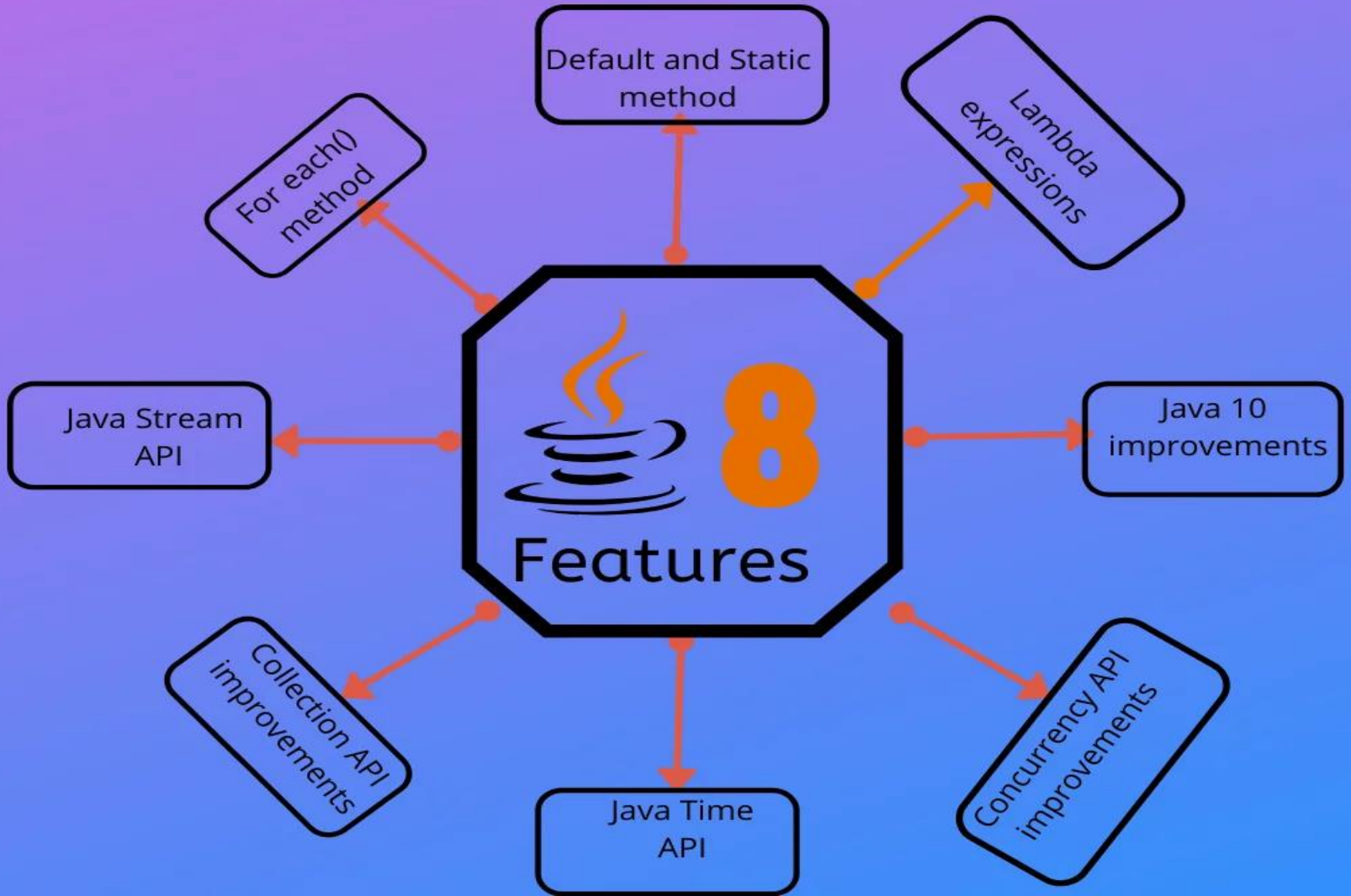
# FEATURES OF JAVA 8


Default and Static method
For each() method
Lambda expressions
Java Stream API
8
Features
Java 10 improvements
Collection API improvements
Java Time API
Concurrency API improvements

Java 7: July 28, 2011

Java 8 : 18th March, 2014

Java 9 : 21st Sep, 2017

Java 10 : 20th March, 2018

Java 11 : 25th Sep 2018

Java 12 : 19th March, 2019

Java 19 20 sep,2022

Java 20 21 Mar, 2023

Java 13 : 17th Sep, 2019

Java 14 : 17th March, 2020

Java 15 : 15th Sep, 2020

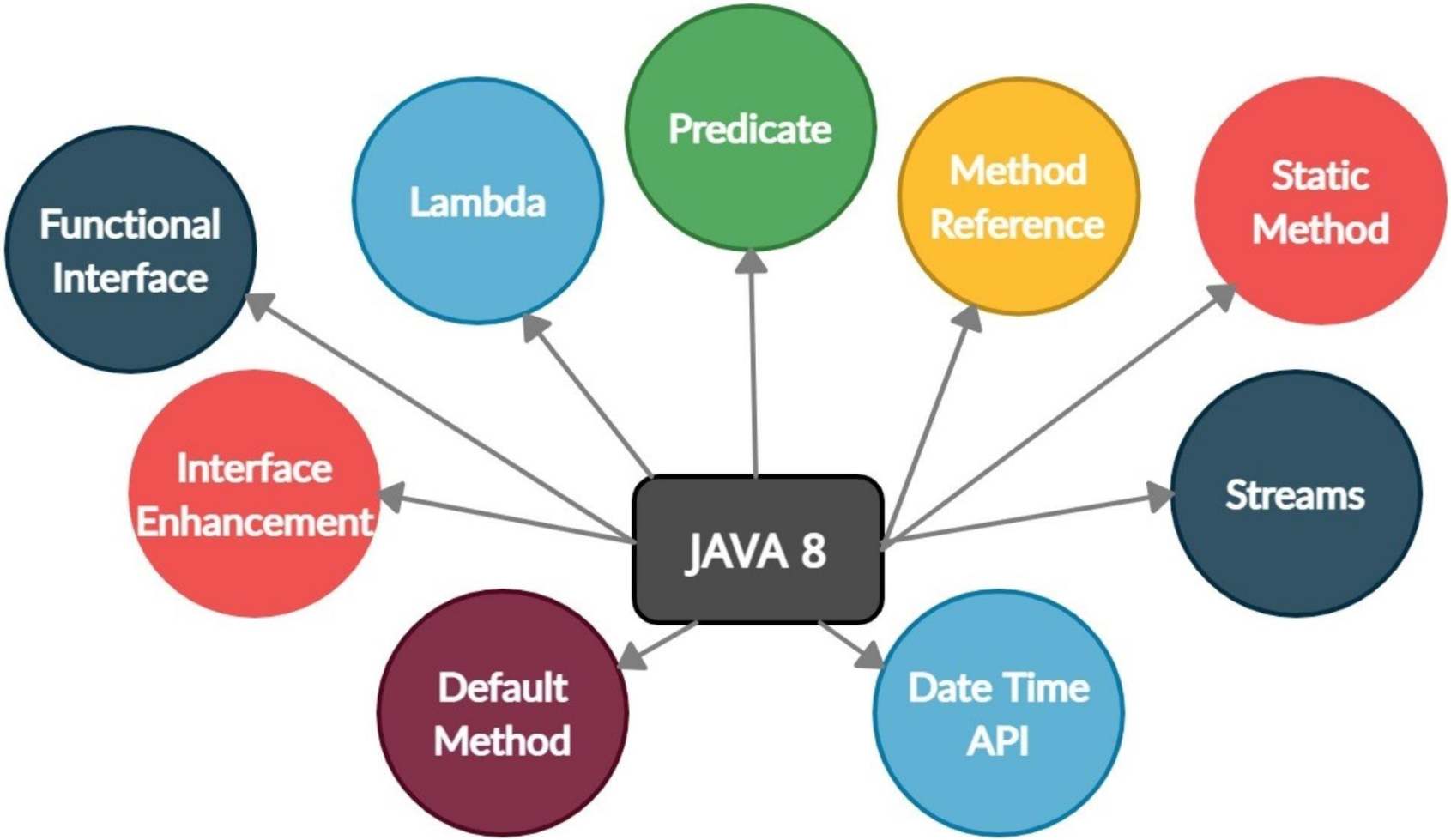
Java 16: 16th March 2021

Java 17 : 14 September 2021

Java 18 : 22 March, 2022

Java 21 19th Sep, 2023

1. Lamda Expression
2. Functional Interface
3. Default Methods and static methods
4. Predefined functional interface
    a. Predicate
    b. Functional
    c. Consumer
    d. Supplier
5. Double Colon Operator (::)
    Reference to static method
    Reference to instance method
    Reference to a Constructor
6. Streams API
7. Date and Time API
8. Optional Class
9. Nashron Java Script Engine


Functional Interface
Lambda
Predicate
Method Reference
Static Method
Interface Enhancement
JAVA 8
Streams
Default Method
Date Time API

# Interface Enhancements

1. Default Method

2. Static Method

# 1. Default Method

```
interface TestInterface
{
     public void square(int a);   // abstract method

      default void show() // default method
      {
              System.out.println("Default Method Executed");
       }
}
class TestClass implements TestInterface
{
   // implementation of square abstract method
   public void square(int a)
   {
         System.out.println(a*a);
   }
```

# Default Method

```
    public static void main(String args[])
    {
        TestClass d = new TestClass();

         d.square(4);   // default method executed


          d.show();
    }
}
```

# Advantages of  Default Method

Backward Compatibility:-

Means without effecting the implementation of classes, we can add new functionality to the interface.

# Static Method

- Java 8 also support static method  inside interface.
- Static method by default are not available to implementation classes.
- We can not call using class object name, but with the help of interface name we can call it.

# Static Method

```
interface  Engine
{
        public static void display()
        {
              System.out.println("This is java 8 static method ");
        }
}
public class Train implements Engine
{
      public static void main(String args[])
      {
        Engine.display();
      }
}
```

Output:-

---

This is java 8 static method

# Functional Interface

Functional interface contains only one abstract method, such type of interfaces are called functional interfaces

Example:-

1. **Runnable :-** only run() method

2. **Comparable :** - only compareTo() method

But we can write any number of default and static methods inside of functional interface.

- A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit.

- From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default & static methods.

- ***Runnable***, ***ActionListener***, ***Comparable*** are some of the examples of functional interfaces.

- Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

```
interface Eatable
{
   abstract public void eat();
}
class TestAnnonymousInner1
{
        public static void main(String args[])
        {
            Eatable e=new Eatable()
          {
                public void eat()
                {
                    System.out.println(“Nice fruits");
                }
           };
            e.eat();
        }
}
```

# Functional Interface

```
@FunctionalInterface
interface  Engine
{
        abstract public void show();  // Contains exact abstract method
        public static void display()
        {
          System.out.println("This is java 8 static method ");
        }
}
```

# Java 8

## Lambda Expressions

# Content to Cover

- What is Lamda?
- Benefits of Lamda
- How to write Lamda
- Important Rules Of Lamda
- Functional Interface
- How lamda is used to implement Functional Interface
- Lamda Examples, creating thread using Lamda

# What is Lambda Expression

Lambda is anonymous Function

- No Name
- No Modifier
- No Return Type

# Benefit of Lamda

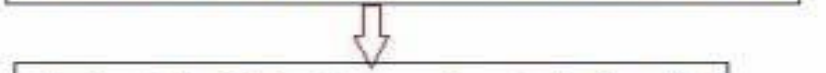
- Reduce the lines of code.
- Sequential and parallel execution support by passing behavior as an argument in methods.
- To call APIs very effectively
- To write more Readable, maintainable and concise code.

# How to write a Lambda Expression Code

```
private void add(int x, int y) {
        System.out.println(x+y);
}
```

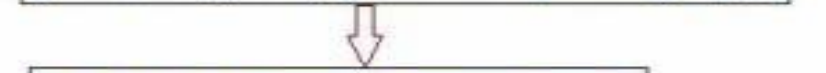
```
void add(int x, int y) {System.out.println(x+y); }
```
//remove access modifiers

```
add(int x, int y) {System.out.println(x+y); }
```
//remove return type



```
(int x, int y) {System.out.println(x+y); }
```
//remove method name

```
(int x, int y) -> {System.out.println(x+y); }
```
//insert symbol '->'

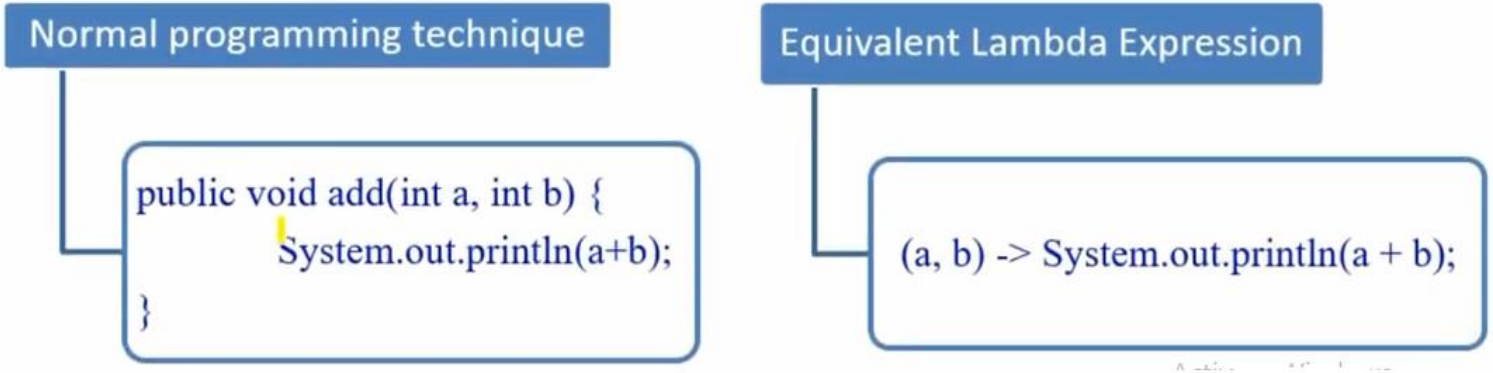


```
(x, y) -> System.out.println(x+y);
```
//remove parameter types & parenthesis

# Q) What is lambda expression?

- Lambda expression is an anonymous function ( without name, return type and access modifier and having one lambda ( → ) symbol )

- Eg :



# Lambda Syntax

- No arguments:

```
() -> System.out.println("Hello")
```

- One argument:

```
s -> System.out.println(s)
```

- Two arguments:

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

- With explicit argument types:

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

- Multiple statements:

```
(x, y) -> {
    System.out.println(x);
    System.out.println(y);
    return (x + y);
}
```

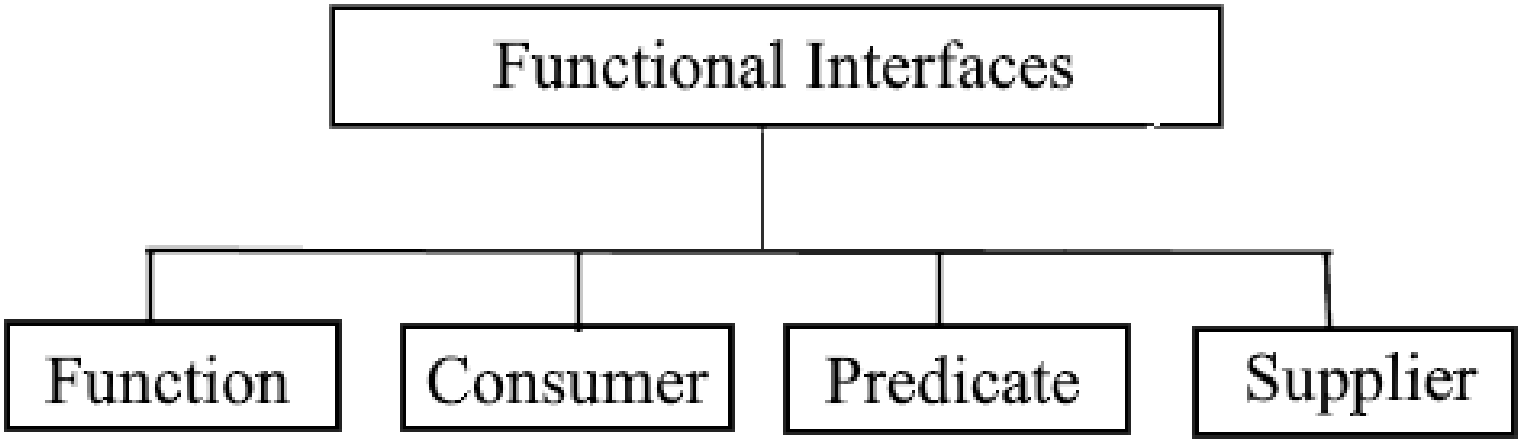
```
// Java program to demonstrate lambda expressions to implement
// a user defined functional interface.
@FunctionalInterface
interface Square
{
        int calculate(int x);
}
class Test
{
        public static void main(String args[])
        {
                int a = 5;
                // lambda expression to define the calculate method
                Square s = (int x)->x*x;

                // parameter passed and return type must be
                // same as defined in the prototype
                int ans = s.calculate(a);
                System.out.println(ans);
        }
}
```

# Importance of Lamda

- 1. If the body of Lamda expression contains only one statement then curly braces are optional.
- 2.

# JAVA 8 FUNCTIONAL INTERFACE CATEGORIES


Functional Interfaces
Function
Consumer
Predicate
Supplier

| Type | Accept Input argument? | Returns Result |
| --- | --- | --- |
| **Predicate** | accepts one argument | It returns a value that is either **true or false.** |
| **Function** | accepts one argument | It returns some value. |
| **Consumer** | accepts one argument | It returns no result |
| **Supplier** | It takes no arguments | It returns some value. |

| Property | Predicate | Function | Consumer | Supplier |
|---|---|---|---|---|
| 1) Purpose | To take some Input and perform some conditional checks | To take some Input and perform required Operation and return the result | To consume some Input and perform required Operation. It won't return anything. | To supply some Value base on our Requirement. |
| 2) Interface Declaration | interface Predicate <T><br>{<br>::::::::::::<br>} | interface Function <T, R><br>{<br>::::::::::::<br>} | interface Consumer <T><br>{<br>::::::::::::<br>} | interface Supplier <R><br>{<br>::::::::::::<br>} |
| 3) Signle Abstract Method (SAM) | public boolean test (T t); | public R apply (T t); | public void accept (T t); | public R get(); |
| 4) Default Methods | and(), or(), negate() | andThen(), compose() | andThen() | - |
| 5) Static Method | isEqual() | identify() | - | - |

| 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> |

| One Argument Functional Interface | Two Argument Functional Interface |
| --- | --- |
| interface Predicate<T><br>{<br>public boolean test(T t);<br>default Predicate and(Predicate P)<br>default Predicate or(Predicate P)<br>default Predicate negate()<br>static Predicate isEqual(Object o)<br>} | interface BiPredicate<T, U><br>{<br>public boolean test(T t, U u);<br>default BiPredicate and(BiPredicate P)<br>default BiPredicate or(BiPredicate P)<br>default BiPredicate negate()<br>} |
| interface Function<T, R><br>{<br>public R apply(T t);<br>default Function andThen(Function F)<br>default Function compose(Function F)<br>static Function identify()<br>} | interface BiFunction<T, U, R><br>{<br>public R apply(T t, U u);<br>default BiFunction andThen(Function F)<br>} |
| interface Consumer<T><br>{<br>public void accept(T t);<br>default Consumer andThen(Consumer C)<br>} | interface BiConsumer<T, U><br>{<br>public void accept(T t, U u);<br>default BiConsumer andThen(BiConsumer C)<br>} |


# Predicate Interface Example

# Java Predicate Interface

It is a functional **interface** which represents a predicate (**boolean-valued function**) of one argument. It is defined in the **java.util.function** package and contains test() a functional method.

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

**Predicate Example to check age is greater than 18 or not.**

**Step: -1**

```
1       public boolean test(Integer age) {
2        if (age > 18) {
3         return true;
4        } else {
5         return false;
6        }
7       }
```

**Step -2**

```
1       public class PredicateExample {
2
3        public static void main(String[] args) {
4         Predicate<Integer> p = age -> (age > 18);
5         System.out.println(p.test(21));  // true
6         System.out.println(p.test(17));  // false
7        }
8
9       }
```

## *Method Summary*

All Methods | Static Methods | Instance Methods | Abstract Methods | Default Methods

| Modifier and Type | Method and Description |
| --- | --- |
| default Predicate<T> | and(Predicate<? super T> other)<br>Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. |
| static <T> Predicate<T> | isEqual(Object targetRef)<br>Returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object). |
| default Predicate<T> | negate()<br>Returns a predicate that represents the logical negation of this predicate. |
| default Predicate<T> | or(Predicate<? super T> other)<br>Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. |
| boolean | test(T t)<br>Evaluates this predicate on the given argument. |

```
// WAP find out all the even numbers in the given array
import java.util.function.Predicate;
public class PredicateDemo
{
	public static void main(String[] args)
	{
		int[] num = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

		Predicate<Integer> even = x -> x % 2 == 0;

		for (int i : num)
		{
			if (even.test(i))
			{
				System.out.println(i);
			}
		}
}
```

**Output:-** 2 4 6 8 10

Interface predicate has three default methods.

1. and(Predicate p)

2. or(Predicate p)

3. negate()

```
import java.util.function.Predicate;
/*Q1. Find out all the even numbers in the given array.
 Q2. Find out all the number which is greater than 5 in the given array.*/
public class PredicateExample2
{
	public static void main(String[] args)
	{
		int[] num = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

		 Predicate<Integer> even = x -> x % 2 == 0;

		 Predicate<Integer> grt = y -> y > 5;

		 for (int i : num) // Enhanced Loop
		   {
			if (grt.or(even).test(i))
			  {
				System.out.println(i);
			}
		}
	}

}
```

For equality predicate interface has isEqual(Object o) method which is static.

```
import java.util.function.Predicate;

public class PredicateExample3
{
    public static void main(String[] args)
    {
            Predicate<String> name = Predicate.isEqual("Mumbai");
            System.out.println(name.test("Mumbai"));
    }
}
```

# Functional Interface in Java

# Functional Chaining

```
import java.util.function.Function;
public class FunctionalChaining
{
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
		  int amount = 3;
		  Function<Integer, Integer> sum = i -> i + i; // 3 + 3 = 6

		  Function<Integer, Integer> sq = i -> i * i; // 6 * 6 = 36
		  System.out.println(sum.apply(amount));

		  System.out.println(sq.apply(amount));
                                                                  //3
		  System.out.println(sum.andThen(sq).apply(amount));
                                  //3(6)     ->       //6(36)

		  System.out.println(sum.compose(sq).apply(amount));
	}                                  9(18)        <-        3(9)
```

Output: 6
36
18

# Differences between predicate and function

| Predicate | Function |
| --- | --- |
| Predicate is use for conditional checks | To perform certain operation and return some result |
| Predicate can take one type Parameter which represents Input argument type. Predicate | Function can take 2 type Parameters. First one represent Input argument type and Second one represent return Type. Function |
| Predicate interface defines only one method called test() | Function interface defines only one Method called apply(). |
| public boolean test(T t) | public R apply(T t) |
| Predicate can return only boolean value. | Function can return any type of value |


# Consumer Interface Example

# Java Consumer Interface

Consumer is a predefined **functional interface** like **Predicate,** present in **java.util.function** package. It has **one abstract method** called **accept();**

Consumer can be used to consume object and perform a certain operation it takes one argument but **won't return** anything.

```
interface Consumer<T>
{
        public void accept(T t);
}
```

```
//Consumer example 1: Take a single argument and print the value.

import java.util.function.Consumer;

public class ConsumerDemo
{
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
		Consumer<String> con = s -> System.out.println(s);
		con.accept("Hello World");
	}

}
```

Output : Hello World

# Consumer Chaining

```
// Consumer Chaining Example
import java.util.function.Consumer;

public class ConsumerChaining
 {

        public static void main(String[] args)
        {
                // TODO Auto-generated method stub
                Consumer<String> con = s -> System.out.println(s.toUpperCase());

                  Consumer<String> con1 = s -> System.out.println("(" + s + ")");

                  con.andThen(con1).accept("Hello World");
        }

}
```

Output: HELLO WORLD
(Hello World)

# Java 8
# Supplier

# Java Supplier Interface

**Supplier** is a **predefined functional** interface like **Consumer**, present in **java.util.function** package. It has **one abstract method** called **get();**

Supplier can be used to supply some value based on some operation, Supplier won't take any input and it will always supply objects.

```
interface Supplier<R>
{
        public R get();
}
```

# Java Supplier Interface

```
import java.util.function.Supplier;

public class SupplierDemo
{
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub
		Supplier<Double> sup= () -> Math.random();
		 System.out.println(sup.get());
	}
}
```

# Method Reference

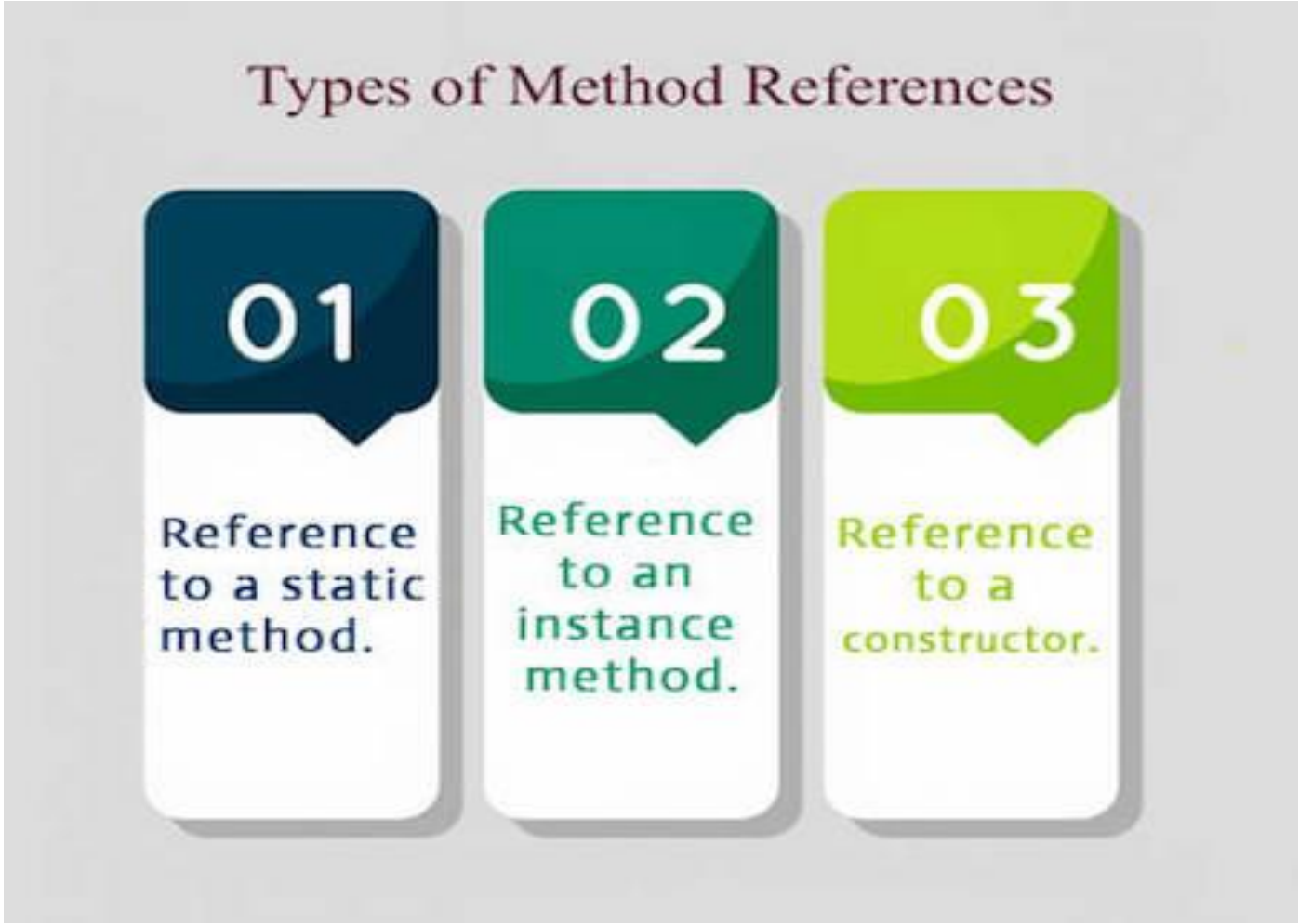
Method References are a special lambda expressions  by references

- They are often used to create simple lamda expression by referencing existing methods.
- Each time when you are using lamda expression to just referring a method you can replace your lamda expression with method reference.
- We can use method reference using double colon operator  **::**

# Types of Method References

There are following types of method references in java:

- Reference to a static method.
- Reference to an instance method.
- Reference to a constructor.


Types of Method References
01
Reference to a static method.
02
Reference to an instance method.
03
Reference to a constructor.

# Reference to a static method

```
package java8;
interface Engine
{
    abstract public void speed();
}
class   Car
{
   public static void average()
   {
        System.out.println("The averge of the car is very Good");
   }
}
public class MethodReference
{
        public static void main(String[] args)
        {
                // Using Lamda epression
                        Engine  obj = ()->System.out.println("Speed is Good");
                        obj.speed();
                                        // Reference to a static method we can use in place of lamda expression
                // Lamda replaced with static method Reference
                        Engine obj1= Car::average;
                        obj1.speed();
        }

}
```

# Reference to a instance method

```
package java8;
interface Engine
{
    abstract public void speed();
}
class   Car
{
   public void wheel()
   {
        System.out.println("Car has wheels of MRF company");
   }
}
public class MethodReference
{
        public static void main(String[] args)
        {
                // Using Lamda epression
                    Engine  obj = ()->System.out.println("Speed is Good");
                    obj.speed();

                // Lamda replaced with instance method Reference
                Car car  = new Car();
                Engine obj4=car::wheel;
               obj4.speed();
        }
```

# Reference to a Constructor

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.

# Reference to a Constructor

```
package java8;
interface Engine
{
    abstract public void speed();
}
class   Car
{
   public Car()
   {
        System.out.println(“This is constructor of Car");
   }
}
public class MethodReference
{
        public static void main(String[] args)
        {
                // Using Lamda epression
                        Engine  obj = ()->System.out.println("Speed is Good");
                        obj.speed();

                // Lamda replaced with instance method Reference
                Engine obj4=Car::new;
                pbj4.speed();
         }
}
```

# Java 8

# Stream Api

# Default Keyword in Interfaces

# Introductions to streams

- These streams are related to collection Framework/(**Group of objects).** These streams are very different from io stream. IO streams are the sequence of data.

- These streams were introduced in java 1.8 version.

- Stream API is basically perform bulk operations and process the objects of collection .

- Streams reduce the code length.

# Introductions to streams

What is Stream?

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream −

**Sequence of elements** − A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.

**Source** − Stream takes Collections, Arrays, or I/O resources as input source.

**Aggregate operations** − Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.

**Non-terminal Operations**

A non-terminal stream operation is an intermediate operation which creates a new Stream by transforming or filtering the original stream.

1. filter()
2. map()
3. flatMap()
4. distinct()
5. limit()
6. peek()

**Terminal Operations**

A terminal stream operation is an operation that returns a result.

1. anyMatch()
2. allMatch()
3. noneMatch()
4. collect()
5. count()
6. findAny()
7. findFirst()
8. forEach()
9. min()
10. max()
11. reduce()
12. toArray()

1. Introduction to Stream API
2. Java code without stream vs with stream api.
3. How to create the object of stream and processing data.
4. Filter and map example
5. Some important methods like
    a.) collect()
    b.) sorted()
    c.) min()
    d.) max()
    e.) forEach(),
    f.) toArrays(), Stream.of()

# Introductions to streams

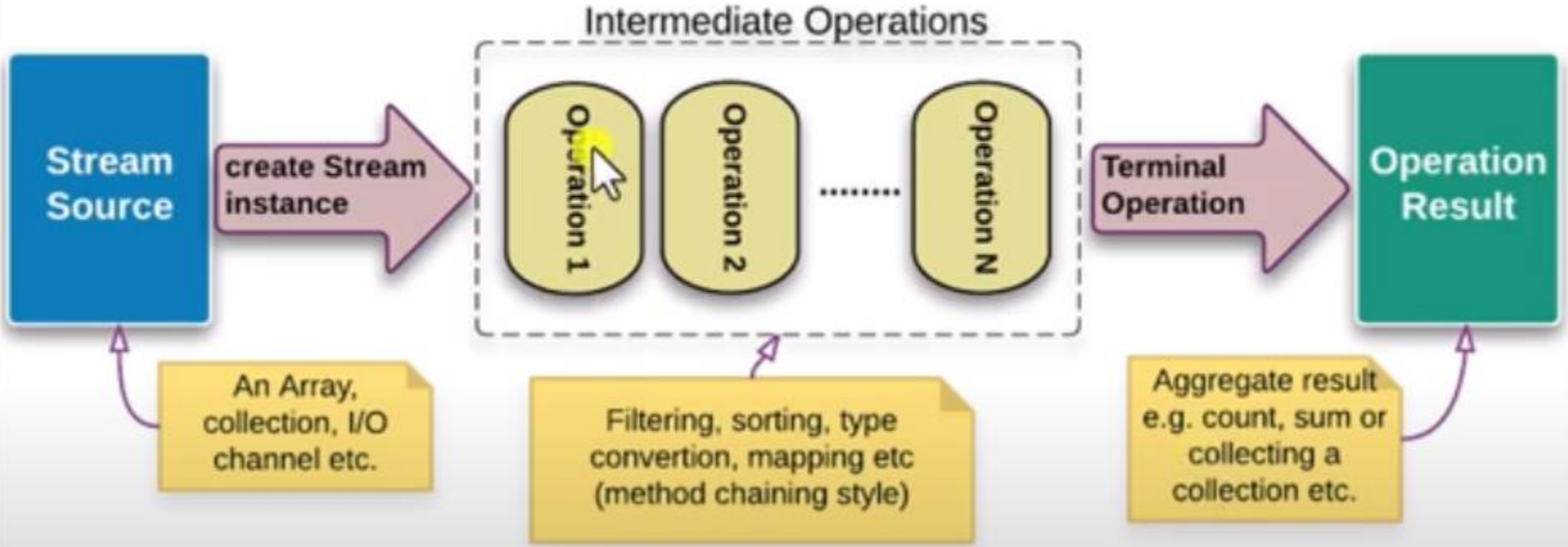
**Pipelining** − Most of the stream operations return stream itself so that their result can be pipelined.

These operations are called intermediate operations and their function is to take input, process them, and return output to the target.

**collect()** method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

**Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is require

# Java Streams



In Java, we will use collection to store the data. We will use stream to process the data.

Few important points about stream
---

1. A stream is not a data structure.

2. Stream is used to perform bunch of operations on the data.

3. Stream will take the data from a collection or Array.

4. Stream will perform operations on the data but it will not change original data structure.

Stream Creation

---

Stream can be created from different sources.

1. Using of() methods we can create stream

2. Using stream() method also we can create stream.

Aproach-1:

---

```
Stream<Integer>  stream =      Stream.of(1,2,3,4,5,6);
```

Approcach-2:

---

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

Stream Creation

---

Approcach-2:

---

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

                list.add("Anushka");
               list.add("Isha");
                list.add("9Tara");
                list.add("Deepika Padukone");
                list.add("Pooja Hegde");

                Stream<String> stream=  list.stream();
```

-> By using Streams we can perform below operations on the data

1. **Intermediate Operations(Non-Terminal)**
2. **Terminal Operations**

-> By using Streams we can perform below operations on the data

1. **Intermediate Operations** method will produce any results. They usually accept functional interface as parameter and always returns new streams. Some examples for intermediate operations are filter() & map() methods.

2. **Terminal Operations** methods will produce some results i.e count(), collect() etc.

---

**Stream with Filtering**

---

1. **Stream interface having filter() method.**
2. **Filter() method will take predicate as input**
3. **Predicate is a functional interface which will take input and return boolean value.**
4. **Predicate interface contains test() as abstract method is used to execute lamda.**

```
//1.  Number filters
public class NumFilters
{

	public static void main(String[] args)
	{
		Stream<Integer> stream = Stream.of(4,8,12,6,7,11,24);

		// How to filter the data
		stream.filter(num->num>=6)
			.forEach(System.out::println);

	}

}
```

```
// 2. Names filters
public class NamesFilter
{
	public static void main(String[] args)
	{
		List<String>   list = new ArrayList<String>();

		 list.add("Anushka");
		list.add("Isha");
		 list.add("9Tara");
		 list.add("Deepika Padukone");
		 list.add("Pooja Hegde");
		 list.add("Anumpama");
		 list.add("Amisha");

		 list.stream().filter(s->s.startsWith("A")).
				forEach(System.out::println);



	}
		output : Anushka	Anumpama	Amisha
}
```

```
class Person                    // Program 3
{
        private String name;

        private Integer age;

        private String job;

        public Person(String name, Integer age, String job) {
        super();
        this.name = name;
        this.age = age;
        this.job = job;
        }


        public String getName() {
        return name;
        }


        public void setName(String name) {
        this.name = name;
        }
```

```
public Integer getAge() {
    return age;
}
public void setAge(Integer age) {
       this.age = age;
}

public String getJob() {
         return job;
}

public void setJob(String job) {
        this.job = job;
}

@Override
public String toString()
{
              return "Person [name=" + name + ", age=" + age + ", job=" + job + "]";
};

}
```

```
// How to sort the data from object
class PersonFilter
{
	public static void main(String[] args)
	{
		Person p1 = new Person("ram", 22, "Software");
		Person p2 = new Person("syam", 23, "Teacher");
		Person p3 = new Person("rohan", 25, "Driver");
		Person p4 = new Person("mohan", 21, "Mechanic");

		    List<Person> personList = Arrays.asList(p1,p2,p3,p4);



		    List<Person> collect = personList.stream().sorted((o1,o2)->o1.getAge()-o2.getAge())
		        .collect(Collectors.toList());

		    collect.forEach(System.out::println);



	}
}
```

# Mapping Operations in Stream

1. Mapping operations are belongs to Intermediate operations

2. Mapping Operations are those operations that transform the elements of a stream and returns a new stream with transformed elements.

```
public class NamesMapping {

   public static void main(String[] args)
   {
      List<String>   list = new ArrayList<String>();

      list.add("Anushka");
     list.add("Isha");
      list.add("9Tara");
      list.add("Deepika Padukone");
      list.add("Pooja Hegde");
      list.add("Anumpama");
      list.add("Amisha");

      list.stream().map(name>name.toUpperCase()).forEach(System.out::println);

   }
}
```

**Slicing operations in the streams**

---

1. distincts() -> It is used to get unique elements from the stream

2. limit() -> It is used to get number of elements from the stream based on given size.

3. skip(long maxSize) -> it is used to skip elements from starting to given length and return remaining elements.

```
public class SlicingDemo
{

    public static void main(String[] args)
    {
        List<String> countries = new ArrayList<String>();

        countries.add("India");
        countries.add("USA");
        countries.add("UK");
        countries.add("China");
        countries.add("India");


        // Getting unique values from collections
        countries.stream().distinct().
                forEach(System.out::println);
```

```
System.out.println("================");

// Getting specific number of records from collection using limit()
countries.stream().limit(2).
        forEach(System.out::println);
```

India
USA

```
System.out.println("================");

// Removing first N no, of records

countries.stream().skip(2).
        forEach(System.out::println);
}

}
```

UK
China
India

Matching Operations in the Streams
---

Matching operations are terminals operations that are used to check if elements with certain  are present in the streams or not.

There are mainly  three matching functions in the strea . These are

1. anyMatch()
2. allMatch()
3. noneMatch()

```
public class MatchingDemo
{

	public static void main(String[] args)
	{

		List<Person1> list = new ArrayList<Person1>();

		list.add(new Person1("David",23,"India"));
		list.add(new Person1("Joy",25,"USA"));
		list.add(new Person1("Ryan",50,"Canada"));
		list.add(new Person1("Ram",45,"India"));
		list.add(new Person1("Ching",56,"China"));

		boolean isIndiaAvailable = list.stream().anyMatch(p->p.getCountry().equals("India"));

		 System.out.println(isIndiaAvailable);

		  boolean allMatch = list.stream().allMatch(p->p.getCountry().equals("India"));
		  System.out.println("All persons are indians or not  "+allMatch);


		  boolean noneMatch = list.stream().noneMatch(p->p.getCountry().equals("Germany"));
		  System.out.println("No German are avaiable "+noneMatch);

	}

}
```

# Finding Operations in stream

In Matching operations we can check weather data elements present in the stream or not based on given criteria. After checking condition it returns true or false value.

-> By using Finding operation we can check the condition and we can get the data based on condition.

-> Finding Operation always return Optional type of Object.

1. findFirst()

2. findAny()

```
public class FindingDemo
{

	public static void main(String[] args)
	{
		List<Person1> list = new ArrayList<Person1>();

		 list.add(new Person1("David",23,"India"));
		 list.add(new Person1("Joy",25,"USA"));
		 list.add(new Person1("Ryan",50,"Canada"));
		 list.add(new Person1("Ram",45,"India"));
		 list.add(new Person1("Ching",56,"China"));

		 Optional<Person1> findFirst = list.stream().filter(p->p.getCountry().equals("India")).findFirst();

		  if(findFirst.isPresent())
		   {
		       System.out.println(findFirst.get());
		   }

	}

}
```

# Collectors in Streams

Collectors operations are used to collect the data from streams

We are having below methods to perform Collectors Operations

1. Collectors.toList()
2. Collectors.toSet()
3. Collectors.toMap()
4. Collectors.toCollections(0 etc.

```
public class CollectorsToListDemo {

	public static void main(String[] args)
	{
		List<Employee> list = new ArrayList<Employee>();

		list.add(new Employee("Ram",23,20000));
		list.add(new Employee("Ashok",25,30000));
		list.add(new Employee("Suresh",33,25000));
		list.add(new Employee("charan",26,40000));


		List<String> empNames = list.stream().map(e->e.getName()).collect(Collectors.toList());

		System.out.println(empNames);

	}

}
```

```
public class CollectorsToMapDemo
{

	public static void main(String[] args)
	{
		List<String> list = new ArrayList<String>();
		list.add("Rahul");
		list.add("Sachin");
		list.add("Hardik");

		list.add("Dhoni");

		Map<String, Integer> namesMap = list.stream().collect(Collectors.toMap(s->s,s->s.length()));
		System.out.println(namesMap);

	}

}
```

# Stream API Methods

1. Getting Min salary of the employee from collection
2. Getting Max salary of the employee from the collection.
3. Getting averaging salary
4. Group By

```
class Employee
{
        private String name;
        private int age;

        private int salary;

        public Employee(String name, int age, int salary) {
        super();
        this.name = name;
        this.age = age;
        this.salary = salary;
        }

        public String getName() {
        return name;
        }

        public void setName(String name) {
        this.name = name;
        }

        public int getAge() {
        return age;
        }

        public void setAge(int age) {
        this.age = age;
        }

        public int getSalary() {
        return salary;
        }

        public void setSalary(int salary)
        {
        this.salary = salary;
        }

        @Override
        public String toString() {
        return "Employee [name=" +
        name + ", age=" + age + ",
        salary=" + salary + "]";
        }
```

```
public class EmpMinMaxAvgDemo {

public static void main(String[] args)
{
        List<Employee> list = new ArrayList<Employee>();

         list.add(new Employee("Ram",23,20000));
         list.add(new Employee("Ashok",25,30000));
         list.add(new Employee("Suresh",33,25000));
         list.add(new Employee("charan",26,40000));

// Getting Average salary
Double avgSalary = list.stream().collect(Collectors.averagingInt(emp->emp.getSalary()));
        System.out.println("Employee Average salary  "+avgSalary);
```

```
        Optional<Employee> minSalary =
list.stream().collect(Collectors.minBy(Comparator.comparing(Employee::getSalary)));

    if(minSalary.isPresent())
    {
    System.out.println("Minimum Salary     "+minSalary);
    }

    Optional<Employee> maxSalary =
list.stream().collect(Collectors.maxBy(Comparator.comparing(Employee::getSalary)));
    if(maxSalary.isPresent())
    {
    System.out.println("Maximum Salary     "+maxSalary);
    }
```

Find the frequency of each character

```
public class FrequencyCount {

public static void main(String[] args)
{
        List<Integer> list = Arrays.asList(1,2,2,3,4,5,5,6);

        Map<Integer, Long> mapList = list.stream().collect(Collectors.groupingBy(num->num,
        Collectors.counting()));



        List<Integer> list2 = mapList.entrySet().stream().filter(entry->entry.getValue()>1)
        .map(entry->entry.getKey()).collect(Collectors.toList());

        System.out.println(list2);

}

}
```

# How Stream Works?

# Java 8 Streams #2 - Filter and forEach Example

- Java stream provides a filter() method to filter stream elements on the basis of given predicate. This method takes predicate as an argument and returns a stream of consisting of resulted elements.
- In this example, we will create list of products and we filter products whole price is greater than 25k.
- We display list of products usinf forEach() method

# Filter()

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collector;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Test
{

    public static void main(String[] args)
    {

        List<Integer> list = new ArrayList<Integer>();

        list.add(33);
        list.add(44);
        list.add(55);
        list.add(66);
        list.add(77);

        Stream<Integer> obj = list.stream();

        //  obj.forEach(e->System.out.println(e));;
         List<Integer> l2= list.stream().filter(e->e>44).collect(Collectors.toList());
         System.out.println(l2);
    }
}
```

# Java 8 Stream API – Sorting List

- Sorting List of String objects in Ascending order using Stream API
- Sorting List of String objects in Descending order using Stream API

- Sorting List of Employee objects (By Salary) in Ascending order using Stream API
- Sorting List of Employee objects (By Salary) in Descending order using Stream API

- Sorting List of Employee objects (By Age) in Ascending order using Stream API
- Sorting List of Employee objects (By Age) in Descending order using Stream API

- Sorting List of Employee objects (By Name) in Ascending order using Stream API
- Sorting List of Employee objects (By Name) in Descending order using Stream API

# Java 8 Stream #4 - map() and collect() Methods

- The Stream.map() method is used to transform one object into another by applying a function.
- The collect() method of Stream class can be used to accumulate elements of any Stream into a Collection.

# How to Create Stream Object

# Parallel Streams

The powerful feature of streams is that stream pipelines may execute either sequentially or in parallel. All collections support the parallelStream() method that returns a possibly parallel stream:

```
Stream<Student> parallelStream = studentList.parallelStream();
```

When a stream executes in parallel, the Java runtime divides the stream into multiple sub streams. The aggregate operations iterate over and process these sub streams in parallel and then combine the results.


The advantage of parallel streams is performance increase on large amount of input elements, as the operations are executed currently by multiple threads on a multi-core CPU.

# Java 8 Interview QA

# Optional Class

Optional.empty()
Optional.of()
Optional.ofNullable()
.isPresent()
.ifPresent()
.get()
.orElse()
.orElseGet()

orElse v/s orElseGet()
.orElseThrow()
.filter()
.map()
.flatMap()

# What is optional in java 8 ?

Optional is final class in java 8 to handle values as **available** or **not available** **instead** of **checking null values**.

What are the different ways to create an optional Object

1) Optional.empty().

2) Optional.of()

3) Optional.ofNullable()

java.util package

# .get()

If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

```
// Check the value
  if(opStr1.isPresent()) {
          System.out.println(opStr1.get());
  }
```

java.util package

# .ifPresent(consumer)

if a value is present, it invokes the specified consumer with the value, otherwise does nothing.

```
opStr2.ifPresent(s -> System.out.println(s.toUpperCase()));
```

java.util package

# .orElse() / "default value"

Returns the value if present, otherwise returns other.

```
T orElse(T other)
```

```
Optional<String> empty = Optional.empty();
// set default value
System.out.println(empty.orElse("default value"));

// orElse
String s1 = null;
String out = Optional.ofNullable(s1).orElse("some default value");
```

java.util package

# .orElseGet(supplier)

Returns the value if present, otherwise invokes the supplier

```
// orElseGet (Supplier interface)
Optional<Date> emptyDt = Optional.empty();
System.out.println(emptyDt.orElseGet(() -> new Date()));
```

java.util package

# . orElseThrow(supplier)

Returns the contained value, if present, otherwise throws an exception to be created by the provided supplier.

```
// orElseThrow
Optional<Integer> item = Optional.empty();
//Optional<Integer> item = Optional.of(1);
int x  = item.orElseThrow(IllegalArgumentException ::  new);
System.out.println(x);
```

java.util package

# . filter(predicate)

If a value is present and the value matches a given predicate, it returns an Optional describing the value, otherwise returns an empty Optional.

```
// Filter - apply a predicate
Optional<Integer> age = Optional.of(20);
int a = age.filter(n -> n> 18).get();
System.out.println(a);

Optional<String> text = Optional.ofNullable("okayjava");
System.out.println(text.filter(st -> st.startsWith("o")).isPresent());
```

```
import java.util.HashMap;
import java.util.Map;
public class IterateMapUsingLambda {
                public static void main(String[] args) {
                Map<String, Integer> prices = new HashMap<>();
                prices.put("Apple", 50);
                prices.put("Orange", 20);
                prices.put("Banana", 10);
                prices.put("Grapes", 40);
                prices.put("Papaya", 50);

                /* Iterate without using Lambda
                   for (Map.Entry<String, Integer> entry : prices.entrySet())
                   {
                 System.out.println("Fruit: " + entry.getKey() + ", Price: " + entry.getValue())
                    }
                */

                 prices.forEach((k,v)->System.out.println("Fruit: " + k + ", Price: " + v));

                }
}
```

**Output:**

```
Fruit: Apple, Price: 50
Fruit: Grapes, Price: 40
Fruit: Papaya, Price: 50
Fruit: Orange, Price: 20
Fruit: Banana, Price: 10
```

# Java 8 Date and Time API

```
// To print local Date and Time.

import Java.time.*;
public class Java8DateTime
{
        public static void main(String[] args)
        {
                LocalDate date = LocalDate.now();
                System.out.println(date);
                LocalTime time=LocalTime.now();
                System.out.println(time);
        }
}
```

```
import Java.time.*;
class DateAPI
{
	public static void main(String[] args)
	{
		LocalDate date = LocalDate.now();

		System.out.println(date);

		int day   = date.getDayOfMonth();
		int month = date.getMonthValue();
		int year  = date.getYear();

		System.out.println(day+"..."+month+"..."+year);
		System.out.printf("\n%d-%d-%d",day,month,year);
	}
}
```