(String[] args) {
      List<String> tutorialsList = Arrays.asList("Java", "\n", "HTML", " ");

      List<String> tutorials = tutorialsList.stream()
         .filter(Predicate.not(String::isBlank))
         .collect(Collectors.toList());

      tutorials.forEach(tutorial -> System.out.println(tutorial));
   }
}
```

HTTP Client

Java 11 standardizes the Http CLient API. The new API supports both HTTP/1.1 and HTTP/2. It is designed to improve the overall performance of sending requests by a client and receiving responses from the server. It also natively supports WebSockets.

An enhanced HttpClient API was introduced in Java 9 as an experimental feature. With Java 11, now HttpClient is a standard. It is recommended to use instead of other HTTP Client APIs like Apache Http Client API. It is quite feature rich and now Java based applications can make HTTP requests without using any external dependency.

Steps

Following are the steps to use an HttpClient.

- Create HttpClient instance using HttpClient.newBuilder() instance
- Create HttpRequest instance using HttpRequest.newBuilder() instance
- Make a request using httpClient.send() and get a response object.

```java
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.time.Duration;

public class APITester {
   public static void main(String[] args) {
      HttpClient httpClient = HttpClient.newBuilder()
         .version(HttpClient.Version.HTTP_2)
         .connectTimeout(Duration.ofSeconds(10))
         .build();
         try {
            HttpRequest request = HttpRequest.newBuilder()
            .GET()
            .uri(URI.create("https://www.google.com"))
            .build();
            HttpResponse<String> response = httpClient.send(request,
            HttpResponse.BodyHandlers.ofString());

         System.out.println("Status code: " + response.statusCode());
         System.out.println("Headers: " + response.headers().allValues("content-type"));
         System.out.println("Body: " + response.body());
      } catch (IOException | InterruptedException e) {
         e.printStackTrace();
      }
   }
}
```

Output

It will print the following output.

```
Status code: 200
Headers: [text/html; charset=ISO-8859-1]
Body: <!doctype html>
...
</html>
```

File APIs

Java 11 introduced an easy way to read and write files by providing new overloaded methods without writing much boiler plate code.

```java
import java.io.File;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;

public class APITester {
   public static void main(String[] args) {
      try {
         Path tempFilePath = Files.writeString(
            Path.of(File.createTempFile("tempFile", ".tmp").toURI()),
            "Welcome to TutorialsPoint",
            Charset.defaultCharset(), StandardOpenOption.WRITE);

         String fileContent = Files.readString(tempFilePath);

         System.out.println(fileContent);
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Output

```
Welcome to TutorialsPoint
```

Optional Class

Java 11 introduced new method to Optional class as isEmpty() to check if value is present. isEmpty() returns false if value is present otherwise true.

It can be used as an alternative of isPresent() method which often needs to negate to check if value is not present.

Consider the following example –

```
import java.util.Optional;

public class APITester {
   public static void main(String[] args) {
      String name = null;

      System.out.println(!Optional.ofNullable(name).isPresent());
      System.out.println(Optional.ofNullable(name).isEmpty());

      name = "Joe";
      System.out.println(!Optional.ofNullable(name).isPresent());
      System.out.println(Optional.ofNullable(name).isEmpty());
   }
}
```

Output

```
true
true
false
false
```

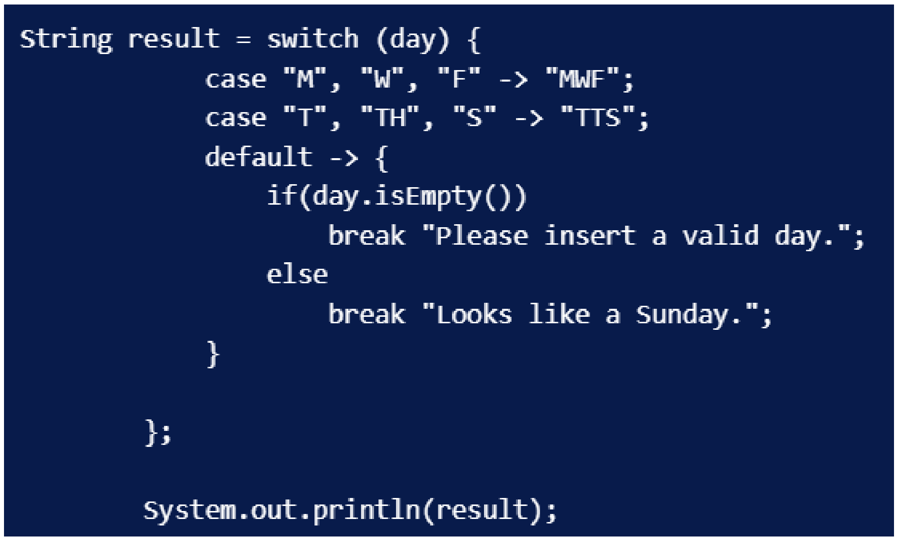
Switch Expressions (Preview)

Java 12 has enhanced Switch expressions for Pattern matching. Introduced as a preview language feature, the new Syntax is `L ->`. Following are some things to note about Switch Expressions:

- The new Syntax removes the need for break statement to prevent fall throughs.
- Switch Expressions don't fall through anymore.
- Furthermore, we can define multiple constants in the same label.

- default case is now compulsory in Switch Expressions.
- break is used in Switch Expressions to return values from a case itself.

With the new Switch expression, we don’t need to set break everywhere thus prevent logic errors!



```
String result = switch (day) {
            case "M", "W", "F" -> "MWF";
            case "T", "TH", "S" -> "TTS";
            default -> {
                if(day.isEmpty())
                    break "Please insert a valid day.";
                else
                    break "Looks like a Sunday.";
            }

        };

        System.out.println(result);
```

File.mismatch method

Java 12 added the following method to compare two files:

```
public static long mismatch(Path path, Path path2) throws IOException
```

This method returns the position of the first mismatch or -1L if there is no mismatch.
Two files can have a mismatch in the following scenarios:

- If the bytes are not identical. In this case, the position of the first mismatching byte is returned.
- File sizes are not identical. In this case, the size of the smaller file is returned.

Compact Number Formatting

```java
import java.text.NumberFormat;
import java.util.Locale;

public class CompactNumberFormatting {


    public static void main(String[] args)
    {
        System.out.println("Compact Formatting is:");
        NumberFormat upvotes = NumberFormat
                .getCompactNumberInstance(new Locale("en", "US"), NumberFormat.Style.SHORT);
        upvotes.setMaximumFractionDigits(1);

        System.out.println(upvotes.format(2592) + " upvotes");


        NumberFormat upvotes2 = NumberFormat
                .getCompactNumberInstance(new Locale("en", "US"), NumberFormat.Style.LONG);
        upvotes2.setMaximumFractionDigits(2);
        System.out.println(upvotes2.format(2011) + " upvotes");
    }


}
```

Teeing Collectors

Teeing Collector is the new collector utility introduced in the Streams API. This collector has three arguments - Two collectors and a Bi-function. All input values are passed to each collector and the result is available in the Bi-function.

```java
double mean = Stream.of(1, 2, 3, 4, 5)
                .collect(Collectors.teeing(
                        summingDouble(i -> i),
                        counting(),
                        (sum, n) -> sum / n));

System.out.println(mean);
```

The output is **3.0**.

Java Strings New Methods

4 new methods have been introduced in Java 12 which are:

indent(n) method

Adjust the indention of each line of string based on argument passed.

Usage

- **n > 0** - insert space at the begining of each line.
- **n < 0** - remove space at the begining of each line.
- **n < 0 and n < available spaces** - remove all leading space of each line.
- **n = 0** - no change.

transform(Function<? super String,? extends R> f) method

Transforms a string to give result as R.

Usage

```
String transformed = text.transform(value -> new StringBuilder(value).reverse().toString());
```

Optional<String> describeConstable() method

Returns Optional Object containing description of String instance.

Usage

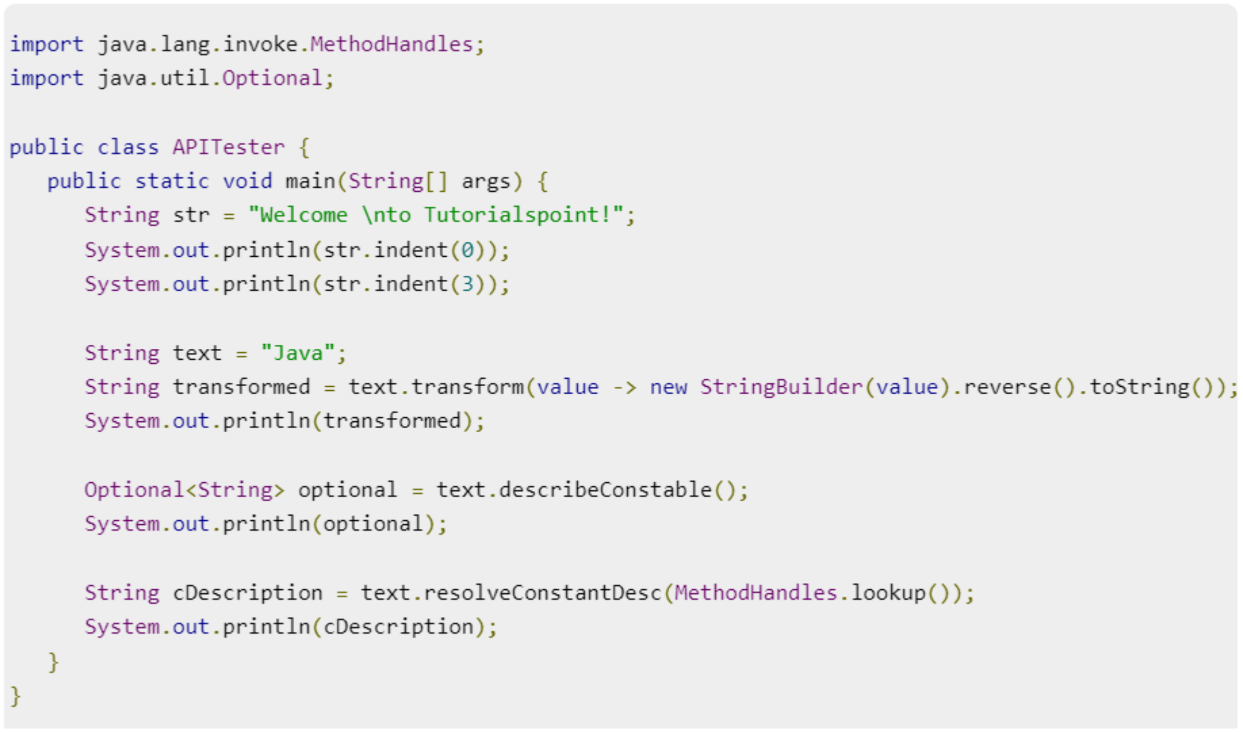
```
Optional<String> optional = message.describeConstable();
```

resolveConstantDesc(MethodHandles.Lookup lookup) method

Returns descriptor instance string of given string.

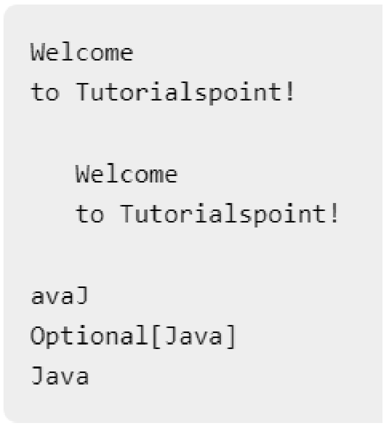
Usage

```
String constantDesc = message.resolveConstantDesc(MethodHandles.lookup());
```



```java
import java.lang.invoke.MethodHandles;
import java.util.Optional;

public class APITester {
   public static void main(String[] args) {
      String str = "Welcome \nto Tutorialspoint!";
      System.out.println(str.indent(0));
      System.out.println(str.indent(3));

      String text = "Java";
      String transformed = text.transform(value -> new StringBuilder(value).reverse().toString());
      System.out.println(transformed);

      Optional<String> optional = text.describeConstable();
      System.out.println(optional);

      String cDescription = text.resolveConstantDesc(MethodHandles.lookup());
      System.out.println(cDescription);
   }
}
```

Output



```
Welcome
to Tutorialspoint!

   Welcome
   to Tutorialspoint!

avaJ
Optional[Java]
Java
```

Pattern Matching for instanceof (Preview)

Another Preview Language feature! The old way to typecast a type to another type is:

```
if (obj instanceof String) {

    String s = (String) obj;

    // use s in your code from here

}
```

The new way is :

```
if (obj instanceof String s) {

    // can use s directly here

}
```

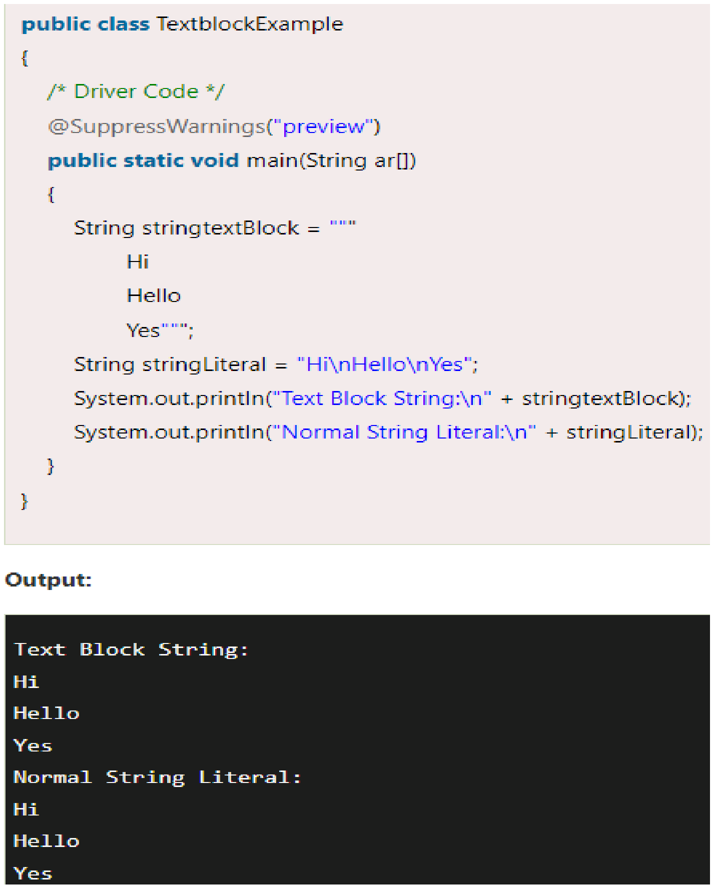
This saves us some typecasting which were unnecessary.

Text Blocks

This is an example of a preview feature. It makes it simple to construct multiline strings. A pair of triple-double quotations must surround the multiline string.

There are no additional properties on the string object formed with text blocks. It's a more convenient approach to make multiline strings. We can't make a single-line string with text blocks.

A line terminator must follow the initial triple-double quotations.



```
public class TextblockExample
{
    /* Driver Code */
    @SuppressWarnings("preview")
    public static void main(String ar[])
    {
        String stringtextBlock = """
                Hi
                Hello
                Yes""";
        String stringLiteral = "Hi\nHello\nYes";
        System.out.println("Text Block String:\n" + stringtextBlock);
        System.out.println("Normal String Literal:\n" + stringLiteral);
    }
}
```

Output:

```
Text Block String:
Hi
Hello
Yes
Normal String Literal:
Hi
Hello
Yes
```

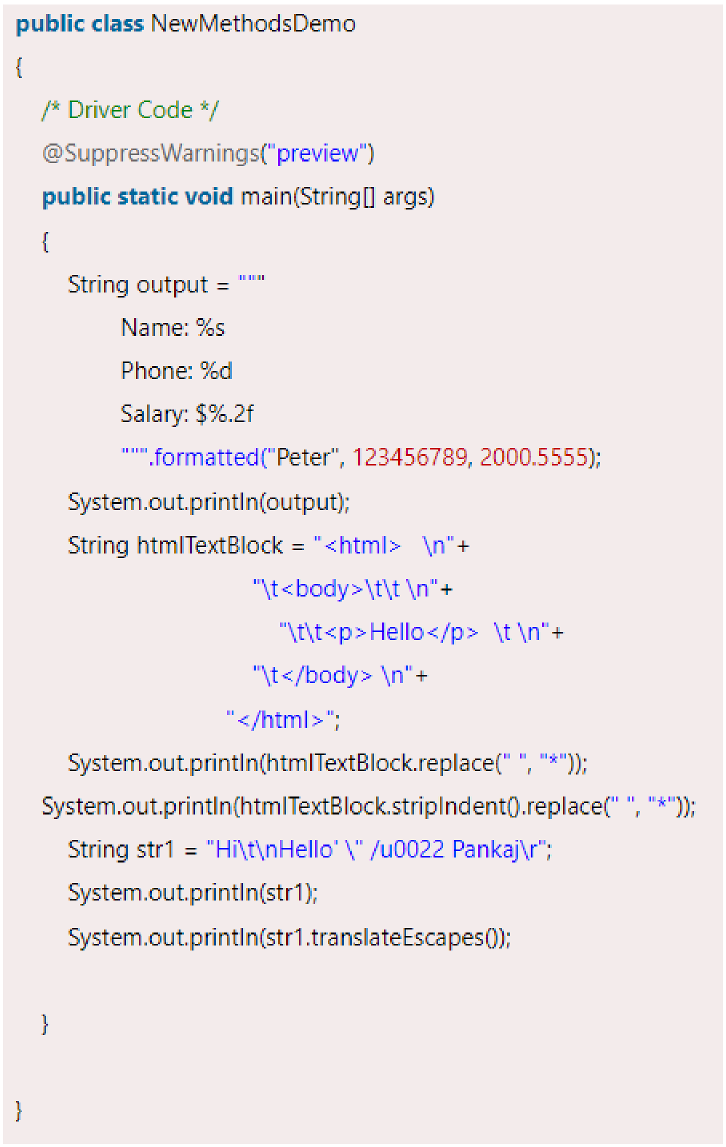
New Methods in String Class for Text Blocks

The String class now has three additional methods related to the text blocks functionality.

formatted(Object... args) is a function comparable to String format(). It's there to help with text block formatting.

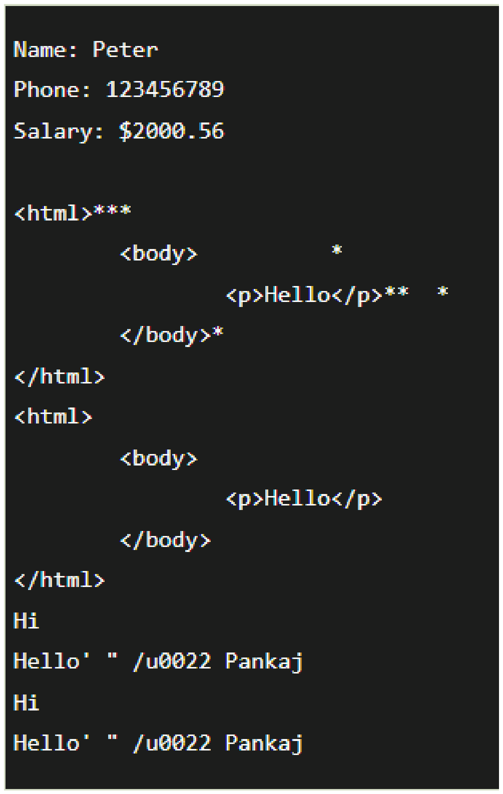
stripIndent() is a function that removes the white space characters at the start and end of each line in a text block. The text blocks employ this mechanism to preserve the relative indentation of the content.

translateEscapes() It returns a string containing the value this string, with escape sequences translated as if they were in a string literal.



```java
public class NewMethodsDemo
{
  /* Driver Code */
  @SuppressWarnings("preview")
  public static void main(String[] args)
  {
    String output = """
        Name: %s
        Phone: %d
        Salary: $%.2f
        """.formatted("Peter", 123456789, 2000.5555);
    System.out.println(output);
    String htmlTextBlock = "<html>   \n"+
                           "\t<body>\t\t \n"+
                             "\t\t<p>Hello</p>  \t \n"+
                           "\t</body> \n"+
                         "</html>";
    System.out.println(htmlTextBlock.replace(" ", "*"));
  System.out.println(htmlTextBlock.stripIndent().replace(" ", "*"));
    String str1 = "Hi\t\nHello' \" /u0022 Pankaj\r";
    System.out.println(str1);
    System.out.println(str1.translateEscapes());

  }

}
```

Output:



```
Name: Peter
Phone: 123456789
Salary: $2000.56

<html>***
        <body>          *
                <p>Hello</p>**   *
        </body>*
</html>
<html>
        <body>
                <p>Hello</p>
        </body>
</html>
Hi
Hello' " /u0022 Pankaj
Hi
Hello' " /u0022 Pankaj
```

Switch Expressions Enhancements

In the 12 release of Java, switch expressions were added as a trial feature. The only difference is that "break" has been replaced with "yield" to return a value from the case statement in Java 13.

```
    // from java 13 onwards - multi-label case statements
    switch (choice) {
    case 1, 2, 3:
        System.out.println(choice);
        break;
    default:
        System.out.println("integer is greater than 3");
    }
    // switch expressions, use yield to return, in Java 12 it was break
    int x = switch (choice) {
    case 1, 2, 3:
        yield choice;
    default:
        yield -1;
    };
    System.out.println("x = " + x);
}
enum Day {
    SUN, MON, TUE
};
@SuppressWarnings("preview")
public String getDay(Day d) {
    String day = switch (d) {
    case SUN -> "Sunday";
    case MON -> "Monday";
    case TUE -> "Tuesday";
    };
    return day;
```

Reimplement the Legacy Socket API

The underlying implementation of the java.net.Socket and java.net.ServerSocket APIs has been updated in Java version 13. NioSocketImpl, the new implementation, is a drop-in replacement for PlainSocketImpl.

Instead of synchronized methods, it employs java.util.concurrent locks. Utilize the java option -Djdk.net.usePlainSocketImpl if we want to use the legacy implementation.