**Why was Java 8(1.8) introduced?**

1. Java 8 (also known as Java 1.8) was introduced on March 18, 2014

***Main Features of JAVA 8:-***

1. **Lambda Expression:-**

**🡪It is an anonymous function. (No name, no modifiers, no return type)**

* (->) symbol is used to write Lambda expression
* () -> System.out.println(“Hello”); <- way of writing lambda expression
* If more than one line then {} are mandatory
* (a,b) -> System.out.println(a+b); <- parameterized method
* (int n) -> {return n;}

(int n) -> n\*n;

(n) ->n\*n;

n -> n\*n;

* s -> s.length(); Calculate length of String s
* Without curly braces we cannot use return keyword
* (Limitations)We can use Lambda expressions only and only with Functional Interfaces only.
* In curly braces each n every statement should end with “ ; “.

1. **Functional Interface:-** Interfaces **which have only single abstract method but can have multiple default and static methods** is called as Functional Interface

**Eg:-** Callable call(), Runnable run(),Comparator compare(),Comparable compareTo()

Once we write Lambda Interface, to call that expression we should go for Functional Interface

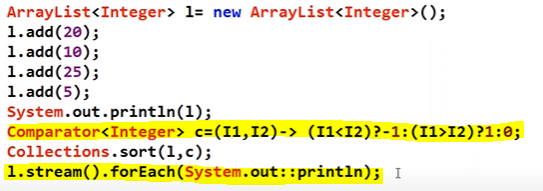
**What is the task of Functional Interface?**

* Functional interface is used to invoke lambda expression.
* @FunctionalInterface annotation is written above interface to imply that it is FunctionalInterface. But it is not mandatory, even if we don’t give it , still it will be FI.

**For eg:-**

If Functional interface is going to have multiple abstract methods then we cannot tell lambda expression explicitly which method to call, Thus it is important for Lambda expression to have **FunctionalInterface** only so it can call that method without its name as it is only one.

Comparator Interface has only one abstract method 🡪 **compare(Obj o1, Obj o2)**



Here, comparator by default calls its compare method, when used with Lambda Expression.

**What is more powerful anonymous inner class or Lambda Expression?**

**🡪**Annonymous inner class is more powerful because it can have multiple abstract methods, whose implementation can be provided without implementing the interface and just by making interface reference.

**IQ) Why Were Lambda Expressions Introduced When We Already Had Anonymous Inner Classes?**

Before Java 8, Anonymous Inner Classes were commonly used to implement functional interfaces (interfaces with a single abstract method). However, they had several limitations that made them verbose and harder to read.

**🔹 Problems with Anonymous Inner Classes**

**Too Much Boilerplate Code**

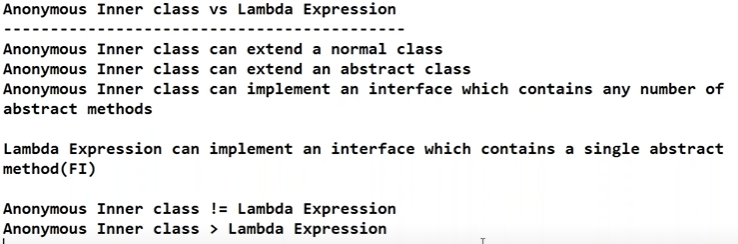
* Even for simple tasks, we had to write extra code like new InterfaceName(), {} braces, and @Override methods.

**Readability Issues**

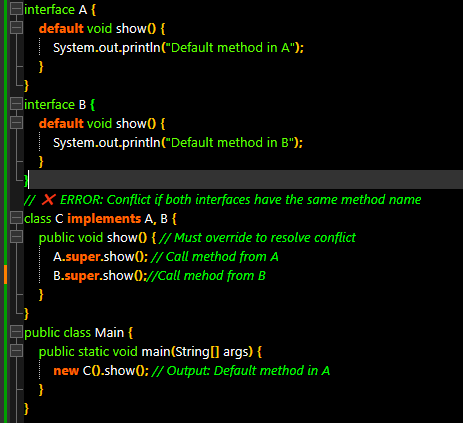
* Large and nested anonymous classes reduce readability.

**No Direct Representation of Behavior**

* Anonymous inner classes define both the implementation and object at once, while lambda expressions only represent the behavior.



1. **Default() in interface:-**

* Default() is also called as **virtual extension method, defender method**
* Without affecting implementation classes if we want to add new method to the interface then we can use default() methods.
* Java 8 introduced **default methods** (methods with a body inside interfaces) to provide a default implementation **without breaking existing code**.
* We can use default() to share common functionality among all classes without affecting existing code.
* It is called default method because it already has default implementation for all the classes which implement that interface.
* Default methods can be overrided in the implementing class but it is not good practise because it would be of no meaning if we are overriding default method.
* Before Java 8, multiple inheritance was **not possible** because Java does not support multiple class inheritance. We can do it here by using super keyword.
* 

**🡪Static() in interface:-**

* We can write main method inside interface from java (v1.8)
* Before Java 8, utility methods had to be placed in **separate helper classes** (e.g., Collections, Arrays). **(Utility Methods Inside Interfaces (Without Helper Classes))**  
  🔹 With static methods inside interfaces, we can **group related utility methods within the interface itself**.
* Before Java 8, interface utility methods were scattered in different classes.  
  🔹 Now, static methods in interfaces **keep related logic inside the interface itself**, making code **more modular and readable**.
* **They cannot be overridden** but can be accessed via InterfaceName.methodName()
* Static methods in interfaces are useful for creating **factory methods** that return instances of a class.

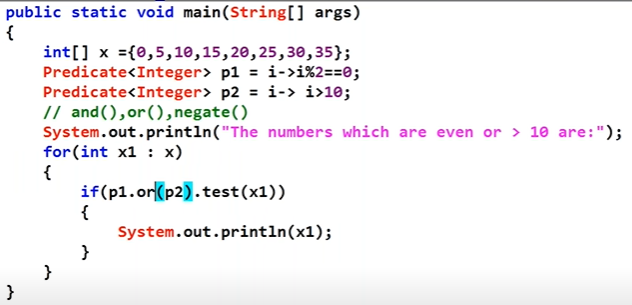


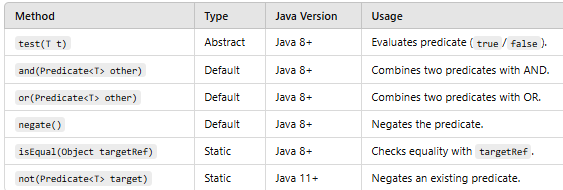
**Predefined Functional Interfaces:-**

1. **Predicate(I) :- public abstract boolean test(T t)**

* **Predicate** is a functional interface (java.util.function.Predicate<T>) that represents a boolean-valued function. It has a single method, **public abstract boolean test(T t),** which evaluates a condition and **returns true or false**.
* Predicates are commonly used for filtering in the Stream API and can be combined using methods like **and(), or(),** and **negate().**
* Example usage includes checking conditions on **collections, filtering lists, and writing cleaner, functional-style code**.
* **Predicate Joins:-** We can combine multiple predicates to check complex conditions.

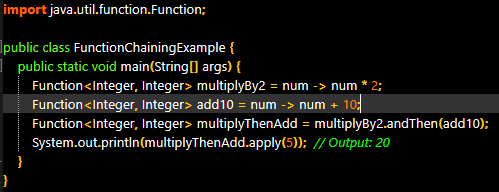
p1.**and**(p2).test(20) , p1.**negate**() , p1.**or**(p2).test(20)



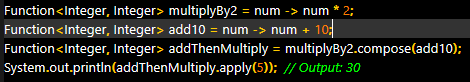
* **Applied in e-commerce, CRM systems, and social media apps to filter users based on status.**
* **Used in form validation for login, registration, and contact forms**
* 

1. **Function<T,R> (I) :- public abstract R apply(T t)**

* T🡪Input Type , R🡪 Return Type
* The Function<T, R> interface in Java 8 represents a function that **accepts an input of type T and returns a result of type R**. It is commonly used for **data transformation** (e.g., converting a String to an Integer or applying a calculation).
* Function<T, R> is **used for transformations** (e.g., String → Integer).
* Supports **method chaining** using andThen() and compose().
* **Widely used in Streams** (map()) for data processing.
* Function chaining is also done by function interface
* **andThen()** – Executes first function, then second function

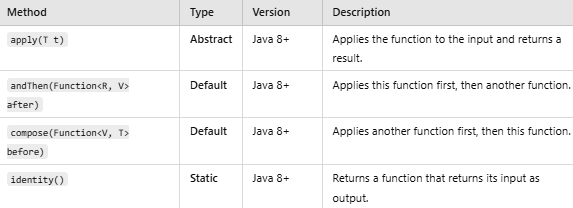


* **compose()** – Executes second function first, then first function

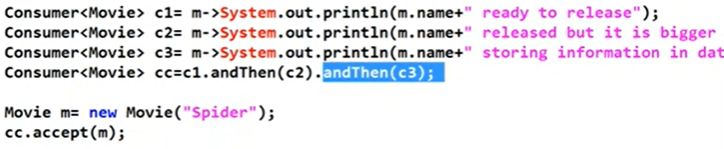
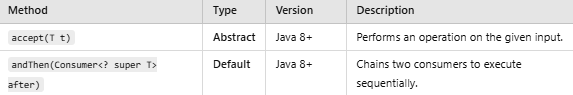


* The Function<T, R> interface is **widely used in real-world projects** for:

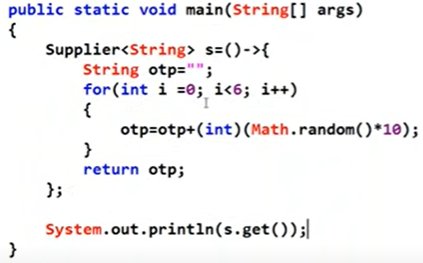
✔ **Data transformation (DTO conversion in APIs)**  
✔ **Payroll and salary calculations**  
✔ **Image processing and resizing**  
✔ **Dynamic pricing in e-commerce and ride-sharing**



1. **Consumer<T> (I):-** **public abstract void accept(T t)**

* Consumer chaining is possible by **andThen()** method.
* 
* **Takes an argument but does NOT return anything.**  
  ✔️ **Used for operations like logging, printing, updating records, etc.**  
  ✔️ **Supports function chaining using andThen()**  
  ✔️ **Often used in Java Streams (forEach()).**
* Used in **Banking, E-commerce, and FinTech applications** for transaction alerts.
* Used in **Spring Boot**, **Microservices**, and **Middleware** to log HTTP requests and responses.
* Used in **FinTech, Online Payment Gateways, and Digital Wallets** (e.g., PayPal, Stripe).
* 

1. **Supplier (I) :- public abstract R get() 🡪R is return type**

* 
* The **Supplier<T>** interface is a **functional interface** in Java introduced in **Java 8**, which is part of the java.util.function package. It represents a **supplier of results**, meaning it **does not take any input but returns a result**.
* The Supplier<T> interface is widely used in **real-world applications** where a value needs to be **dynamically generated or fetched** **without requiring input**. It is commonly used in **lazy initialization, caching, random data generation, database connections, and microservices**.
* Banking & Finance 🡪Generating transaction ID’s and random OTP’s.
* Cloud & Microservices 🡪 Generating API tokens on demand.
* ✅ **Supplier<T> is useful when you need to want a value dynamically without input!**

**4(b)🡪** **BiPredicate<T1,T2> 🡪Accepts two inputs T1,T2**

* Normal predicate can take only one input argument and perform some condtional checks. But sometimes programming requirement is we have to **take 2 inputs args** and perform some conditional checks, for this requirement we should go for **BiPredicate**
* **Interface BiPredicate<T1,T2> {**

**Public boolean test(T1 t1,T2 t2)**

**}**

* Can be used to check sum of 2 given integers is even or not by using BiPredicate.

**Similarly BiFunction -> accepts two Inputs and returns one Output**

**, BiConsumer -> accepts two inputs and returns void**

**Predicate**

**✔** Takes some input, performs a conditional check, and returns a boolean value.

* **boolean test(T t);**

**Function**

✔ **Takes some input**, performs an operation, and **returns a result** (which is not necessarily a boolean).

* **R apply(T t);**

**Consumer**

**✔** Accepts some input, performs the required operation, and does not return anything.

* **void accept(T t);**

**Supplier**

**✔** Does not take any input and returns a value.

* **T get():**

**Other FunctionalInterfaces:-**

1. **DoubleFunction <double T, R>** 🡪**input type:** double **return type:**Any result
2. **IntFunction <int T, R>** 🡪 **input type:** int **return type:**Any result
3. **LongFunction <int T, R>** 🡪 **input type:** long **return type:**Any result
4. **DoubleToIntFunction<double T, int R> 🡪 input type:** double **return type:**int **applyAsInt()**
5. **DoubleToLongFunction<double T, long R> 🡪 input type:** double **return type:**long

[**https://docs.oracle.com/javase/8/docs/api/**](https://docs.oracle.com/javase/8/docs/api/)🡪**Other inerfaces of java.util.function package**

1. **Method() & constructor reference:-**

* **Method()🡪** In case of method reference different return types are allowed.
* But only restriction is arguments type should be same as argument of SAM of interface.
* If the implementation is already available then we should use Lambda Expression
* If it is available then to resuse we can go for method() reference.
* A method reference is a shortcut that lets us refer to an existing method instead of writing a full lambda expression.

✅ **Why use this?**

* Less code
* More readability
* No need to explicitly define arguments
* **Constructor reference**🡪 A **constructor reference** is a shortcut that lets us call a constructor without writing a full lambda expression.

✅ **Why use this?**

* **Simplifies object creation**
* **Removes redundant lambda code**

**Why Were These Introduced? (Industry Need)**

* Before Java 8, code was often **verbose**. Java introduced **lambdas** to simplify it, but even with lambdas, some expressions were still unnecessary.  
  Method and constructor references **further reduce complexity** and improve readability.

**Real-World Applications of Method & Constructor References**

✅ **1. Logging Systems**

* Logger logger = System.out::println;
* Used in applications to log messages efficiently without writing extra lambda code.

✅ **2. Sorting Data in Collections**

* names.sort(String::compareToIgnoreCase);
* Used in e-commerce (sorting products by name) and finance (sorting transactions).

✅ **3. Factory Pattern & Object Creation**

* Supplier<MyService> serviceSupplier = MyService::new;
* Used in **Spring Framework** for creating objects dynamically.

✅ **4. Stream API & Parallel Processing**

* list.forEach(System.out::println);
* Used in **big data** processing (e.g., log file analysis, recommendation systems).

✅ **5. Database Entity Creation (ORM - Hibernate)**

* Function<String, User> userCreator = User::new;
* Used in **Hibernate** for dynamically creating objects from database queries.

**Method references work only with functional interfaces** (interfaces with a **SAM**).  
✅ **Default and static methods don't count as abstract methods**, so method references still work.  
⛔ **Won’t work with interfaces that have multiple abstract methods** (non-functional interfaces).

**Constructor references work only with functional interfaces** (i.e., interfaces with a **single abstract method**).  
✅ **Supports different types of constructors (no-arg, parameterized, overloaded).**  
⛔ **Won’t work with non-functional interfaces (multiple abstract methods).**

1. **Streams:-**

* If we want to process objects from Collection then we should use -Stream.
* Stream is sequence / series of elements /object.
* If we want to get Stream then **Stream s =collection.stream();**
* It is present in **java.util.stream** package**.**
* **By using Stream we can write very concise code.**

**.filter(Predicate)** 🡪 Use this when filtering data based on some condition

**Eg:- Students who scored marks above 60.**

**.map(Function)** 🡪 Use this when some processing needed on collection object.

**Eg:- Add 5 grace marks for all students having marks =60.**

* **Stream concept can be used with arrays, collection, any group of objects.**

**Foreach-Loop was introduced in jdk1.5**

**stream()🡪**Take all collection objects into stream

**filter()🡪**Filter objects of stream based on some criteria

**map() 🡪**Provide some operation on elements of stream.

**Collect()🡪** collect elements of stream and store them in Collection object(Collectors)

**Count()🡪** Count the elements in stream

**Sorted()🡪** Sorts all elements of Stream in DNSO

**Sorted((i1,i2) -> ((i1<i2)?1:(i1>i2)?-1:0))** 🡪 Reverse sorting of elements(Descending)

Alternative-**Sorted((i1,i2) -> -i1.compareTo(i2)) 🡪** Reverse sorting of elements (Descending)

**Min()** 🡪First element inside the stream **\* We need to pass comparator compulsorily**

**Max() 🡪**Last element of the stream **\* We need to pass comparator compulsorily**

**Min((i1,i2) -> i1.compareTo(i2)) vice-versa**

**Foreach() 🡪**Used for iteration works same as foreach loop

l.Stream().foreach(System.out::println) 🡪Will print all elements of List l

**toArray()** 🡪Used to convert stream of elements to Array

toArray(Interger[]::new) 🡪 Returns integer array object.

**✅ Key Properties of Streams:**

1. **Does Not Store Data - It works on data from a Collection, Array, or I/O channel but does not hold data itself.**
2. **Lazily Constructed - Intermediate operations are lazy, they don’t execute until a terminal operation is invoked.**
3. **Functional - You can use lambdas to process data.**
4. **Can be Parallel - You can parallelize a stream easily with .parallelStream().**

**✅ Why use Stream?**

* **Less boilerplate code**
* **More readable and expressive**
* **Supports parallel processing**
* **Reduces mutable state**
* **Helps you write declarative code instead of imperative code**

[**https://chatgpt.com/c/67e4d3b0-8454-8004-9118-c5e208b87248**](https://chatgpt.com/c/67e4d3b0-8454-8004-9118-c5e208b87248)

1. **Date & Time API:-**