**Java 8 : (2014)**

1. **Lambda Expression**
2. **Functional Interface**
3. **Stream API**
4. **Default and static methods in Interface**
5. **Optional Class**
6. **Collectors**
7. **Nashorn Java**
8. **CompletableFuture(in java.util.concurrent) (Thread Notes)**

**Lambda Function / Expression :**

A Lambda Expression is an **anonymous** **(nameless) function**. This means it is a function that does not have a name, return type, or access modifiers.

The main objective of Lambda Expression is to bring the benefits of functional programming into Java

Advantages of Lambda Expression

* We can reduce the length of the code to provide readability.
* We can provide Lambda expression in the place of an object.
* We can pass lambda expression as an argument to methods

**Optional :**

Optional<T> is a **container object** which may or may not contain a **non-null value** of type T.

**Usage :**

* **Avoids null checks**
* **Prevents NullPointerException**
* Makes code more readable and expressive. Encourages functional-style programming.

**Common Methods in Optional**

1. Optional.of(value) → wraps a non-null value
2. Optional.empty() → creates an empty Optional
3. Optional.ofNullable(value) → allows null or non-null values

In **Spring Data JPA**, repository methods often return Optional<T> to handle missing data:

**Optional<User> user = userRepository.findById(1L);**

**Functional Interface:**

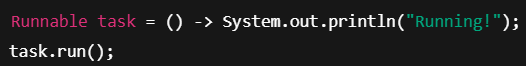
an interface has **only Single Abstract** **Method (SAM)**, it can have any number of default and static method is called functional interface.

1. use annotation **@FunctionalInterface** (**optional**) for custom functional interface, It’s like a validation that in future if anyone try to modify this interface then he can’t. JVM will throw compile time error if rule is violated
2. It is used to provide implementation for lambda expression/function
3. To write **cleaner, more readable code** using **lambdas** instead of anonymous classes.

**Types of Functional interfaces :**

1. **Runnable :**

Method: void run(), No input, No output



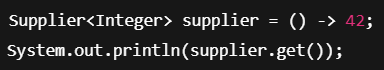
1. **Callable:**

Method: V call() throws Exception, No input, returns V.



1. **Supplier<T>:**

Method: T get(), no input, returns T



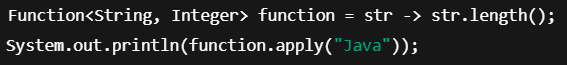
1. **Consumer<T>:**

Method: void accept(T t), input T, no return type (void)



1. **Function<T, R>**

Method: R apply(T t), input T, return R.



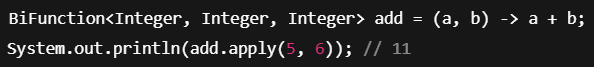
1. **Predicate<T>**

Method: Boolean test(T t), input T, return Boolean



1. **BiFunction<T, U, R>**

Method: R apply(T t, U u), inputs T, U, return R

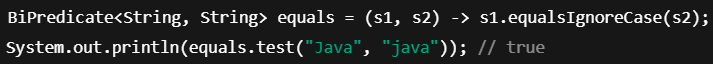


1. **BiConsumer Functional Interface :**

BiConsumer accept(T t, U u), two input arguments of types T and U and performs an operation on them and returns no result.

1. **BiPredicate<T, U>**

Method: Boolean test(T t, U u), inputs T, U, returns Boolean



1. **Comparator<T>**

**Method: int compare(T o1, T o2), inputs, t, returns int**

(Result: <0 if o1 < o2, 0 if o1 == o2, >0 if o1 > o2)



**Default & Static Method in Interface:**

**Default Method**

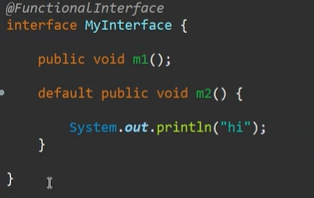
A default method is a method of Java Interface that has default implementation using default keyword and can be overridden.

**When two interfaces have same default method then we can use –**

A.**super**.m1(); //InterfaceName.super.methodName();

**Use:**

1. **Backward Compatibility –** you can add new behavior without affecting classes that already implement the interface.
2. **Provide a Common Implementation**

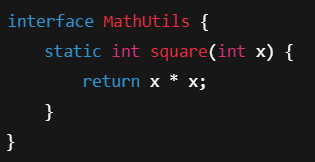


**Static Methods in Interfaces :**

* introduced in **Java 8**.
* Defined using the static keyword.
* Can only be called using the **interface name**, not by the implementing class.
* Static methods in interfaces **belong to the interface**, not to implementing classes.
* Can’t be overridden.

**Use:**

It is used as a utility or helper methods related to the interface.





**Marker Interface :**

A **marker interface** in Java is an **interface with no methods or fields** — it's empty.

**Purpose**: It is used to **"mark"** or **tag** a class so that **JVM or frameworks** can recognize it and give it **special behavior**.

Examples :

* Serializable Marks a class as serializable (for I/O)
* Cloneable Allows objects to be cloned using .clone()
* Remote Used in RMI to mark remote objects

**Stream API:**

**Stream :**

It is representation of sequence of objects. It’s not the original data. It does not store actual objects, operates on the data **lazily**. Used to perform various operation like: filtering, mapping sorting etc.

Streams don’t directly support **reverse iteration** or **index-based access**

There are two types of operation:

1. **Intermediate operations –**

(to modify, it will return only stream), Includes methods like

**filter(Predicate<T> predicate)** Keep only elements that match

**map(Function<T, R> mapper)** Transform each element

**sorted() (for wrapper class) / sorted(Comparator<T>) (for custom objects)** Sort elements

**distinct()** Remove duplicates

1. **Terminal operation –**

(used to collect and return an object and to end the stream pipeline). Includes methods like

**toList()** Collects stream elements into a list. Return List<T>. Introduced in Java 16, it **returns immutable** list and throw exception (UnsupprotedOperationException) if we try to modify that list, more readable.

**collect()** Performs a mutable reduction using a collector. Return depends on collector (e.g., List<T>, Set<T>, Map<K, V>)

**count()** Counts the number of elements in the stream. Return long

**forEach()** Performs an action on each element. Return void

1. **Short-Circuiting Terminal Operations**

Short-circuiting terminal operations are stream operations that **do not necessarily process all elements** of the stream.  
They **terminate the processing early** as soon as the result is determined, which improves performance.

These operations are particularly useful when working with **large datasets**, as they can **stop execution once the desired condition is met**.

Methods like :

**anyMatch(Predicate)**  Returns true as soon as **any** element matches the condition.

**allMatch(Predicate)**  Returns false as soon as it finds **any element that doesn’t match**.

**noneMatch(Predicate)** Returns false as soon as it finds **any matching element**

**findFirst()** Returns the **first element**, then stops processing further.

**findAny()**  Returns **any element** (useful for parallel streams). **Optional<T>**

**map()**

Transforms each element individually.

Returns a Stream of transformed elements.

One-to-one mapping structure.

When we want to apply function to each element and keep the structure flat.

Internally it uses functional interface

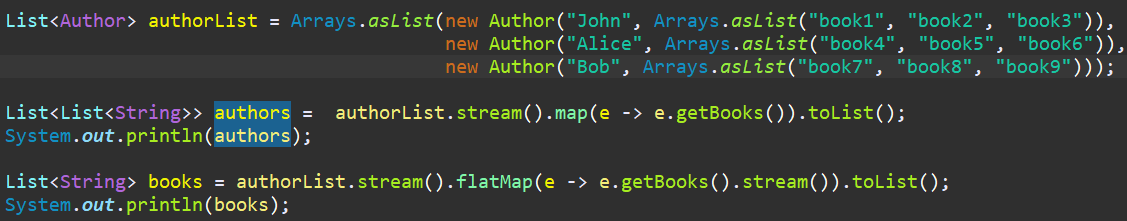
**flatMap() :**

flatMap is a method in Java Streams used to flatten nested structures (like List<List<T>>) and then map each element.

Flattens a stream of streams into a single stream.

Returns a flattened stream after applying a transformation.

One-to-many mapping.



Filter() -> internally it uses predicate functional interface

**Collect(Collectors.toList())** : Introduces in Java 8, **returns mutable list** we can add or remove from that list, means it is more flexible, internally uses **ArrayList**, slightly more verbose.

**Collectors (Interface):**

It is a **utility class** from the java.util.stream package that provides **reduction operations** to collect result of a stream into a collection, String, Map etc.

collect() is a **terminal operation** on the stream.

Collectors.toList() is a **collector** that tells Java to collect elements into a List

Types of Stream

**Sequential stream :**

* Elements are processed one after another in a single thread
* Preserver the order of elements.
* Suitable for small to medium datasets or tasks that must maintain order.
* No thread safety.

**Parallel stream:**

* Internally splits the stream into multiple substreams and processes them concurrently using the ForkJoinPool.
* May not preserve order unless you use forEachOrdered().
* Can improve performance for large datasets and CPU-intensive tasks – but not always due to overhead.
* Provide thread safety.

**Nashorn Engine :**

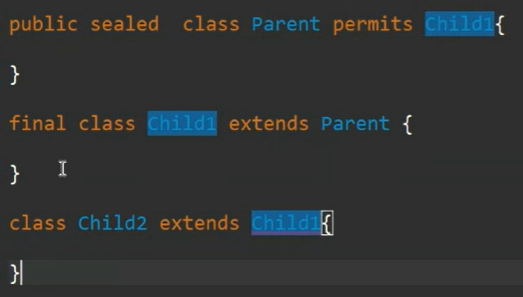
Enables execution of JavaScript code from within Java applications.

**Java 17 :**

1. **Sealed class :**

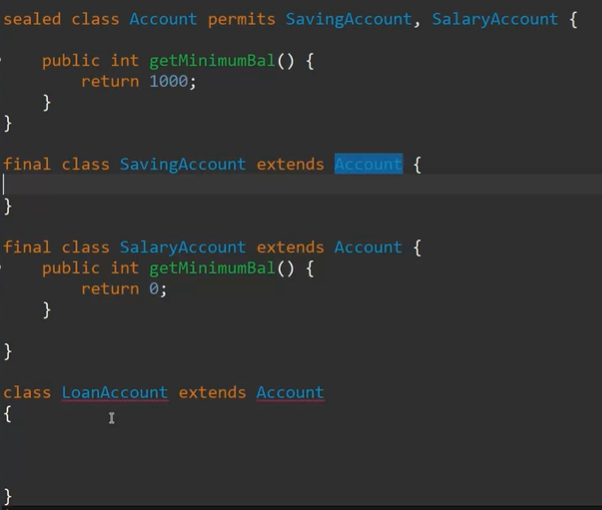
Here we use sealed and permit keyword where sealed is use to seal a particular **class/interface** and permit is used to permit which other classes can extend this class.

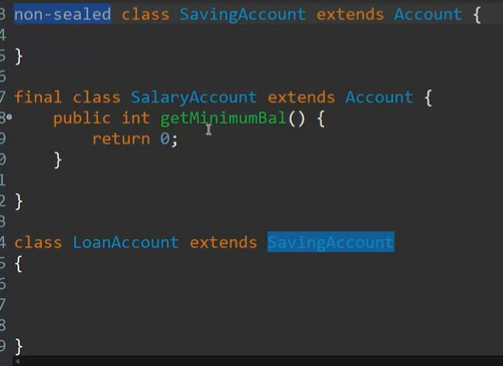
The class that are permitted need to be made as final class i.e. no further implementation of permitted class. Otherwise, it will break the rule of permit



Here we cannot extend Child1 class.

E.g.





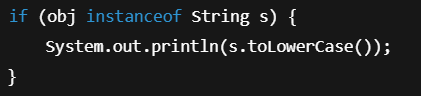
Use Aggregation/Association or non-sealed keyword if you want any functionality of sealed class to be inherited, but there will be partial control of sealed class.

1. **Pattern Matching :**

**No more manual casting, shorter, safer, cleaner code. No ClassCastException after instanceof checks.**

Improves switch statements by :

* Simplifying type checking (No more instanceof + casting)
* Allowing complex data structures (Records, Enums, etc)
* Cannot use switch on generic types ( List<T>, Set<T>)/ non primitives (int, char), only works for wrapper classes (Integer, String etc)
* Need to enable it with –enable-preview



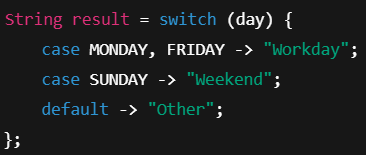
1. **Floating-Point Enhancements in Java 17 (JEP 306) – *In Short*:**

Java 17 **restores strict floating-point semantics** by **removing the "relaxed" floating-point behavior** that was allowed on some older platforms (like x86 with extended precision).

* **All floating-point operations (float, double)** now strictly follow the **IEEE 754 standard**.
* Ensures **predictable and consistent results** across all platforms.
* Ensures float and double give the **same results on all hardware**.
* All operations strictly follow 64-bit double precision rules.

1. **Switch Enhancements**

Switch becomes more powerful and expressive with yield, multiple labels

****

**Questions :**

**Java 8:**

**What is lambda expression?**

The method having no name, no return type and no any access modifiers is known lambda expression.

**What is the benefit of using lambda expressions?**

We can use lambda expression to represent an instance of a functional interface in a more compact and readable way.

Lambda expressions make Java more **concise, expressive, and powerful** by enabling a cleaner way to represent behavior.

**What is the syntax of a lambda expression?**

(parameters) -> { body }

The method having no name, no return type and no any access modifiers is known lambda expression.

**What is a functional interface?**

An interface having only single abstract method (SAM) is known as functional interface. and that abstract method is called as functional method.

We can add @FunctionalInterface annotation over it/on top of it to declare as functional interface.

**Why is a functional interface required for a lambda expression?**

A **functional interface** is **required** for a lambda expression in Java because **lambda expressions don’t exist independently** — they’re just **a shorter way to implement the single abstract method (SAM)** of a functional interface.

**How will you call a default method of an interface in a class?**

* **Use it directly** like any other method.
* **Override it** if you want to change the behavior.
* **Call the original default implementation using a special syntax .**

**InterfaceName.super.methodName();**

**Is it possible to define our own Functional Interface? What is @FunctionalInterface?**

Yes, absolutely! In Java, **you can define your own Functional Interface**, and you can mark it using the @FunctionalInterface annotation.

**Describe some of the functional interfaces in the standard library.**

**Runnable, Callable<V>, Comparator<T>**

* Predicate<T> Tests a condition on an input and returns true/false
* Function<T, R> Converts or maps input T to output R
* Consumer<T> Performs an operation on input without returning anything
* Supplier<T> Supplies a value (no input, just output)
* UnaryOperator<T> A specialization of Function where input and output are same type
* BinaryOperator<T> Like Function, but takes two same-type inputs and returns one result

**Can we perform intermediate operations without terminal operations in stream?**

Yes, **you can perform intermediate operations without terminal operations in Java Streams**, but the key point is:

**Intermediate operations are lazy** — they are not executed until a terminal operation is invoked.

Stream Vs Collection

Stream vs parallel stream

Map vs flatmap