Java 8:

1. Default Methods

Default methods allow developers to add new methods to interfaces without breaking existing implementations. They provide a default implementation that can be overridden by implementing classes if needed.

Default Methods for Backward Compatibility: Allowing the addition of methods to existing interfaces without breaking the implementing classes.

A default method cannot override a method from “java.lang.Object”

```
package DefaultMethods;

2 usages  1 implementation  PramodCGowda
interface Vehicle {
    1 usage  1 implementation  PramodCGowda
    void start();

    // Default method
    1 usage  1 override  PramodCGowda
    default void turnOff() {
        System.out.println("The vehicle is turning off.");
    }
}
```

Combining Default Methods and Multiple Interfaces

When a class implements multiple interfaces with default methods, there might be conflicts if the same method is provided by more than one interface.

```
class Car implements Vehicle, Machine {
    @Override
    public void start() {
        // Resolve conflict by specifying which interface method to use
        Vehicle.super.start();
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();  // Calls the start method from Vehicle interface
    }
}
```

Static Methods

Static methods in interfaces are similar to static methods in classes. They belong to the interface and can be called without an instance of the interface.

Static Methods for Utility Functions: Providing utility methods related to the interface, similar to what is often done with static utility classes.

Default Methods	Static Methods
Default methods are used for backward compatibility.	Static Methods are used for utitlity functions or Helper functions.
Default methods can be overridden in implementing class	Static methods can't be overridden in implementing class

```
interface Calculator {
    // Static method
    1 usage   new *
    static int add(int a, int b) {
        return a + b;
    }
}

no usages   new *
public class StaticMethods {
    no usages   new *
    public static void main(String[] args) {
        // Calling static method of the interface
        int result = Calculator.add( a: 5,  b: 10);
        System.out.println("Sum: " + result);
    }
}
```

Functional Interfaces

A functional interface is an interface with a single abstract method. They are used primarily for lambda expressions and method references in Java 8. The `@FunctionalInterface` annotation can be used to indicate a functional interface, though it is optional.

```
@FunctionalInterface
interface Greeting {
    1 usage   new *
    void sayHello(String name);
}

no usages   new *
public class FuncInterface {
    no usages   new *
    public static void main(String[] args) {
        // Using a lambda expression
        Greeting greeting = (name) -> System.out.println("Hello, " + name);
        greeting.sayHello( name: "Alice");
    }
}
```

Functional Interfaces for Lambdas and Streams: Simplifying the use of lambda expressions and the new Streams API, enhancing code readability and reducing boilerplate code.

Common functional interfaces in Java include `Runnable`, `Callable`, `Comparator`, and those in `java.util.function`.

Lambda Expression:

Lambda expressions in Java SE 8 were introduced to facilitate functional programming and to provide a more expressive way of writing code.

In Java, Lambda expressions basically express instances of functional interfaces (An interface with a single abstract method is called a functional interface).

Lambda Expressions in Java are the same as lambda functions which are the short block of code that accepts input as parameters and returns a resultant value. Lambda Expressions are recently included in Java SE 8.

Functionalities of Lambda Expression in Java Lambda Expressions implement the only abstract function and therefore implement functional interfaces lambda expressions are added in Java 8 and provide the below functionalities.

Enable to treat functionality as a method argument, or code as data.

A function that can be created without belonging to any class.

A lambda expression can be passed around as if it was an object and executed on demand.

```
@FunctionalInterface
interface Test {
    1 usage   new *
    void SayHello(String name);
}
no usages   new *
public class LambdaExpr {
    no usages   new *
    public static void main(String[] args) {
        Test obj = (name) -> System.out.println("hello " + name);
        obj.SayHello( name: "Pramod");
    }
}
```

Predicates:

In Java, `Predicate` is a functional interface that was introduced in Java 8 as part of the `java.util.function` package.

It represents a single argument function that returns a boolean value.

This is particularly useful for filtering or matching conditions in collections or streams.

```java
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

```java
public class Predicates1 {

    no usages   new *
    public static void main(String[] args) {
        Predicate<Integer> predicate = (num) -> num == 2020;
        System.out.println(predicate.test( t: 2020));
        System.out.println(predicate.test( t: 2021));
    }
}
```

```java
public class Predicate2 {
    no usages   new *
    public static void main(String[] args) {
        List<Country> countries = new ArrayList<>();
        countries.add(new Country( countryName: "India",  countryCode: 91));
        countries.add(new Country( countryName: "USA",  countryCode: 1));

        Predicate<Country> predicate = (country) -> country.getCountryName().startsWith("I");
        for(Country c : countries) {
            if(predicate.test(c)) {
                System.out.println(c);
            }
        }
    }
}
```

Function:

Function<T, R> Represents a function that accepts one argument of type T and produces a result of type R. Method: R apply(T t)

```
Function<String, Integer> lengthFunction = str -> str.length();
int length = lengthFunction.apply("Hello"); // 5
```

Consumer:

Consumer<T> Represents an operation that accepts a single input argument and returns no result. Method: void accept(T t)

```
Consumer<String> printConsumer = str -> System.out.println(str);
printConsumer.accept("Hello, World!"); // Prints "Hello, World!"
```

Supplier:

Supplier<T> Represents a supplier of results. This is a functional interface that takes no arguments and returns a result of type T. Method: T get()

```
Supplier<Double> randomSupplier = () -> Math.random();
double randomValue = randomSupplier.get(); // A random number
```

Method References:

Method references in Java 8 are a shorthand syntax for invoking a method directly in situations where a lambda expression would be used.

Types of Method References

There are four main types of method references in Java:

1. **Reference to a static method**
2. **Reference to an instance method of a particular object**
3. **Reference to an instance method of an arbitrary object of a particular type**
4. **Reference to a constructor**

Reference to a static method: ClassName::staticMethodName

Reference to an instance method of a particular object: instance::instanceMethodName

Reference to an instance method of an arbitrary object of a particular type: ClassName::instanceMethodName

Reference to a constructor: ClassName::new

Function Type	Method Signature	Input parameters	Returns	When to use?
Predicate<T>	boolean test(T t)	one	boolean	Use in conditional statements
Function<T, R>	R apply(T t)	one	Any type	Use when to perform some operation & get some result
Consumer<T>	void accept(T t)	one	Nothing	Use when nothing is to be returned
Supplier<R>	R get()	None	Any type	Use when something is to be returned without passing any input
BiPredicate<T, U>	boolean test(T t, U u)	two	boolean	Use when Predicate needs two input parameters
BiFunction<T, U, R>	R apply(T t, U u)	two	Any type	Use when Function needs two input parameters
BiConsumer<T, U>	void accept(T t, U u)	two	Nothing	Use when Consumer needs two input parameters
UnaryOperator<T>	public T apply(T t)	one	Any Type	Use this when input type & return type are same instead of Function<T, R>
BinaryOperator<T>	public T apply(T t, T t)	two	Any Type	Use this when both input types & return type are same instead of BiFunction<T, U, R>

Image source: https://i0.wp.com/javatechonline.com/wp-content/uploads/2021/04/image6-1.jpg

Predefine Functional Interfaces

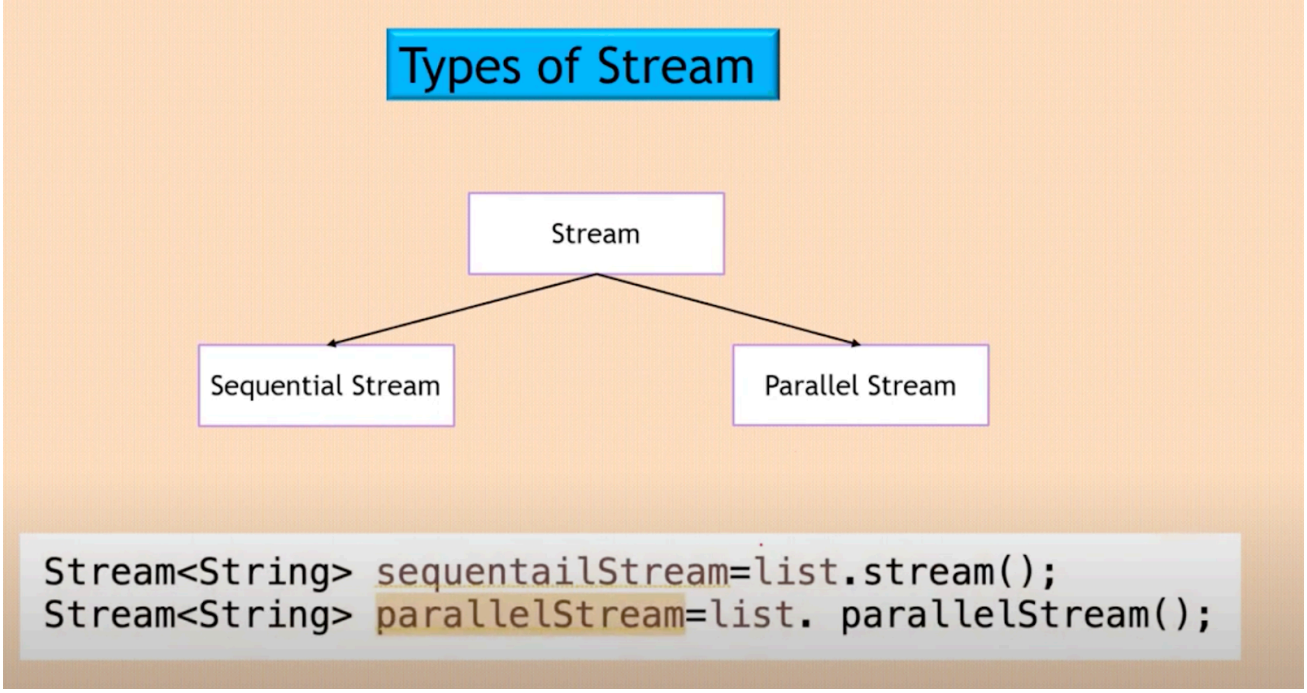
Property	Predicate	Function	Consumer	Supplier
Definition	To take some input value and perform conditional checks	To take some input value and return the result of any type	To Consume some input value and perform required operations and won't return anything	To Supply some value based on our requirement
Interface Declaration	Interface Predicate<T> { }	Interface Function<T,R> { }	Interface Consumer<T> { }	Interface Supplier<R> { }
SAM	Public boolean test(T t)	Public R apply(T t)	Public void accept(T t)	Public R get()
Default Method	and() , or() ,negate()	andThen() Compose()	andThen	
Static Method	isEqual()	Identify()		
Real Time Usages	filter method	Comparing	Foreach	Generate in Stream

Image source: https://www.youtube.com/c/JavaExpress

Stream API:

Stream is a powerful tool for processing sequences of elements, such as collections or arrays, in a functional style. Streams allow you to perform operations like filtering, mapping, and reducing on data without modifying the underlying data structure.

Operation	Method Signature	Type Of Operation	What It Does?
filter()	Stream<T> filter(Predicate<T> predicate)	Intermediate	Returns a stream of elements which satisfy the given predicate.
map()	Stream<R> map(Function< T, R> mapper)	Intermediate	Returns a stream consisting of results after applying given function to elements of the stream.
distinct()	Stream<T> distinct()	Intermediate	Returns a stream of unique elements.
sorted()	Stream<T> sorted()	Intermediate	Returns a stream consisting of elements sorted according to natural order.
limit()	Stream<T> limit(long maxSize)	Intermediate	Returns a stream containing first *n* elements.
skip()	Stream<T> skip(long n)	Intermediate	Returns a stream after skipping first *n* elements.
forEach()	void forEach(Consumer<T> action)	Terminal	Performs an action on all elements of a stream.
toArray()	Object[] toArray()	Terminal	Returns an array containing elements of a stream.
reduce()	T reduce(T identity, BinaryOperator<T> accumulator)	Terminal	Performs reduction operation on elements of a stream using initial value and binary operation.
collect()	R collect(Collector<T> collector)	Terminal	Returns mutable result container such as List or Set.
min()	Optional<T> min(Comparator<T> comparator)	Terminal	Returns minimum element in a stream wrapped in an Optional object.
max()	Optional<T> max(Comparator<T> comparator)	Terminal	Returns maximum element in a stream wrapped in an Optional object.
count()	long count()	Terminal	Returns the number of elements in a stream.
anyMatch()	boolean anyMatch(Predicate<T> predicate)	Terminal	Returns true if any one element of a stream matches with given predicate.
allMatch()	boolean allMatch(Predicate<T> predicate)	Terminal	Returns true if all the elements of a stream matches with given predicate.
noneMatch()	boolean noneMatch(Predicate< T> predicate)	Terminal	Returns true only if all the elements of a stream doesn't match with given predicate.
findFirst()	Optional<T> findFirst()	Terminal	Returns first element of a stream wrapped in an Optional object.
findAny()	Optional<T> findAny()	Terminal	Randomly returns any one element in a stream.



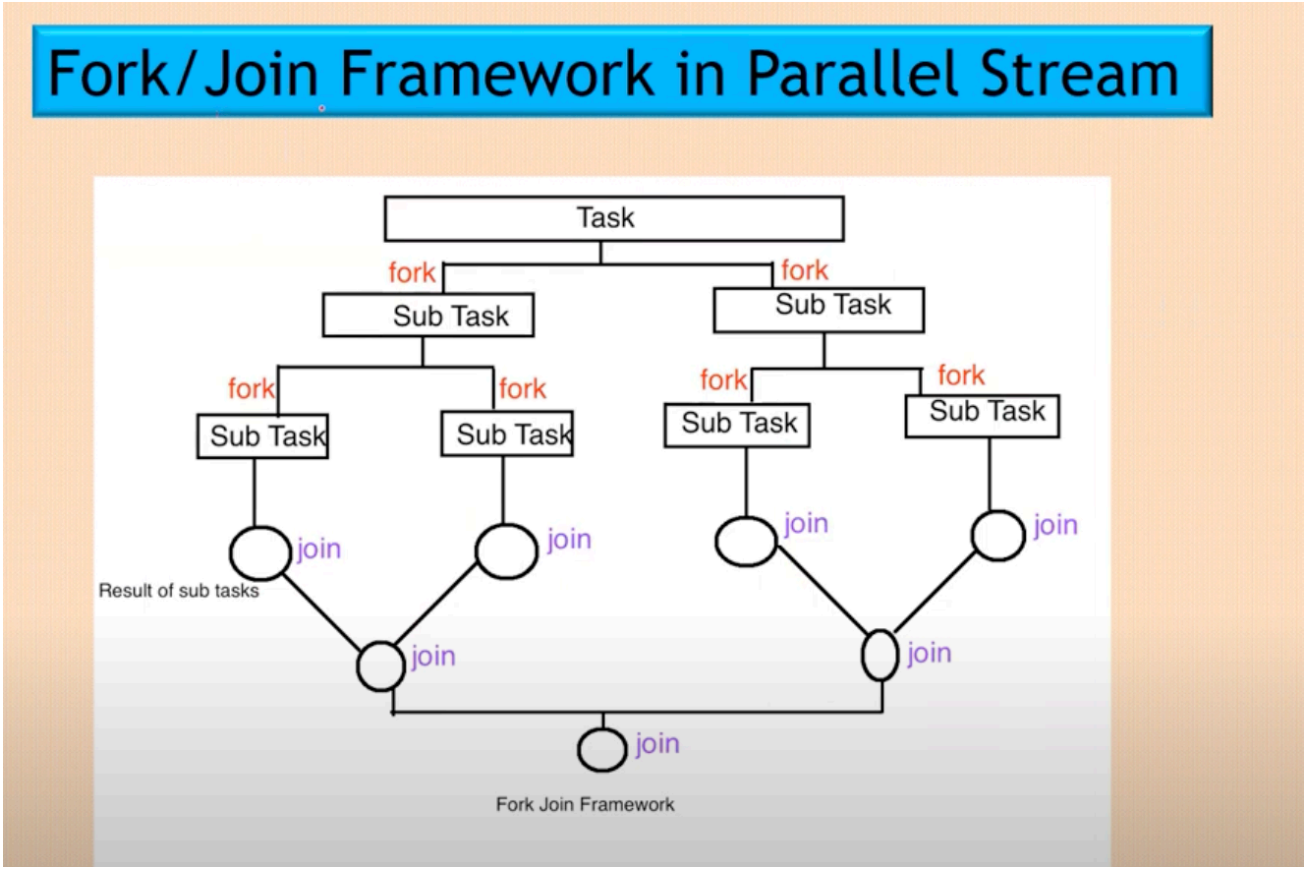
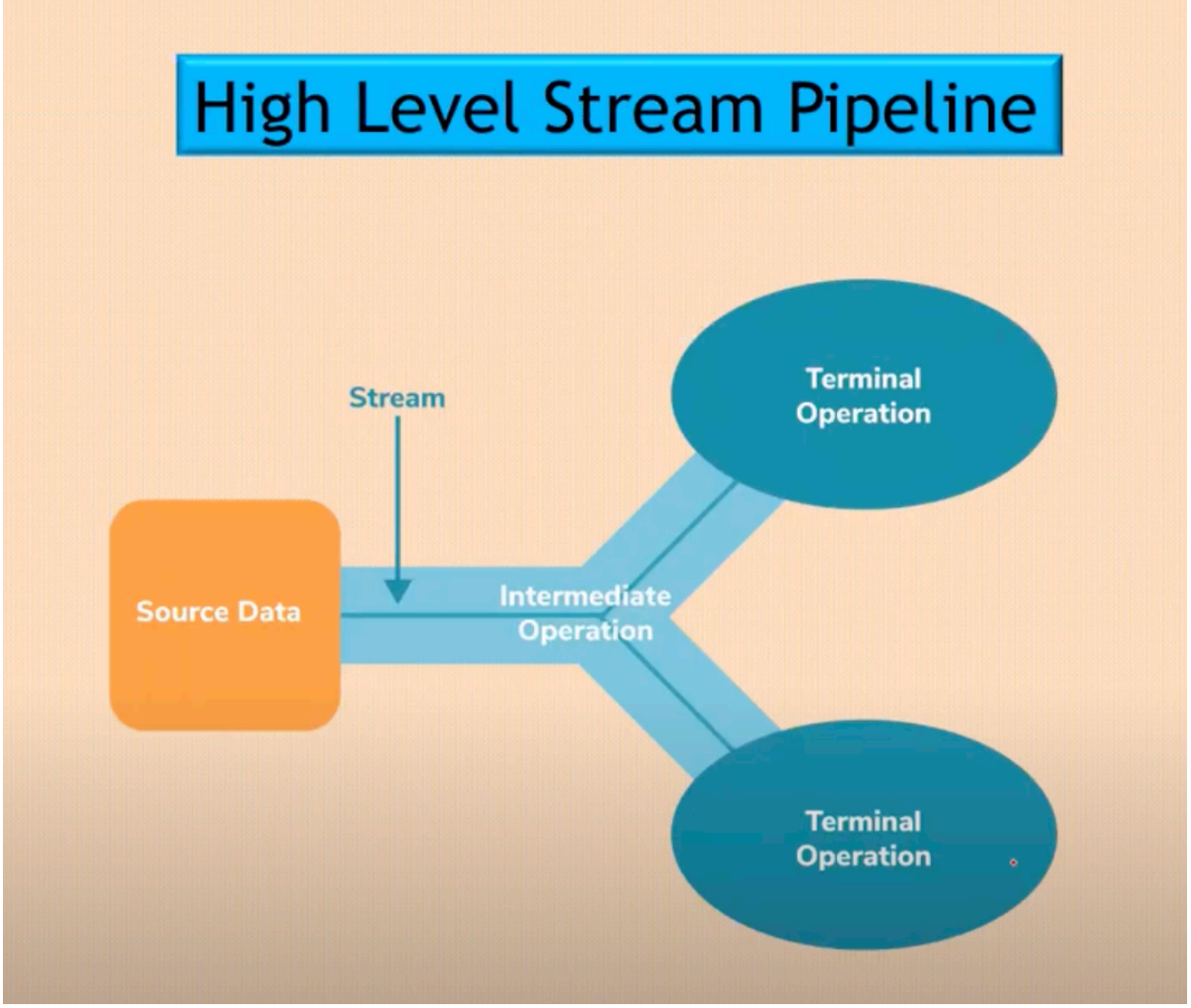


Image source: https://www.youtube.com/c/JavaExpress



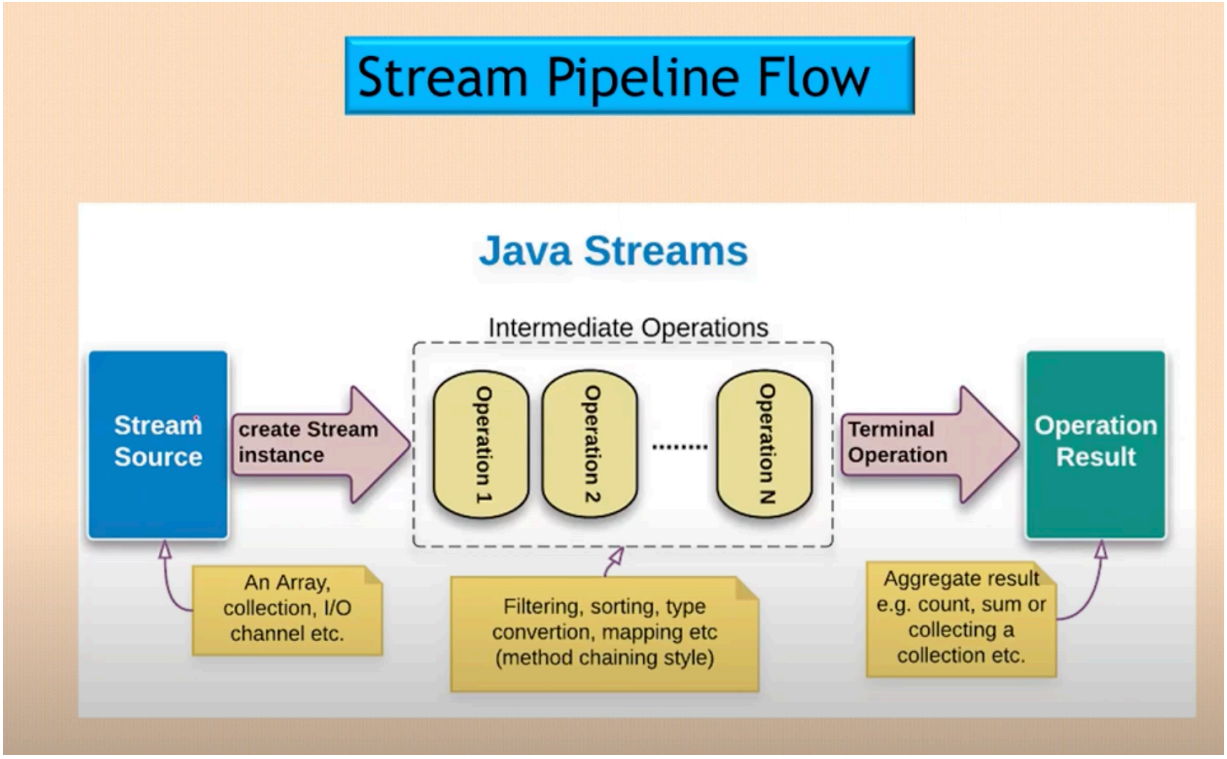


Image source: https://www.youtube.com/c/JavaExpress

Date and Time API:

Java 8 introduced a new java.time package that provides a comprehensive and robust Date and Time API, replacing the outdated and cumbersome java.util.Date and java.util.Calendar classes. The new API is immutable, thread-safe, and designed to handle a variety of date and time operations in a clear and concise manner.

Key Classes in the java.time Package

1. LocalDate:
    - Represents a date (year, month, day) without time and timezone information.

Example:
```
LocalDate date = LocalDate.now();  // Current date

LocalDate specificDate = LocalDate.of(2024, Month.AUGUST, 15);  // Specific date
```

2. LocalTime:
    - Represents a time (hour, minute, second, nanosecond) without date and timezone information.

Example:
```
LocalTime time = LocalTime.now();  // Current time

LocalTime specificTime = LocalTime.of(14, 30);  // Specific time 14:30
```

3. LocalDateTime:
    - Combines LocalDate and LocalTime to represent both date and time without timezone information.

Example:
```
LocalDateTime dateTime = LocalDateTime.now();  // Current date and time

LocalDateTime specificDateTime = LocalDateTime.of(2024, Month.AUGUST, 15, 14, 30);  // Specific date and time
```

4. ZonedDateTime:
    - Represents a date and time with timezone information.

Example:
```
ZonedDateTime zonedDateTime = ZonedDateTime.now();  // Current date and time with timezone

ZonedDateTime specificZonedDateTime = ZonedDateTime.of(2024, Month.AUGUST, 15, 14, 30, 0, 0,
ZoneId.of("America/New_York"));
```

5. Instant:
    - Represents a specific moment on the timeline in UTC (Coordinated Universal Time).

Example:
```
Instant instant = Instant.now();  // Current moment in UTC
```

6. Duration:
    - Represents a time-based amount of time, such as "34.5 seconds."

Example:
```
Duration duration = Duration.ofHours(5);  // Duration of 5 hours
```

7. Period:
    - Represents a date-based amount of time, such as "3 years, 2 months, and 1 day."

Example:
```
Period period = Period.ofYears(2);  // Period of 2 years
```

8. DateTimeFormatter:
    - Used to format and parse date-time objects.

Example:
```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");

String formattedDate = LocalDate.now().format(formatter);  // Format the current date

LocalDate parsedDate = LocalDate.parse("15/08/2024", formatter);
```

Java 9:

Private Interface Methods

- Private Methods in Interfaces: Java 9 allows interfaces to have private methods. These methods can be used to share common code between default and static methods in the interface.

```
interface MyInterface {

    default void defaultMethod() {

        privateMethod();

    }



    private void privateMethod() {

        System.out.println("Private method in interface");

    }

}
```

Try-With-Resources

- Improved Try-With-Resources: Java 9 enhanced the try-with-resources statement to allow a final or effectively final variable to be used as a resource without declaring a new variable.

```
AutoCloseable resource = new MyResource();

try (resource) {

    // Use the resource

}
```

Collection Factory Methods:

Collection Factory Methods were introduced in Java 9 to simplify the creation of immutable collections like **List, Set, and Map**. These methods allow you to create collections with a concise and convenient syntax. The collections created using these factory methods are immutable, meaning they cannot be modified after their creation.

Key Points:

- **Immutability**: The collections created using these factory methods are immutable. Attempting to modify them (e.g., adding or removing elements) will result in an `UnsupportedOperationException`.
- **Convenience**: These factory methods provide a more concise way to create collections, compared to the traditional approach.

Factory Methods:

1. **List.of():** Creates an immutable list containing the specified elements.

Example:
```
List<String> list = List.of("a", "b", "c");

System.out.println(list);  // Output: [a, b, c]
```

2. **Set.of():** Creates an immutable set containing the specified elements.

Example:
```
Set<String> set = Set.of("a", "b", "c");

System.out.println(set);  // Output: [a, b, c]
```

Note: Duplicate elements are not allowed. Attempting to create a set with duplicates will throw an IllegalArgumentException.

3. **Map.of():** Creates an immutable map containing the specified key-value pairs.

Example:
```
Map<Integer, String> map = Map.of(1, "one", 2, "two", 3, "three");

System.out.println(map);  // Output: {1=one, 2=two, 3=three}
```

For larger maps (more than 10 entries), use Map.ofEntries().

4. **Map.ofEntries():** Creates an immutable map with an arbitrary number of entries using Map.Entry objects.

Example:
```
Map<Integer, String> map = Map.ofEntries(

   Map.entry(1, "one"),

   Map.entry(2, "two"),

   Map.entry(3, "three")

);

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

Stream API Enhancements:

takeWhile(Predicate<? super T> predicate):

- Returns a stream consisting of the longest prefix of elements that match the given predicate.
- The operation stops as soon as the predicate returns false for an element.

Example:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

List<Integer> result = numbers.stream()

                        .takeWhile(n -> n < 5)

                        .collect(Collectors.toList());

System.out.println(result);  // Output: [1, 2, 3, 4]
```

dropWhile(Predicate<? super T> predicate):

- Returns a stream consisting of the remaining elements after dropping the longest prefix of elements that match the given predicate.
- The operation skips elements as long as the predicate returns true, and then returns the remaining elements.

Example:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

List<Integer> result = numbers.stream()

                        .dropWhile(n -> n < 5)

                        .collect(Collectors.toList());

System.out.println(result);  // Output: [5, 6, 7, 8, 9]
```

ofNullable(T t):

- Returns a sequential Stream containing a single element if the provided element is non-null; otherwise, returns an empty stream.
- This method is useful when you want to create a stream from a potentially null object without having to handle the null explicitly.

Example:

```
Stream<String> nonNullStream = Stream.ofNullable("Hello");

Stream<String> nullStream = Stream.ofNullable(null);

System.out.println(nonNullStream.count());  // Output: 1

System.out.println(nullStream.count());     // Output: 0
```

iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> f):

- This method allows the creation of a stream that produces elements iteratively, as long as the specified predicate evaluates to true.
- It's an enhancement over the existing iterate method introduced in Java 8, which only took a seed and an update function.

Example:

```
Stream<Integer> stream = Stream.iterate(1, n -> n < 10, n -> n + 2);

stream.forEach(System.out::println);  // Output: 1, 3, 5, 7, 9
```

Improved Optional Integration:

- Java 9 improved the integration between Stream and Optional classes.
- Optional now has a stream() method that returns a stream with zero or one element, depending on whether a value is present.

Example:

```
Optional<String> optional = Optional.of("Hello");

List<String> result = optional.stream().collect(Collectors.toList());

System.out.println(result);  // Output: [Hello]
```

New Collectors Methods:

- Java 9 introduced new methods in the Collectors class to work more efficiently with Stream operations.
- Collectors.filtering() allows you to filter elements before collecting them.
- Collectors.flatMapping() is used to flatten nested structures before collecting.

Example:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);

List<Integer> result = numbers.stream()

                        .collect(Collectors.filtering(n -> n % 2 == 0, Collectors.toList()));

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

Java 10:

Local-Variable Type Inference (var)

- Java 10 introduced the var keyword, which allows local variables to have their types inferred by the compiler. This means that you can declare a variable without specifying its type, and the compiler will determine the type based on the assigned value.

Example:

- var list = new ArrayList<String>(); // The type of 'list' is inferred as ArrayList<String>
- var num = 10; // The type of 'num' is inferred as int

Copy Methods for Collections:

“Immutable collections can be safely shared among the multiple threads without the need for synchronization”.

Java 10 introduced new copyOf methods in the List, Set, and Map interfaces. These methods are used to create a new unmodifiable copy of an existing collection.

- List.copyOf(Collection<? extends E> coll)
- Set.copyOf(Collection<? extends E> coll)
- Map.copyOf(Map<? extends K, ? extends V> map)

These methods are particularly useful when you want to create a copy of an existing collection and ensure that the new collection is immutable.

Example:

```
List<String> originalList = List.of("one", "two", "three");

List<String> copiedList = List.copyOf(originalList);



Set<String> originalSet = Set.of("apple", "banana", "cherry");

Set<String> copiedSet = Set.copyOf(originalSet);



Map<String, Integer> originalMap = Map.of("key1", 1, "key2", 2);

Map<String, Integer> copiedMap = Map.copyOf(originalMap);



// copiedList, copiedSet, and copiedMap are immutable copies of the originals
```

- Memory Efficiency: If the original collection is already immutable and of the same type, copyOf returns the original collection itself rather than creating a new one. This helps in reducing unnecessary memory usage.

Enhancements to Collectors.toUnmodifiableList, toUnmodifiableSet, toUnmodifiableMap:

Although initially introduced in Java 9, these methods became more widely recognized with the copyOf methods. They allow us to collect elements into an unmodifiable collection directly when using the Stream API.

Example:

```
List<String> immutableList = Stream.of("a", "b", "c")
                                .collect(Collectors.toUnmodifiableList());

// immutableList is an unmodifiable list
```

Optional:

Changes aimed to reduce boilerplate code and improve the handling of optional values, especially when using Optional in a functional programming style.

1. Optional.orElseThrow() Method

- The orElseThrow() method as a more explicit replacement for the existing get() method in the Optional class. This method returns the contained value if it is present, and throws a NoSuchElementException if the value is absent.
    - Java 8: optional.get() throws NoSuchElementException if the value is not present.
    - Java 10: optional.orElseThrow() is the preferred way to achieve the same result. It improves code readability by making the intention of throwing an exception clearer.

Example:

```
Optional<String> optional = Optional.of("Hello");

String value = optional.orElseThrow(); // Returns "Hello"



Optional<String> emptyOptional = Optional.empty();

String value2 = emptyOptional.orElseThrow(); // Throws NoSuchElementException
```

2. Optional.orElseThrow(Supplier<? extends X> exceptionSupplier) Method

- This method allows us to throw a custom exception if the value is not present, using a supplier to provide the exception.

Example:

```
Optional<String> optional = Optional.empty();

String value = optional.orElseThrow(() -> new IllegalArgumentException("No value present!"));
```

3. Optional.ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction) Method

- Allows us to perform one action if a value is present and another if it is not.
- This can be useful for handling both scenarios in a concise manner without additional conditionals.

Example:

```
Optional<String> optional = Optional.of("Hello");

optional.ifPresentOrElse(

    value -> System.out.println("Value is present: " + value),

    () -> System.out.println("Value is not present")

);

Optional<String> emptyOptional = Optional.empty();

emptyOptional.ifPresentOrElse(

    value -> System.out.println("Value is present: " + value),

    () -> System.out.println("Value is not present")

);
```

4. Optional.stream() Method

- The stream() method returns a Stream containing the value if present, or an empty Stream if the value is absent. This is particularly useful when we want to integrate Optional values into a Stream pipeline.

Example:

```
Optional<String> optional = Optional.of("Hello");

Stream<String> stream = optional.stream();

stream.forEach(System.out::println); // Prints "Hello"


Optional<String> emptyOptional = Optional.empty();

Stream<String> emptyStream = emptyOptional.stream();

emptyStream.forEach(System.out::println); // Prints nothing
```

Java 11:

Local-Variable Syntax for Lambda Parameters

- Java 11 extends the var keyword, introduced in Java 10, to lambda expressions.
- Now we can use var for the parameters of implicitly typed lambda expressions.

Example:

```
(var x, var y) -> x + y
```

HttpClient API

- The HttpClient API, which was incubated in Java 9 and 10, became a standard feature in Java 11.
- This new API supports both synchronous and asynchronous programming models and is used to send HTTP requests and receive responses.

Example:

```
HttpClient client = HttpClient.newHttpClient();

HttpRequest request = HttpRequest.newBuilder()

    .uri(URI.create("https://example.com"))

    .build();

HttpResponse<String> response = client.send(request, BodyHandlers.ofString());

System.out.println(response.body());
```

Launch Single-File Source-Code Programs

Java 11 allows us to run a single Java source file without explicitly compiling it first using javac. The Java compiler and launcher handle this behind the scenes.

Example:

```
java HelloWorld.java
```

New String Methods

Java 11 introduced several new methods to the String class, including:

isBlank(): Checks if a string is empty or contains only white spaces.

lines(): Returns a stream of lines extracted from a string.

strip(), stripLeading(), stripTrailing(): Trims white spaces from the string.

repeat(int count): Repeats the string the specified number of times.

Example:

```
String text = "  Hello World  ";

System.out.println(text.isBlank());  // false

System.out.println(text.strip());    // "Hello World"

System.out.println(text.repeat(2));  // "  Hello World   Hello World  "
```

New Files Methods

Files.readString(): Reads all content from a file as a string.

Files.writeString(): Writes a string to a file.

Example:

```
Path path = Paths.get("sample.txt");

Files.writeString(path, "Hello, World!");

String content = Files.readString(path);

System.out.println(content);  // "Hello, World!"
```

Optional.isEmpty()

A new isEmpty() method was added to the Optional class to complement isPresent().

Example:

```
Optional<String> opt = Optional.of("Hello");

System.out.println(opt.isEmpty());  // false
```

Java 12:

- In Java 12, the Files.mismatch(Path, Path) method was introduced as part of the java.nio.file.Files class.
- This method is used to find the first position at which two files differ.
- It can be useful in scenarios where we want to compare two files and determine where the difference starts, rather than simply checking if they are identical.

Key Details of Files.mismatch in Java 12:

Method Signature:
public static long mismatch(Path path, Path path2) throws IOException

- Parameters:
  - path: The path to the first file.
  - path2: The path to the second file.
- Returns:
  - The position of the first byte that differs between the two files.
  - If the files are identical, it returns -1.
  - If one file is a prefix of the other, it returns the size of the smaller file.
- Exceptions:
  - IOException: If an I/O error occurs reading from the files.

Java 14:

Switch Expressions:

- Description: Switch expressions, which were introduced as a preview in earlier versions, became a standard feature in Java 14. This feature allows the switch statement to return a value, making it more versatile.

Syntax:

```
int numLetters = switch (day) {

    case MONDAY, FRIDAY, SUNDAY -> 6;

    case TUESDAY -> 7;

    case THURSDAY, SATURDAY -> 8;

    case WEDNESDAY -> 9;

};
```

- **Benefits:**
  - Reduces boilerplate code and potential errors with more concise syntax.
  - Supports modern programming patterns by making switch more powerful.

When a case block requires multiple statements, `yield` makes it clear which value should be returned from the `switch` expression.

Java 15:

Text Blocks (Standard Feature)

- What It Is: Text blocks, first previewed in Java 13 and 14, became a standard feature in Java 15. They allow for multi-line string literals, making it easier to write readable code when dealing with large blocks of text (e.g., HTML, JSON, SQL).

Syntax:

```
String json = """
    {
        "name": "John",
        "age": 30,
        "city": "New York"
    }
    """;
```

- Benefits: Text blocks simplify the inclusion of multi-line strings, reduce the need for escape sequences, and improve the readability of code.

Java 16:

Records:

Records are a special kind of class in Java that are designed to be a concise way to model immutable data. They automatically generate constructors, equals(), hashCode(), toString(), and getter methods.

Syntax:

```
public record Point(int x, int y) {}
```

Benefits: Records reduce boilerplate code, making it easier to create simple data carrier classes without writing all the typical methods manually.

Example:

```
Point p = new Point(10, 20);

System.out.println(p.x()); // Output: 10

System.out.println(p);     // Output: Point[x=10, y=20]
```

Pattern Matching for instanceof (Standard Feature)

- What It Is: This feature, introduced in preview in earlier versions, became a standard feature in Java 16. It allows us to simplify type checks and casting in instanceof expressions.

Syntax:

```
Object obj = "Hello, World!";

if (obj instanceof String s) {

   System.out.println(s.toUpperCase()); // No need for explicit cast

}
```

- **Benefits:** Simplifies code and reduces the need for explicit type casting after an instanceof check.

Stream toList() Method

- What It Is: A new toList() method was added to the Stream interface to collect elements into an immutable list.

Syntax:

```
List<String> list = Stream.of("a", "b", "c").toList();
```

- Benefits: Simplifies collecting stream elements into an immutable list, making code more concise.

Java 17:

Sealed Classes (Standard Feature)

- **What It Is:** Sealed classes, introduced as a preview feature in Java 15 and 16, became a standard feature in Java 17. Sealed classes restrict which classes can extend or implement them.

Syntax:

```
public abstract sealed class Shape

    permits Circle, Square, Rectangle {

        // class body

        }
```

- Benefits: Provides greater control over class hierarchies and helps prevent unauthorized subclasses.