Chapter - 12

JAVA 8 – New Features

Java 8 Features

-> Java 8 introduced lot of new features

-> Java 8 version changed coding style with those new features

Main aim of java - 8

-> To simplify programming

-> To enable functional programming

-> To write more readable & concise code

New Features in Java 8

- Interface changes (default & static methods)
- Lambda Expressions
- Functional Interfaces
    - Consumer
    - Supplier
    - Predicate
    - Function
- Stream API
- Date & Time API changes
- Optional class
- SplIterator
- Stringjoiner
- Method References
- Constructor References
- Collections Framework changes

Interface Changes in Java 8

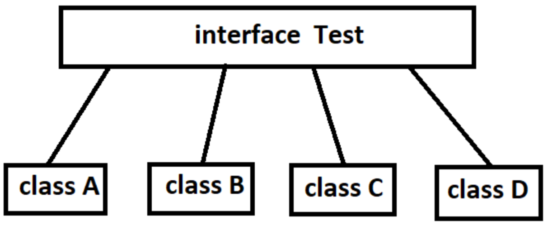
-> Prior to java 8, interface should have only abstract methods (methods without body)

-> Java 8 allows the interfaces to have default and static methods

Q) What is the advantage of having Default & Static methods in java 8?

-> Default & Static methods provides backward compatibility.

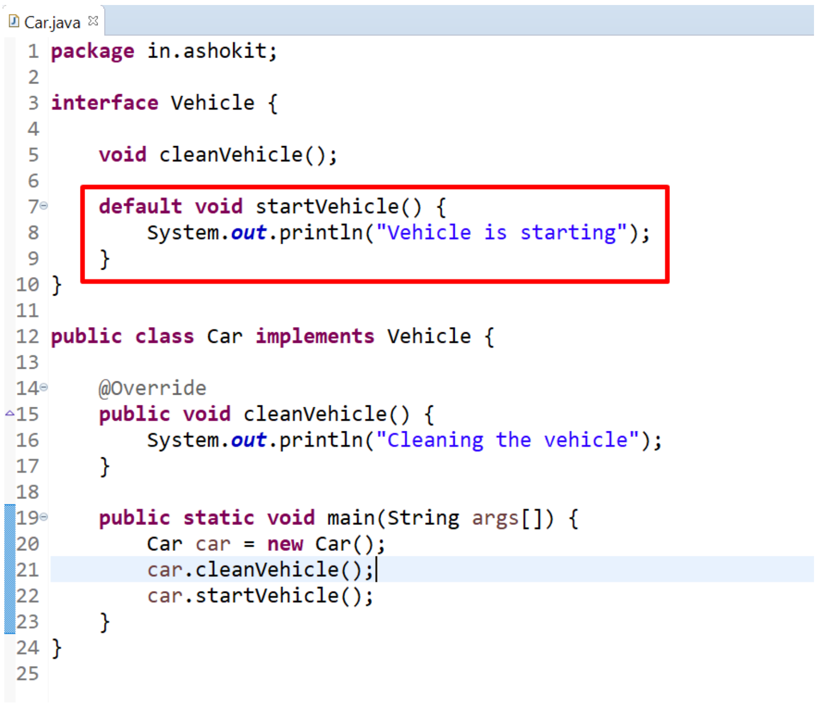
-> For example, if several classes such as A, B, C and D implements an interface Test then if we add a new method to the Test, we have to change the code in all the classes (A, B, C and D) that implements this interface.



-> In this example we have only four classes that implements the interface, but imagine if there are hundreds of classes implementing an interface it will become very difficult to change all the classes which are implementing that interface. This is why in java 8, we have a new concept "default methods".

-> Default methods can be added to any existing interface and we do not need to implement these methods in the implementation classes (if required we can override them in implementation classes)

// java program on Interface – Default Method



```java
package in.ashokit;

interface Vehicle {

    void cleanVehicle();

    default void startVehicle() {
        System.out.println("Vehicle is starting");
    }
}

public class Car implements Vehicle {

    @Override
    public void cleanVehicle() {
        System.out.println("Cleaning the vehicle");
    }

    public static void main(String args[]) {
        Car car = new Car();
        car.cleanVehicle();
        car.startVehicle();
    }
}
```

Some key points about default methods are :

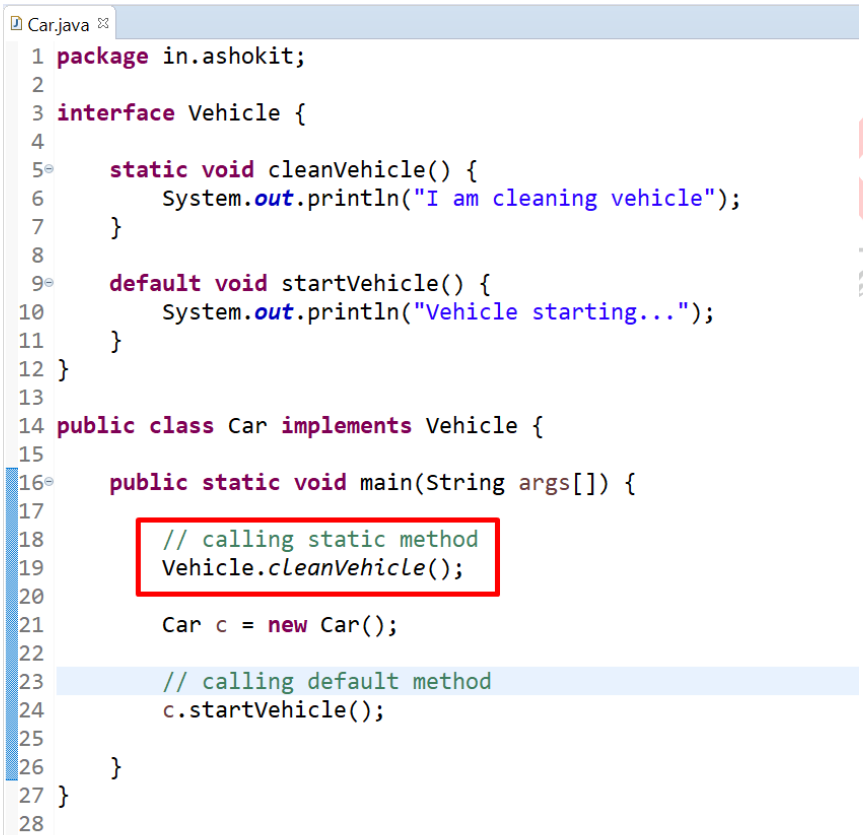
- A default method must have a body.
- The access modifier of default methods are implicitly public.
- The class implementing such interface are not required to implement default methods. If needed, implementing class can override default methods.

Static Methods In Java

-> The static methods in interfaces are similar to default methods but the only difference is that you can't override them. Now, why do we need static methods in interfaces if we already have default methods?

-> Suppose you want to provide some implementation in your interface and you don't want this implementation to be overridden in the implementing class, then you can declare the method as static.

-> In the below example, we will defined a Vehicle interface with a static method called cleanVehicle().


Car.java

```java
package in.ashokit;

interface Vehicle {

    static void cleanVehicle() {
        System.out.println("I am cleaning vehicle");
    }

    default void startVehicle() {
        System.out.println("Vehicle starting...");
    }
}

public class Car implements Vehicle {

    public static void main(String args[]) {

        // calling static method
        Vehicle.cleanVehicle();

        Car c = new Car();

        // calling default method
        c.startVehicle();

    }
}
```

Some key points about static methods are:

- A static method must have a body.
- The access modifier of static methods is implicitly public.
- This method must be called using interface name.
- Since these methods are static, we cannot override them in implementing class.

Functional Interfaces

-> If an interface contains only one abstract method, then it is called as Functional Interface.

-> Functional Interface is used to invoke lambda expressions

-> Runnable, Callable, Comparable and ActionListener are predefined functional interfaces.

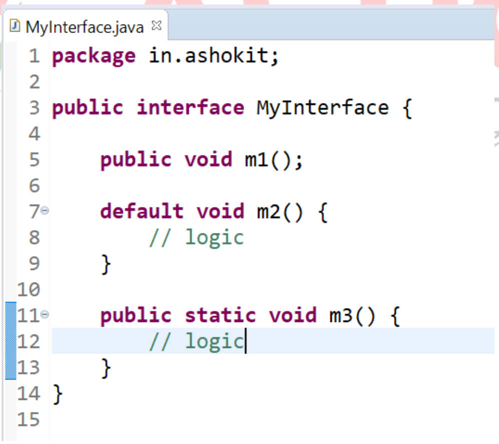
Runnable ----> run ( )

Callable ----> call ( )

ActionListener ----> actionPerformed ( )

Comparable ----> compareTo ( )

Note: We can take default and static methods also in functional interface. Only one method rule is applicable for only abstract methods.


MyInterface.java

```java
package in.ashokit;

public interface MyInterface {

    public void m1();

    default void m2() {
        // logic
    }

    public static void m3() {
        // logic
    }
}

```

-> To represent our interface as functional interface we will use @FunctionalInterface

```
@FunctionalInteface
public interface MyInterface {

    public void m1();
   public void m2();

}
```

// invalid bcz of 2 abstract methods

```
@FunctionalInterface
public interface MyInterface{

}
```

// invalid bcz of no abstract methods

```
@FunctionalInterface
public interface Parent {
   public void m1();
}

@FunctionalInterface
public interface Child extends Parent {

}
```

// valid bcz child inheriting from Parent

What is lambda expression

-> Java is an object-oriented language. By introducing lambdas in Java 8, the authors of Java tried to add elements of functional programming in Java.

Q- Now you might be wondering what the difference between object-oriented programming and functional programming is?

-> In object-oriented programming, objects and classes are the main entities. If we create a function then it should exist within a class. A function has no meaning outside the scope of the class object.

-> In functional programming, functions can exist outside the scope of an object. We can assign them to a reference variable and we can also pass them to other methods as a parameter.

-> A lambda expression is just an anonymous function, i.e., a function with no name and that is not bound to an identifier. We can pass it to other methods as parameters, therefore, using the power of functional programming in Java.

-> Lambda is an Anonymous function

- No Name
- No Modifier
- No Return Type

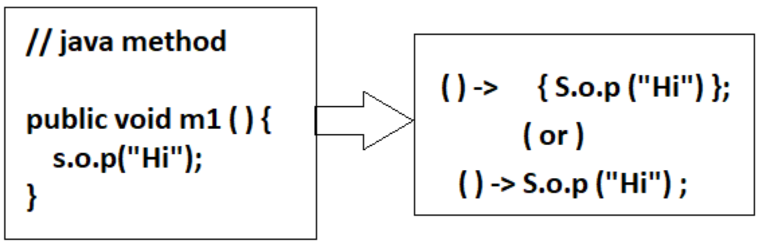
Why to use Lambda expressions?

-> To write functional programming in java

-> To write more readable, maintainable and concise code

-> To enable parallel processing

Example - 1



```
// java method

public void m1 ( ) {
   s.o.p("Hi");
}
```

```
( ) ->    { S.o.p ("Hi") };
        ( or )
 ( ) -> S.o.p ("Hi") ;
```

Example - 2

```
public void add (int a, int b) {
        int c = a+b;
        s.o.p (c);
}
```

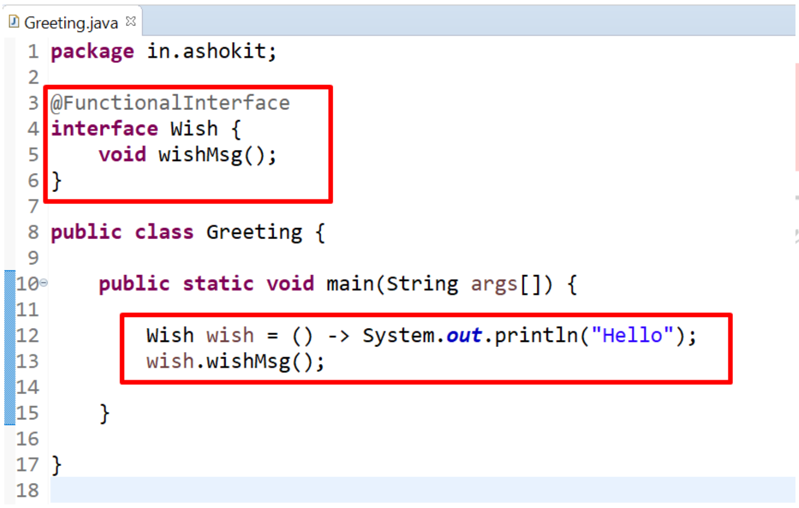
```
(int a, int b) -> { s.o.p  (a+b)  };
      ( or )
(a,b) -> s.o.p (a+b);
```

Example - 3

```
public int getLength(String s) {
    return s.length();
}
```

```
(s) -> s.length();
      ( or )
s -> s.length () ;
```

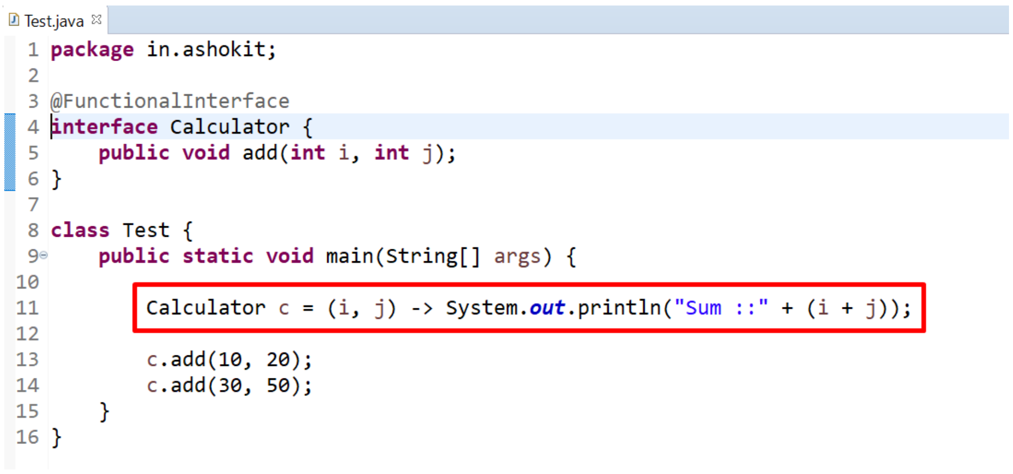
// Java Program with Functional Interface and Lambda Expression



Greeting.java

```java
package in.ashokit;

@FunctionalInterface
interface Wish {
    void wishMsg();
}

public class Greeting {

    public static void main(String args[]) {

        Wish wish = () -> System.out.println("Hello");
        wish.wishMsg();

    }

}
```

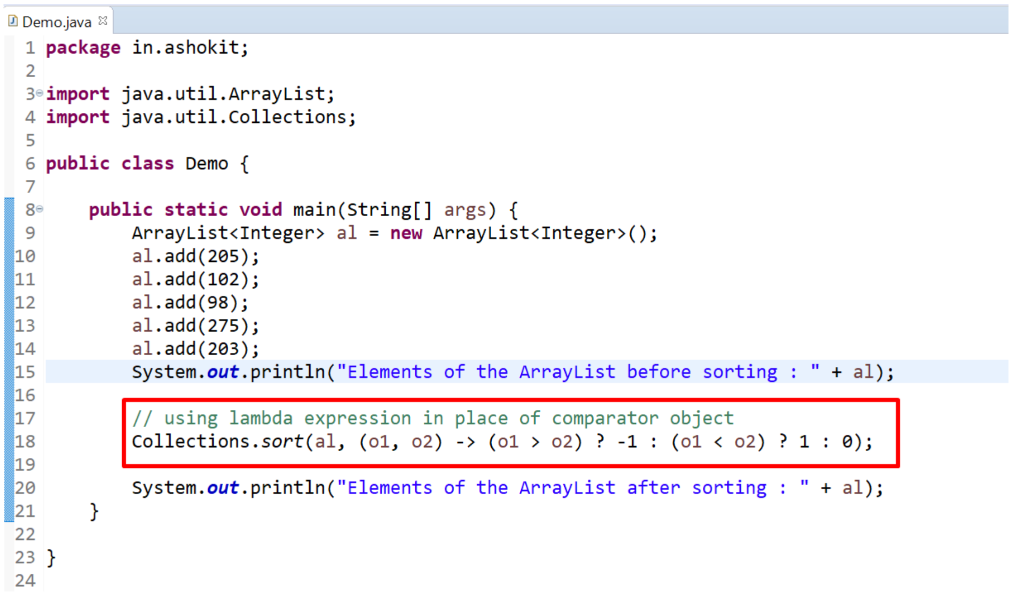
// Java Program with Functional Interface and Lambda Expression



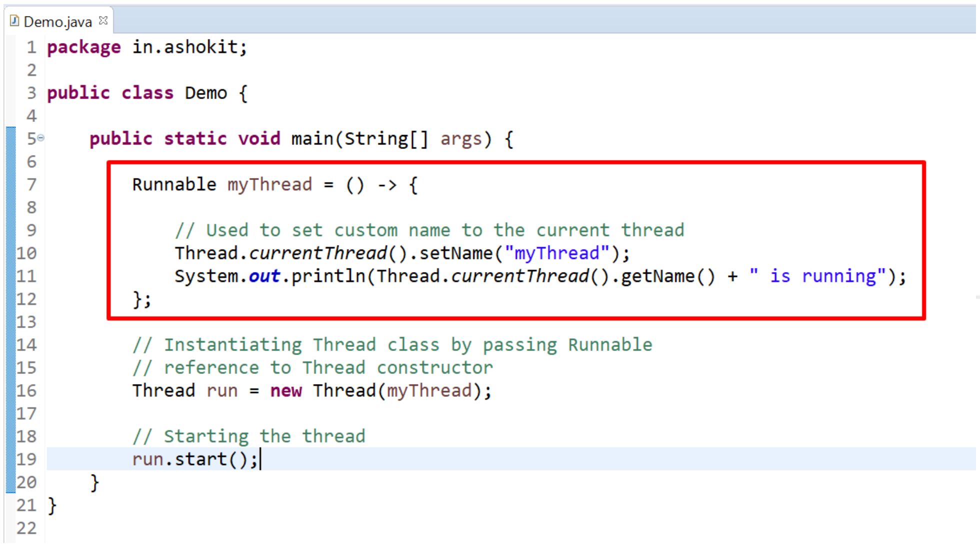
Test.java

```java
package in.ashokit;

@FunctionalInterface
interface Calculator {
    public void add(int i, int j);
}

class Test {
    public static void main(String[] args) {

        Calculator c = (i, j) -> System.out.println("Sum ::" + (i + j));

        c.add(10, 20);
        c.add(30, 50);
    }
}
```

// Java Program with Lambda Expression to Sort ArrayList elements


Demo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.Collections;

public class Demo {

    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(205);
        al.add(102);
        al.add(98);
        al.add(275);
        al.add(203);
        System.out.println("Elements of the ArrayList before sorting : " + al);

        // using lambda expression in place of comparator object
        Collections.sort(al, (o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);

        System.out.println("Elements of the ArrayList after sorting : " + al);
    }

}
```


Demo.java

```java
package in.ashokit;

public class Demo {

    public static void main(String[] args) {

        Runnable myThread = () -> {

            // Used to set custom name to the current thread
            Thread.currentThread().setName("myThread");
            System.out.println(Thread.currentThread().getName() + " is running");
        };

        // Instantiating Thread class by passing Runnable
        // reference to Thread constructor
        Thread run = new Thread(myThread);

        // Starting the thread
        run.start();
    }
}
```

Predefined Functional Interfaces

There are several predefined functional interfaces provided by java 8

1) Predicate
2) Function
3) Consumer
4) Supplier

-> These functional interfaces are provided in java.util.function package

Predicate

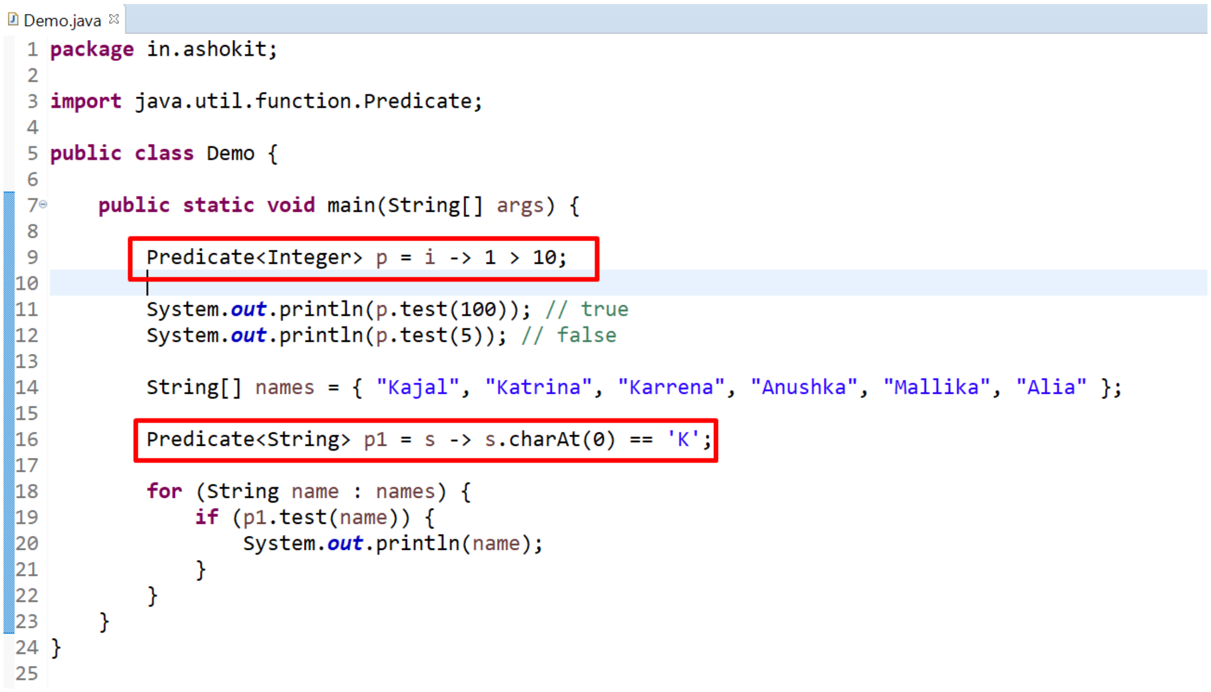
-> It is used to perform some conditional check and returns true or false value

Ex: Check weather no is greater than 10 or not

Note: Predicate is boolean valued function in java 8

Syntax:

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



Demo.java

```java
package in.ashokit;

import java.util.function.Predicate;

public class Demo {

    public static void main(String[] args) {

        Predicate<Integer> p = i -> 1 > 10;

        System.out.println(p.test(100)); // true
        System.out.println(p.test(5)); // false

        String[] names = { "Kajal", "Katrina", "Karrena", "Anushka", "Mallika", "Alia" };

        Predicate<String> p1 = s -> s.charAt(0) == 'K';

        for (String name : names) {
            if (p1.test(name)) {
                System.out.println(name);
            }
        }
    }
}
```

Predicate Joining

-> To combine multiple predicates, we will use predicate joining

-> In Predicate we have below methods

test () - > abstract method

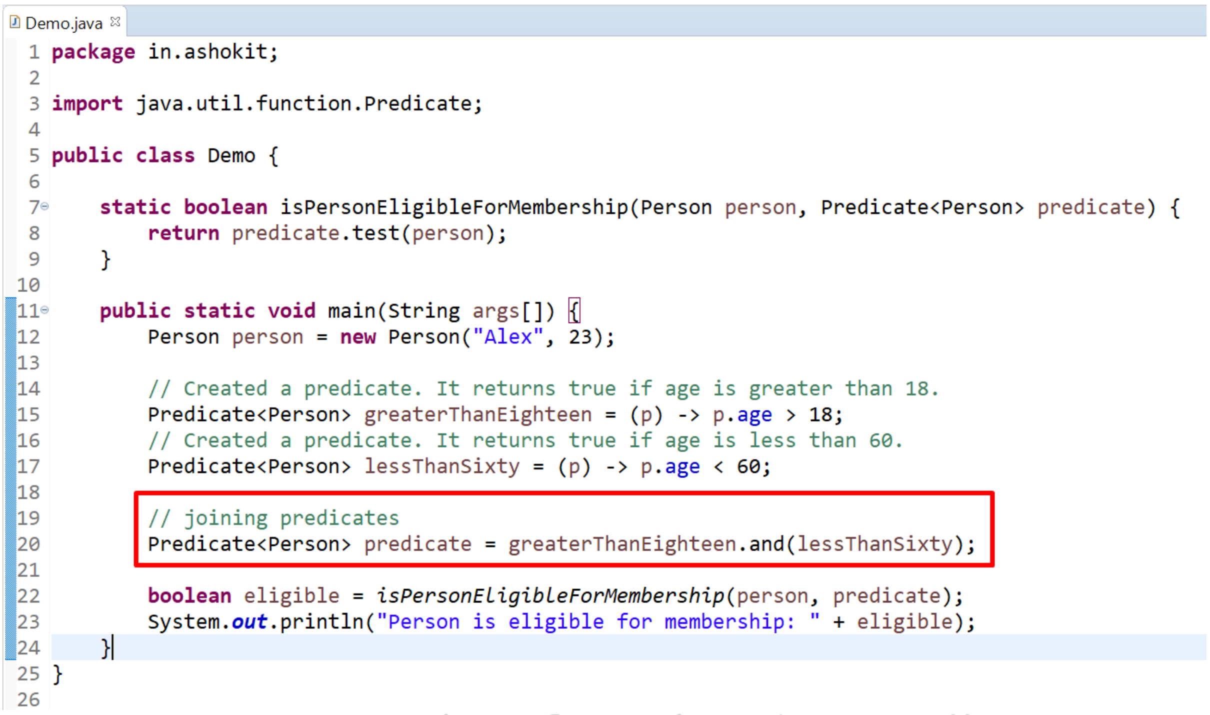
p1.negate();

p1.and(p2);

p1.or(p2);

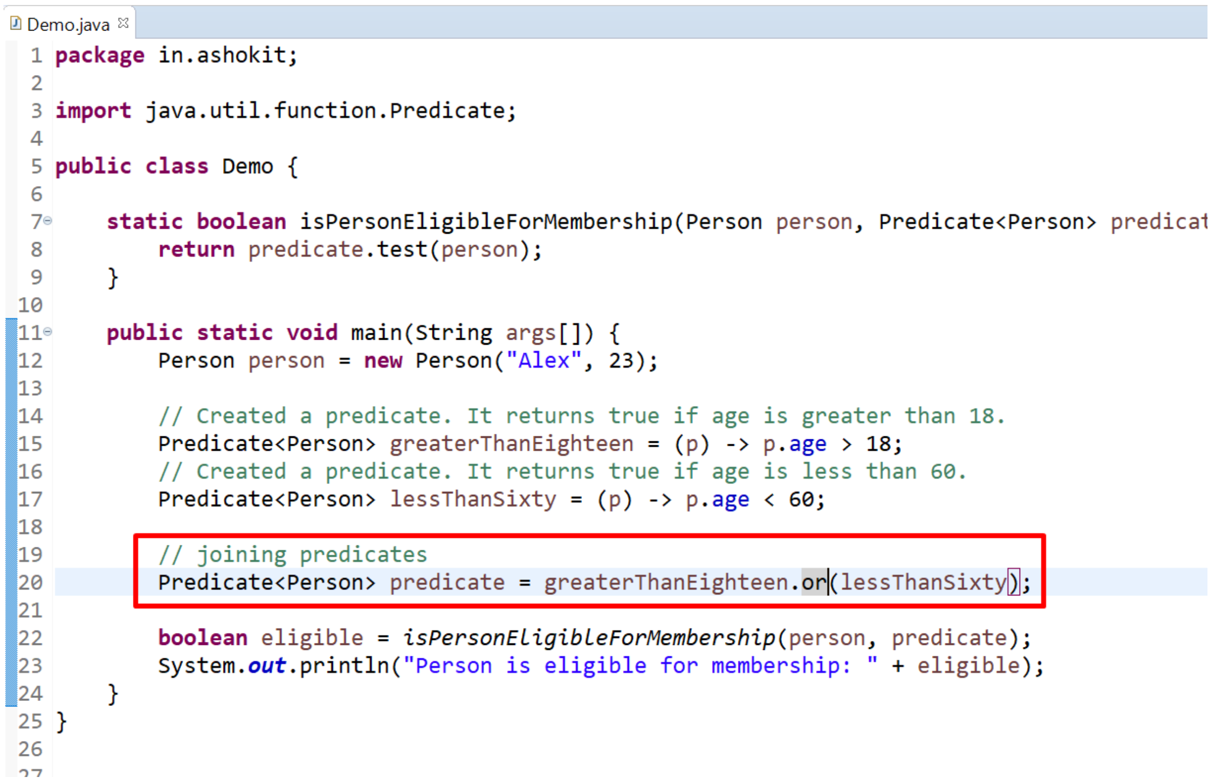
Note: negate(), and() , or() methods are default methods in Predicate interface

```java
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```


Demo.java

```java
package in.ashokit;

import java.util.function.Predicate;

public class Demo {

    static boolean isPersonEligibleForMembership(Person person, Predicate<Person> predicate) {
        return predicate.test(person);
    }

    public static void main(String args[]) {
        Person person = new Person("Alex", 23);

        // Created a predicate. It returns true if age is greater than 18.
        Predicate<Person> greaterThanEighteen = (p) -> p.age > 18;
        // Created a predicate. It returns true if age is less than 60.
        Predicate<Person> lessThanSixty = (p) -> p.age < 60;

        // joining predicates
        Predicate<Person> predicate = greaterThanEighteen.and(lessThanSixty);

        boolean eligible = isPersonEligibleForMembership(person, predicate);
        System.out.println("Person is eligible for membership: " + eligible);
    }
}

```

```java
Person person = new Person("Alex", 23);

// Created a predicate. It returns true if age is greater than 18.
Predicate<Person> greaterThanEighteen = (p) -> p.age > 18;
// Created a predicate. It returns true if age is less than 60.
Predicate<Person> lessThanSixty = (p) -> p.age < 60;

// joining predicates
Predicate<Person> predicate = greaterThanEighteen.or(lessThanSixty);

boolean eligible = isPersonEligibleForMembership(person, predicate);
System.out.println("Person is eligible for membership: " + eligible);
```



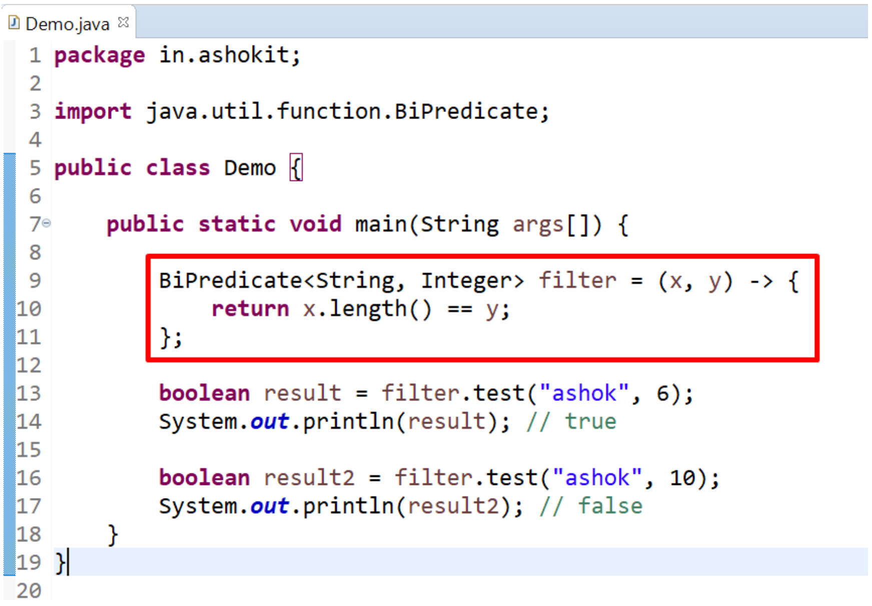
Demo.java

```
package in.ashokit;

import java.util.function.Predicate;

public class Demo {

    static boolean isPersonEligibleForMembership(Person person, Predicate<Person> predicat
        return predicate.test(person);
    }

    public static void main(String args[]) {
        Person person = new Person("Alex", 23);

        // Created a predicate. It returns true if age is greater than 18.
        Predicate<Person> greaterThanEighteen = (p) -> p.age > 18;
        // Created a predicate. It returns true if age is less than 60.
        Predicate<Person> lessThanSixty = (p) -> p.age < 60;

        // joining predicates
        Predicate<Person> predicate = greaterThanEighteen.or(lessThanSixty);

        boolean eligible = isPersonEligibleForMembership(person, predicate);
        System.out.println("Person is eligible for membership: " + eligible);
    }
}
```

BiPredicate interface

-> The Predicate<T> takes only one parameter and returns the result. Now suppose we have a requirement where we need to send two parameters (i.e person object and min age to vote) and then return the result. Here, we can use BiPredicate<T, T>.

-> The BiPredicate<T, T> has a functional method test(Object, Object) . It takes in two parameters and returns a boolean value.



Demo.java

```
package in.ashokit;

import java.util.function.BiPredicate;

public class Demo {

    public static void main(String args[]) {

        BiPredicate<String, Integer> filter = (x, y) -> {
            return x.length() == y;
        };

        boolean result = filter.test("ashok", 6);
        System.out.println(result); // true

        boolean result2 = filter.test("ashok", 10);
        System.out.println(result2); // false
    }
}
```

Supplier

-> Supplier is an interface that does not take in any argument but produces a value when the get() function is invoked. Suppliers are useful when we don’t need to supply any value and obtain a result at the same time.

-> The Supplier<T> interface supplies a result of type T.

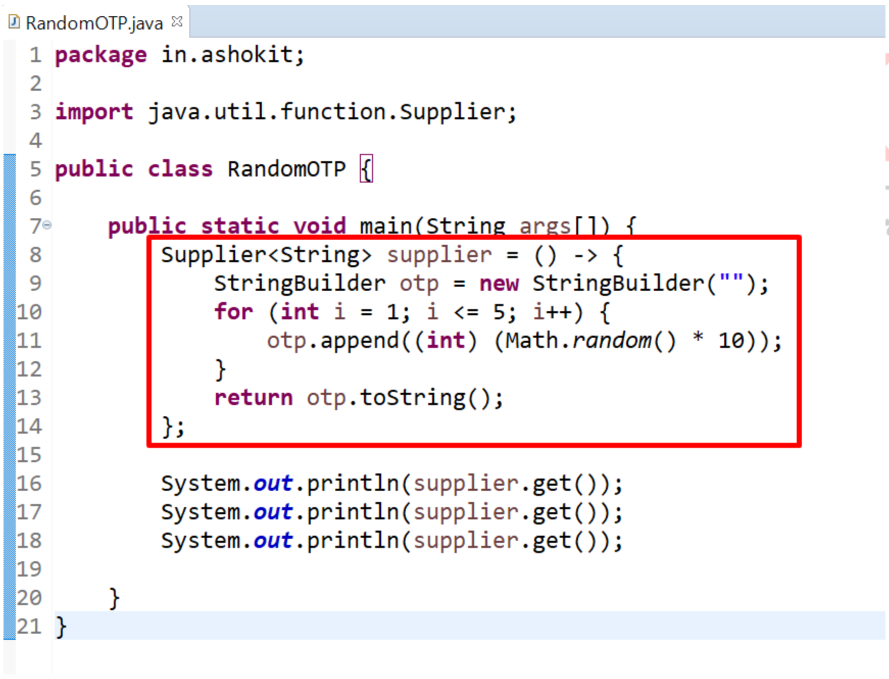
-> It is a predefined functional interface

-> It contains only one abstract method i.e get ( )

-> Supplier will only returns the value R

```
interface Supplier {
    R get();
}
```

// Java program to generate OTP using Supplier interface



RandomOTP.java

```java
package in.ashokit;

import java.util.function.Supplier;

public class RandomOTP {

    public static void main(String args[]) {
        Supplier<String> supplier = () -> {
            StringBuilder otp = new StringBuilder("");
            for (int i = 1; i <= 5; i++) {
                otp.append((int) (Math.random() * 10));
            }
            return otp.toString();
        };

        System.out.println(supplier.get());
        System.out.println(supplier.get());
        System.out.println(supplier.get());

    }
}
```

Consumer

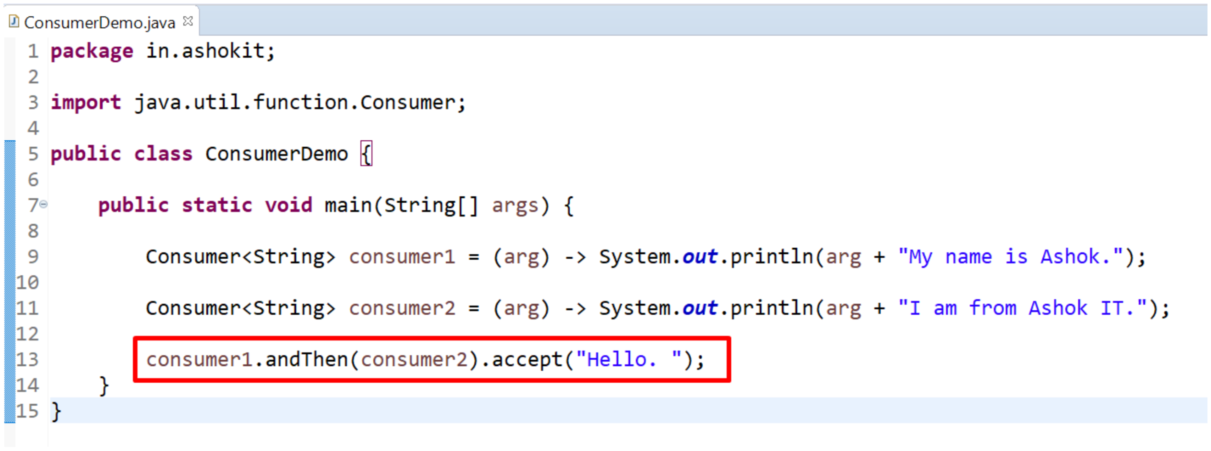
-> It is a predefined functional interface

-> It contains one abstract method i.e accept (T t)

-> Consumer will accept values and will perform operation but it won't return any value

-> A consumer can be used in all contexts where an object needs to be consumed. taken as input, and some operation is performed on the object without returning any result.

```
interface Consumer {
  void accet(T t);
}
```


ConsumerDemo.java

```java
package in.ashokit;

import java.util.function.Consumer;

public class ConsumerDemo {

    public static void main(String[] args) {

        Consumer<String> consumer1 = (arg) -> System.out.println(arg + "My name is Ashok.");

        Consumer<String> consumer2 = (arg) -> System.out.println(arg + "I am from Ashok IT.");

        consumer1.andThen(consumer2).accept("Hello. ");
    }
}
```

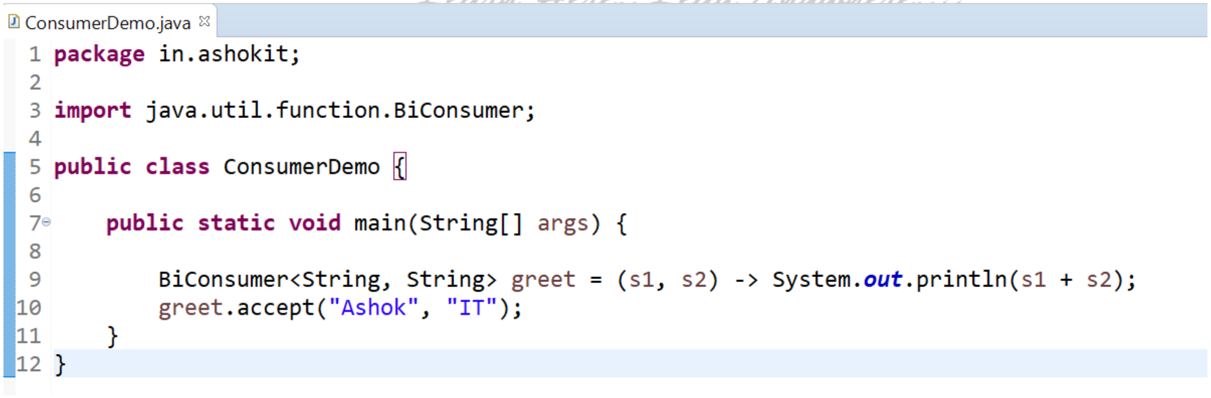
BiConsumer<T,U>

This interface takes two parameters and returns nothing.

T - the type of the first argument to the operation

U - the type of the second argument to the operation.

This interface has the same methods as present in the Consumer<T> interface.


ConsumerDemo.java

```java
package in.ashokit;

import java.util.function.BiConsumer;

public class ConsumerDemo {

    public static void main(String[] args) {

        BiConsumer<String, String> greet = (s1, s2) -> System.out.println(s1 + s2);
        greet.accept("Ashok", "IT");
    }
}
```

Function

-> Function is a category of functional interfaces that takes an object of type T and returns an object of type R.

-> Until now, the functional interfaces that we've discussed have either not taken any argument (Supplier), not returned any value (Consumer), or returned only a boolean (Predicate).

-> Function interfaces are very useful as we can specify the type of input and output.

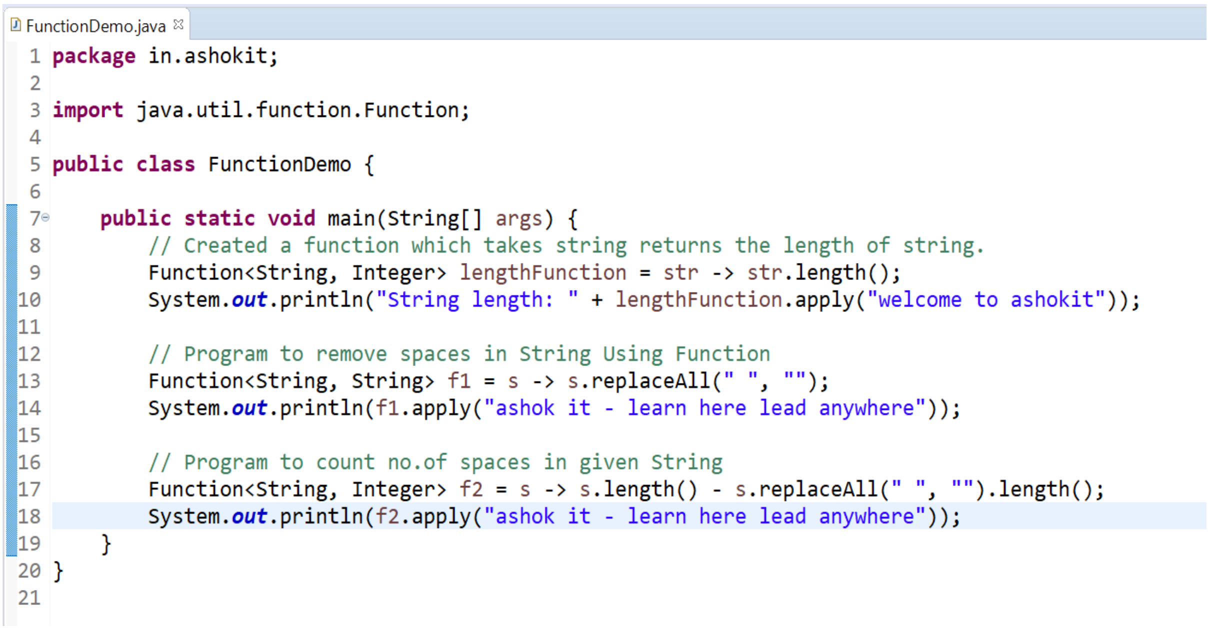
-> It is a predefined functional interface

-> It is having one abstract method that is "apply ()"

-> Function can return any type of value whereas Predicate can return only boolean value

Syntax:

```
interface Function {
   R apply(T t);
}
```



FunctionDemo.java

```java
package in.ashokit;

import java.util.function.Function;

public class FunctionDemo {

    public static void main(String[] args) {
        // Created a function which takes string returns the length of string.
        Function<String, Integer> lengthFunction = str -> str.length();
        System.out.println("String length: " + lengthFunction.apply("welcome to ashokit"));

        // Program to remove spaces in String Using Function
        Function<String, String> f1 = s -> s.replaceAll(" ", "");
        System.out.println(f1.apply("ashok it - learn here lead anywhere"));

        // Program to count no.of spaces in given String
        Function<String, Integer> f2 = s -> s.length() - s.replaceAll(" ", "").length();
        System.out.println(f2.apply("ashok it - learn here lead anywhere"));
    }
}

```

<u>Function Chaining</u>

-> To join one function with another function we will use Function Chaining

Assume that f1 & f2 are 2 functions

f1.andThen(f2) ----> First f1 will be applied and then followed by f2

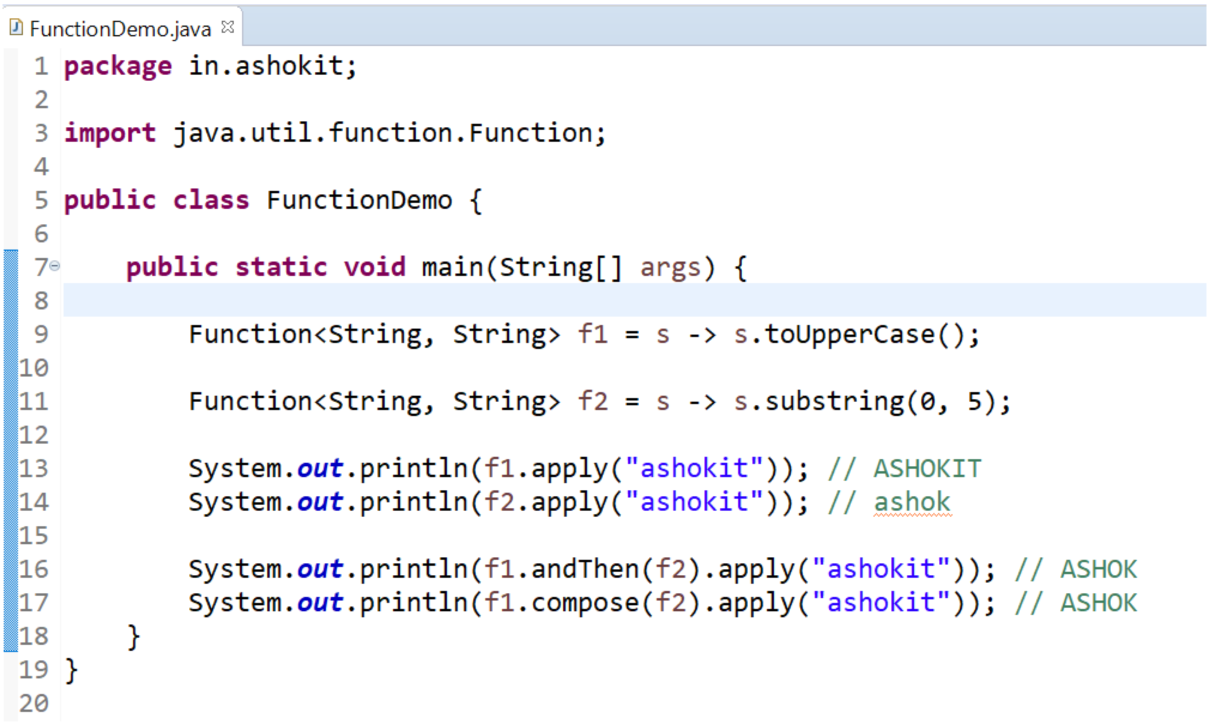
f1.compose(f2) ----> First f2 will be applied and then followed by f1

<u>compose(Function<? super V, ? extends T> before)</u>

Returns a composed function that first applies the function provided as a parameter on the input, and then applies the function on which it is called, to the result.

<u>andThen(Function<? super R,? extends V> after)</u>

This method returns a composed function that first applies the function on which it is called on the input, and then applies the function provided as parameter, to the result.



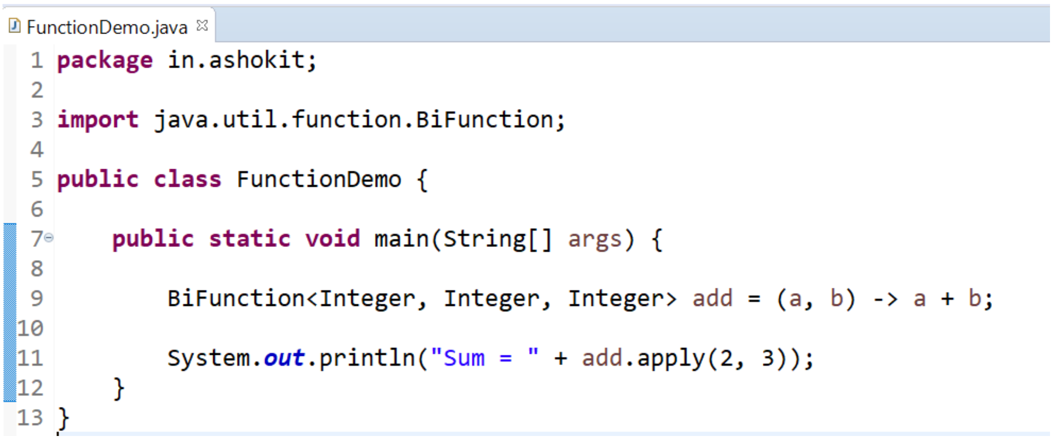
FunctionDemo.java

```java
package in.ashokit;

import java.util.function.Function;

public class FunctionDemo {

    public static void main(String[] args) {

        Function<String, String> f1 = s -> s.toUpperCase();

        Function<String, String> f2 = s -> s.substring(0, 5);

        System.out.println(f1.apply("ashokit")); // ASHOKIT
        System.out.println(f2.apply("ashokit")); // ashok

        System.out.println(f1.andThen(f2).apply("ashokit")); // ASHOK
        System.out.println(f1.compose(f2).apply("ashokit")); // ASHOK
    }
}

```

BiFunction<T,U,R>

The BiFunction<T, U, R> is similar to Function<T, R> interface; the only difference is that the BiFunction interface takes in two parameters and returns an output.

In the below example, we will create a BiFunction that takes two numbers as input and returns their sum.



FunctionDemo.java

```java
package in.ashokit;

import java.util.function.BiFunction;

public class FunctionDemo {

    public static void main(String[] args) {

        BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

        System.out.println("Sum = " + add.apply(2, 3));
    }
}
```

Method Reference :: Operator

-> Method references, as the name suggests are the references to a method. They are similar to object references. As we can have reference to an object, we can have reference to a method as well.

-> Similar to an object reference, we can now pass behaviour as parameters. But, you might be wondering what the difference between a method reference and lambda expressions is. There is no difference. Method references are shortened versions of lambda expressions that call a specific method.

Let's say you have a Consumer as defined below

Consumer<String> consumer = s -> System.out.println(s);

This can be written as below

Consumer<String> consumer = System.out::println;

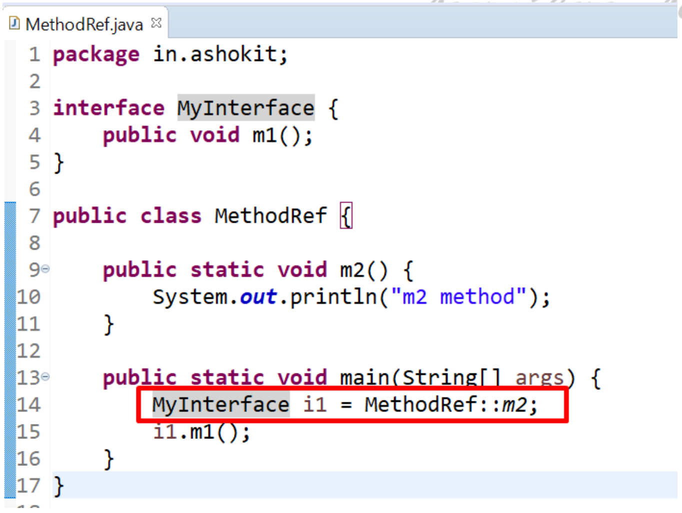
Consider we have a Function<T,R> functional interface as defined below:

Function<Person, Integer> function = p -> p.getAge();

This can be written as:

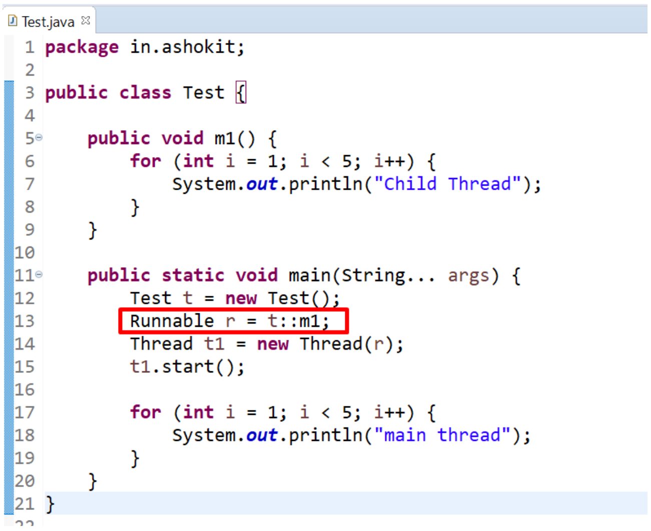
Function<Person, Integer> function = Person::getAge;

// Java program with static method reference


MethodRef.java

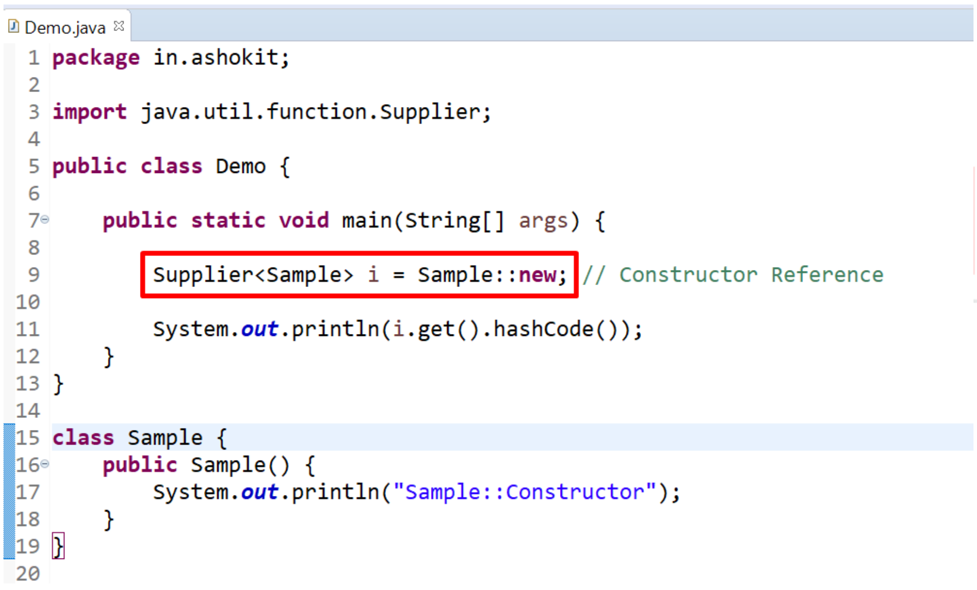
```java
package in.ashokit;

interface MyInterface {
    public void m1();
}

public class MethodRef {

    public static void m2() {
        System.out.println("m2 method");
    }

    public static void main(String[] args) {
        MyInterface i1 = MethodRef::m2;
        i1.m1();
    }
}
```

// java program with instance method reference



```
package in.ashokit;

public class Test {

    public void m1() {
        for (int i = 1; i < 5; i++) {
            System.out.println("Child Thread");
        }
    }

    public static void main(String... args) {
        Test t = new Test();
        Runnable r = t::m1;
        Thread t1 = new Thread(r);
        t1.start();

        for (int i = 1; i < 5; i++) {
            System.out.println("main thread");
        }
    }
}
```

// java program with Constructor Reference



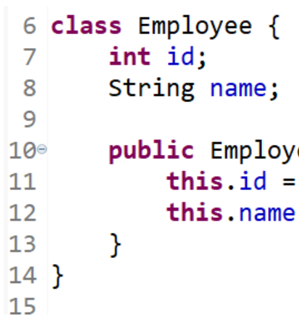
```
package in.ashokit;

import java.util.function.Supplier;

public class Demo {

    public static void main(String[] args) {

        Supplier<Sample> i = Sample::new; // Constructor Reference

        System.out.println(i.get().hashCode());
    }
}

class Sample {
    public Sample() {
        System.out.println("Sample::Constructor");
    }
}
```

<u>Java forEach ( ) method</u>

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

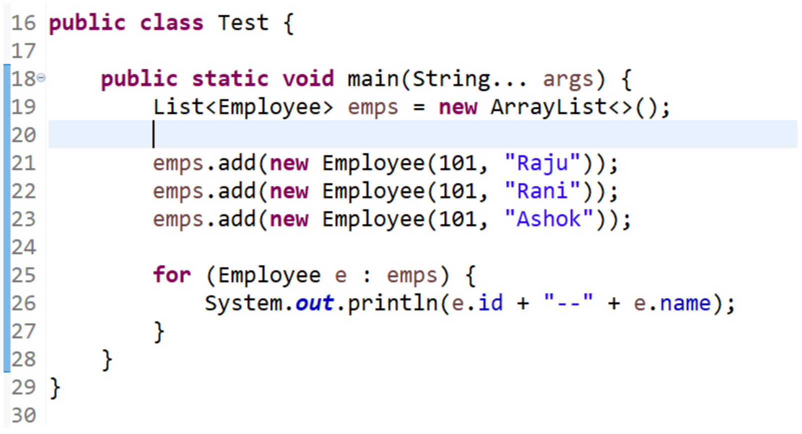
-> It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

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



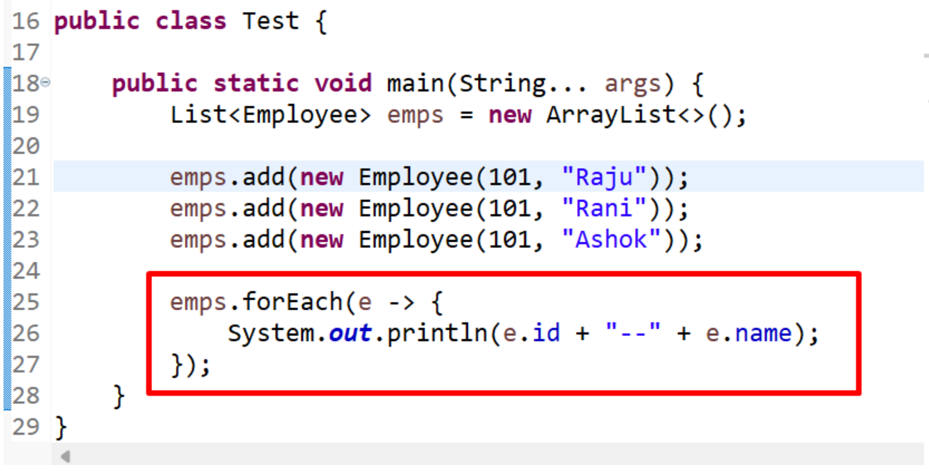
```java
class Employee {
    int id;
    String name;

    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

```

// Java program to print list data with for-each loop



```java
public class Test {

    public static void main(String... args) {
        List<Employee> emps = new ArrayList<>();

        emps.add(new Employee(101, "Raju"));
        emps.add(new Employee(101, "Rani"));
        emps.add(new Employee(101, "Ashok"));

        for (Employee e : emps) {
            System.out.println(e.id + "--" + e.name);
        }
    }
}

```

// Java program to print list data with foreach method



```java
public class Test {

    public static void main(String... args) {
        List<Employee> emps = new ArrayList<>();

        emps.add(new Employee(101, "Raju"));
        emps.add(new Employee(101, "Rani"));
        emps.add(new Employee(101, "Ashok"));

        emps.forEach(e -> {
            System.out.println(e.id + "--" + e.name);
        });
    }
}
```

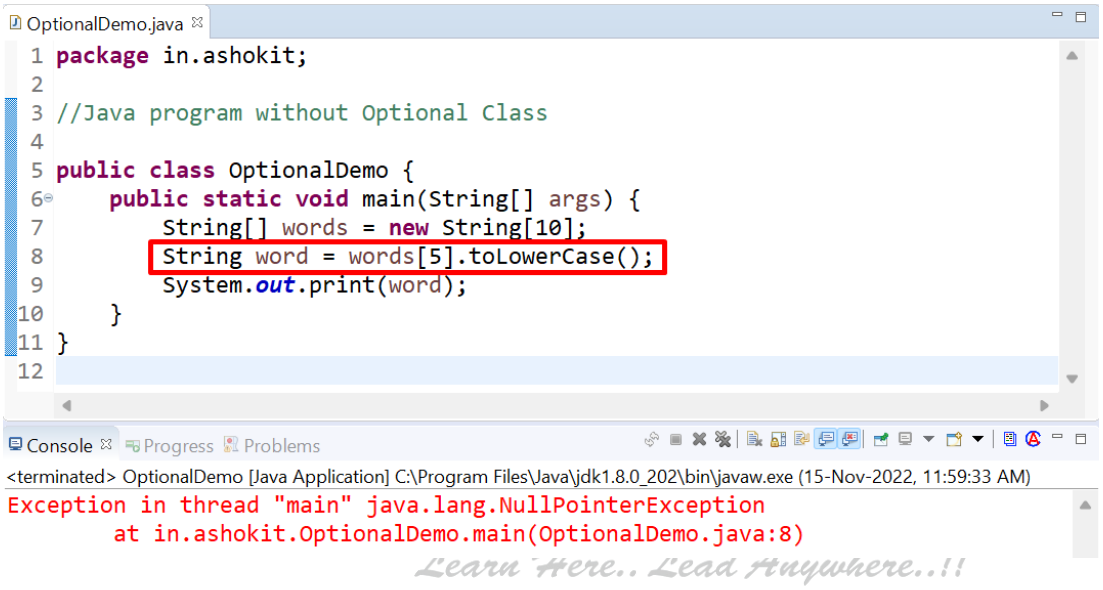
Java 8 Optional Class

-> Every Java Programmer is familiar with NullPointerException. It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in java.util package.

-> Optional class help in writing a neat code without using too many null checks.

-> By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable.

// in below java program we will get NullPointerException

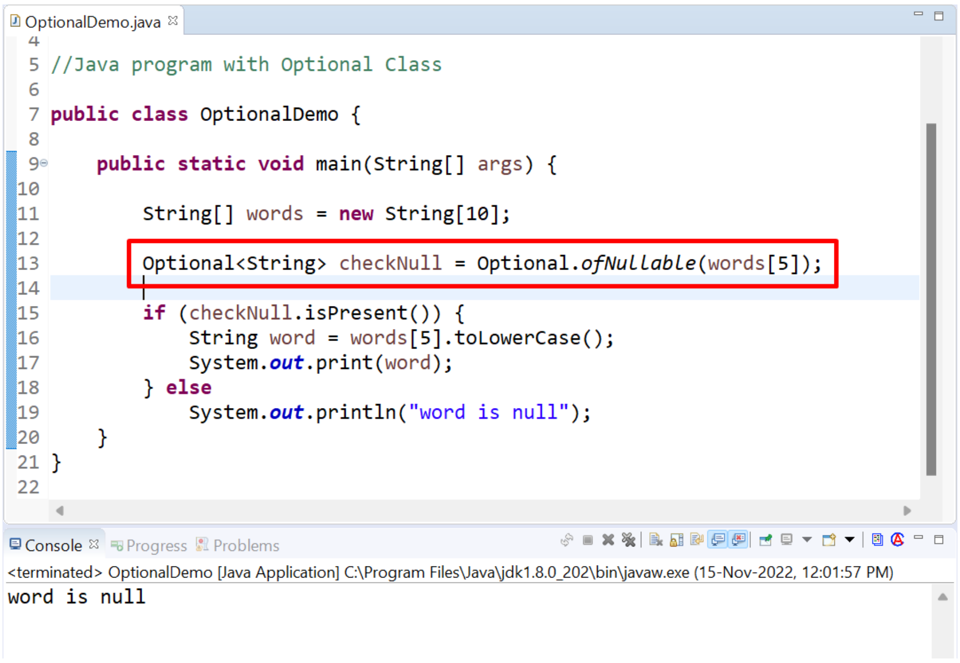


OptionalDemo.java

```java
package in.ashokit;

//Java program without Optional Class

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

```
<terminated> OptionalDemo [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (15-Nov-2022, 11:59:33 AM)
Exception in thread "main" java.lang.NullPointerException
	at in.ashokit.OptionalDemo.main(OptionalDemo.java:8)
```

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



OptionalDemo.java

```java
//Java program with Optional Class

public class OptionalDemo {

    public static void main(String[] args) {

        String[] words = new String[10];

        Optional<String> checkNull = Optional.ofNullable(words[5]);

        if (checkNull.isPresent()) {
            String word = words[5].toLowerCase();
            System.out.print(word);
        } else
            System.out.println("word is null");
    }
}
```

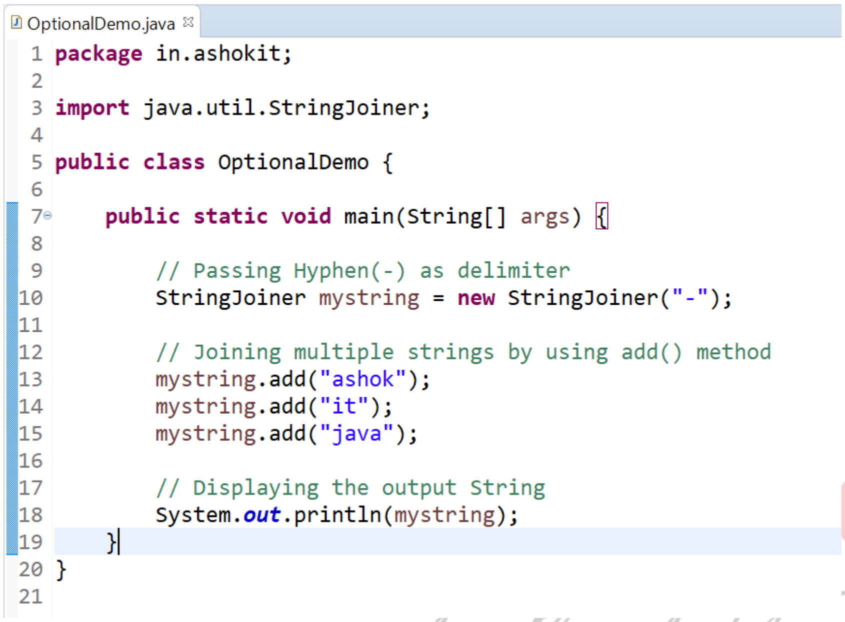
```
<terminated> OptionalDemo [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (15-Nov-2022, 12:01:57 PM)
word is null
```

-> Optional is a container object which may or may not contain a non-null value. You must import java.util package to use this class. If a value is present, isPresent() will return true and get() will return the value.

Java 8 StringJoiner

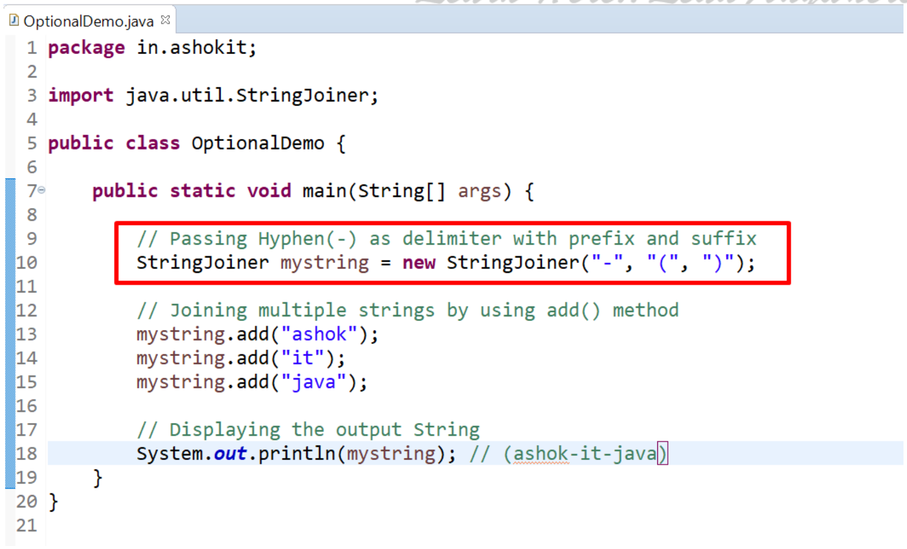
-> In java 8, a new class StringJoiner is introduced in the java.util package.

-> Using this class we can join more than one strings with the specified delimiter, we can also provide prefix and suffix to the final string while joining multiple strings.



OptionalDemo.java

```java
package in.ashokit;

import java.util.StringJoiner;

public class OptionalDemo {

    public static void main(String[] args) {

        // Passing Hyphen(-) as delimiter
        StringJoiner mystring = new StringJoiner("-");

        // Joining multiple strings by using add() method
        mystring.add("ashok");
        mystring.add("it");
        mystring.add("java");

        // Displaying the output String
        System.out.println(mystring);
    }
}
```



OptionalDemo.java

```java
package in.ashokit;

import java.util.StringJoiner;

public class OptionalDemo {

    public static void main(String[] args) {

        // Passing Hyphen(-) as delimiter with prefix and suffix
        StringJoiner mystring = new StringJoiner("-", "(", ")");

        // Joining multiple strings by using add() method
        mystring.add("ashok");
        mystring.add("it");
        mystring.add("java");

        // Displaying the output String
        System.out.println(mystring); // (ashok-it-java)
    }
}
```

Stream API

-> Stream API introduced in java 8

-> Collections are used to Store the data

-> Stream is used to process the data

Note: Collections & Streams both are not same.

-> The addition of stream api was one of the major features added to java8.

-> A stream in java can be defined as a sequence of elements from a source that supports aggregate operations on them.

-> The source here refers to collection or array that provide data to stream

Few Important points about Streams

1) Stream is not a data structure. It is a bunch of operations applied to a source. The source can be a collection, array or i/o channels.

2) Streams don't change the original data structure

3) There can be zero or more intermediate operations that transforms a stream into another stream. Each intermediate operation is lazily executed

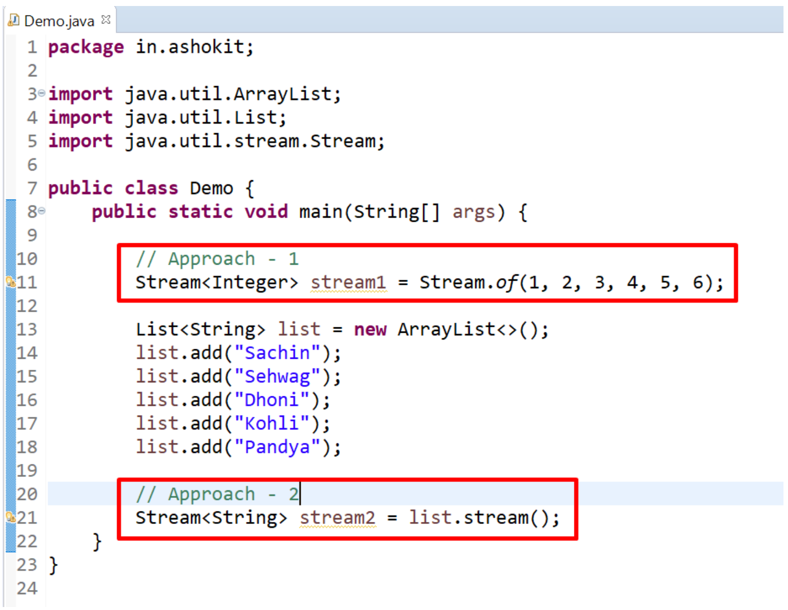
5) Terminal operations produce the result of the stream.

Stream Creation

-> In java 8 we can create stream in 2 ways

1) Stream.of(v1, v2, v3...)

2) stream() method



Demo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class Demo {
    public static void main(String[] args) {

        // Approach - 1
        Stream<Integer> stream1 = Stream.of(1, 2, 3, 4, 5, 6);

        List<String> list = new ArrayList<>();
        list.add("Sachin");
        list.add("Sehwag");
        list.add("Dhoni");
        list.add("Kohli");
        list.add("Pandya");

        // Approach - 2
        Stream<String> stream2 = list.stream();
    }
}
```

-> Stream API provided several methods to perform operations.

-> The methods which are provided by stream API can be categorized into 2 types

1) Intermediate Operational Methods

2) Terminal Operational Methods

-> Intermediate operational methods will not produce any results. They usually accept functional interfaces as parameter and always returns new Stream.

Ex: filter() and map() etc...

-> Terminal Operational methods will take input and produce results.

Ex: count(), toArray() and collect()

Streams with Filtering

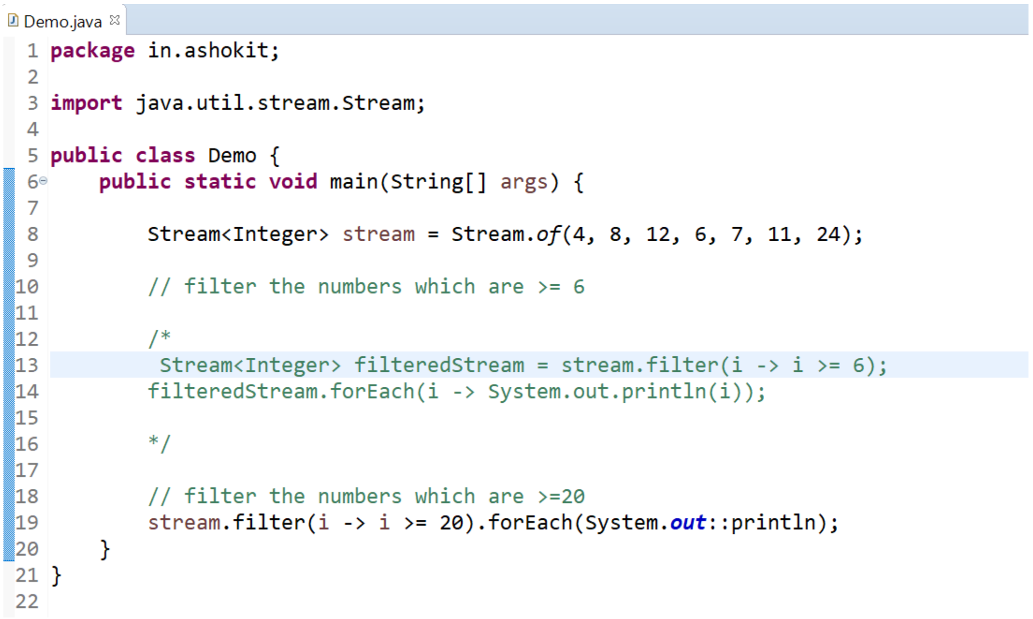
-> Stream interface having filter( ) method.

-> filter( ) method will take Predicate as input

-> Predicate is a functional interface which will take input and returns boolean value

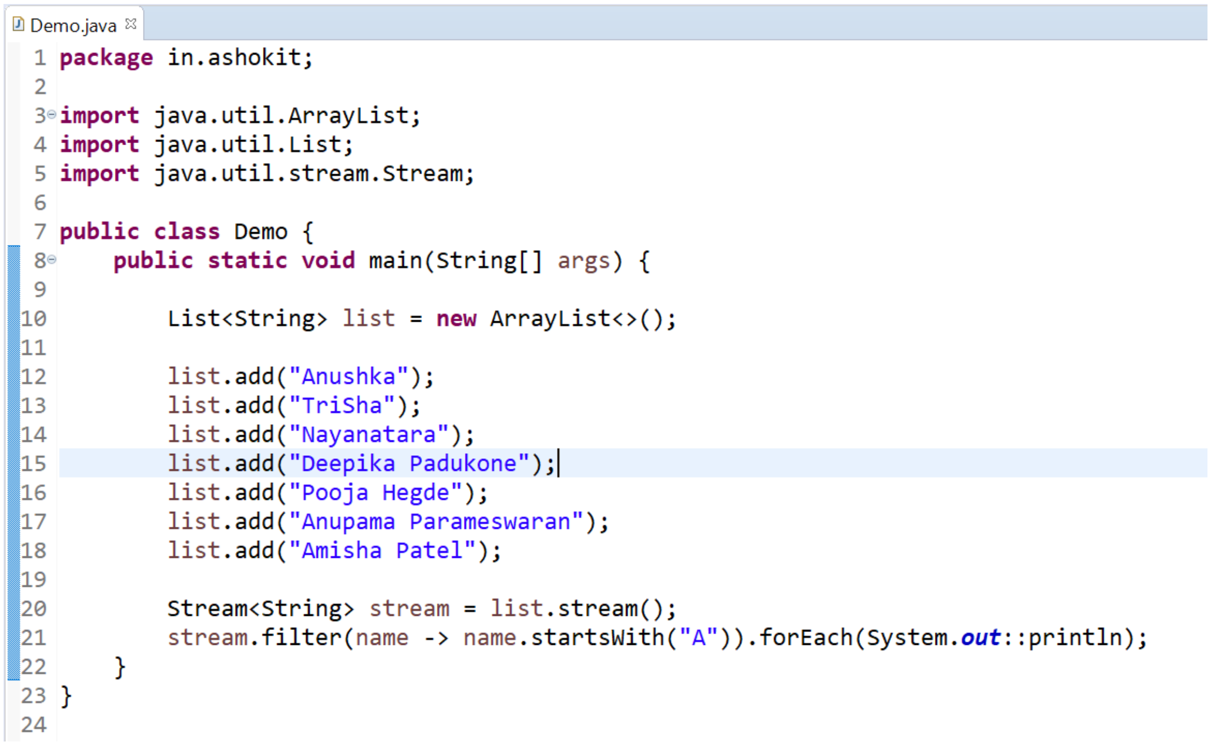
-> Predicate interface contains test( ) as abstract method. This method is used to execute lambda.

// Java program to filter the numbers which are > =20


Demo.java

```java
package in.ashokit;

import java.util.stream.Stream;

public class Demo {
    public static void main(String[] args) {

        Stream<Integer> stream = Stream.of(4, 8, 12, 6, 7, 11, 24);

        // filter the numbers which are >= 6

        /*
         Stream<Integer> filteredStream = stream.filter(i -> i >= 6);
        filteredStream.forEach(i -> System.out.println(i));

        */

        // filter the numbers which are >=20
        stream.filter(i -> i >= 20).forEach(System.out::println);
    }
}
```

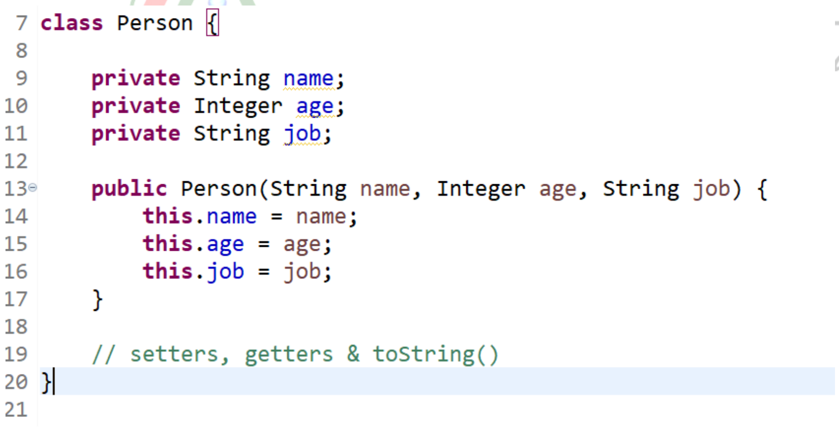
// Java program to filter the names which are starting with ”A”



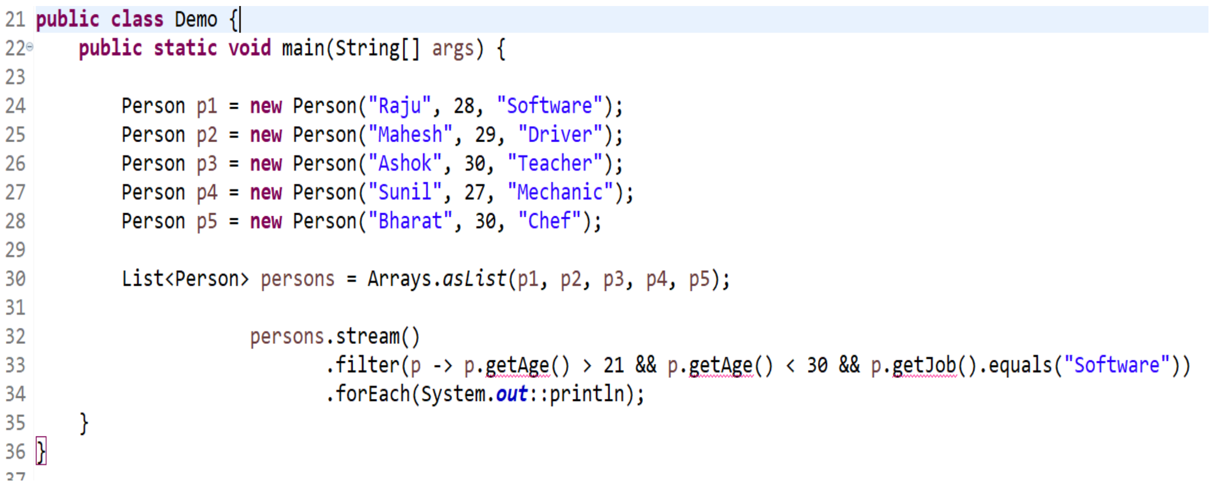
Demo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class Demo {
    public static void main(String[] args) {

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

        list.add("Anushka");
        list.add("TriSha");
        list.add("Nayanatara");
        list.add("Deepika Padukone");
        list.add("Pooja Hegde");
        list.add("Anupama Parameswaran");
        list.add("Amisha Patel");

        Stream<String> stream = list.stream();
        stream.filter(name -> name.startsWith("A")).forEach(System.out::println);
    }
}
```

// Java Program To Filter Person Objects



```java
class Person {

    private String name;
    private Integer age;
    private String job;

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

    // setters, getters & toString()
}
```

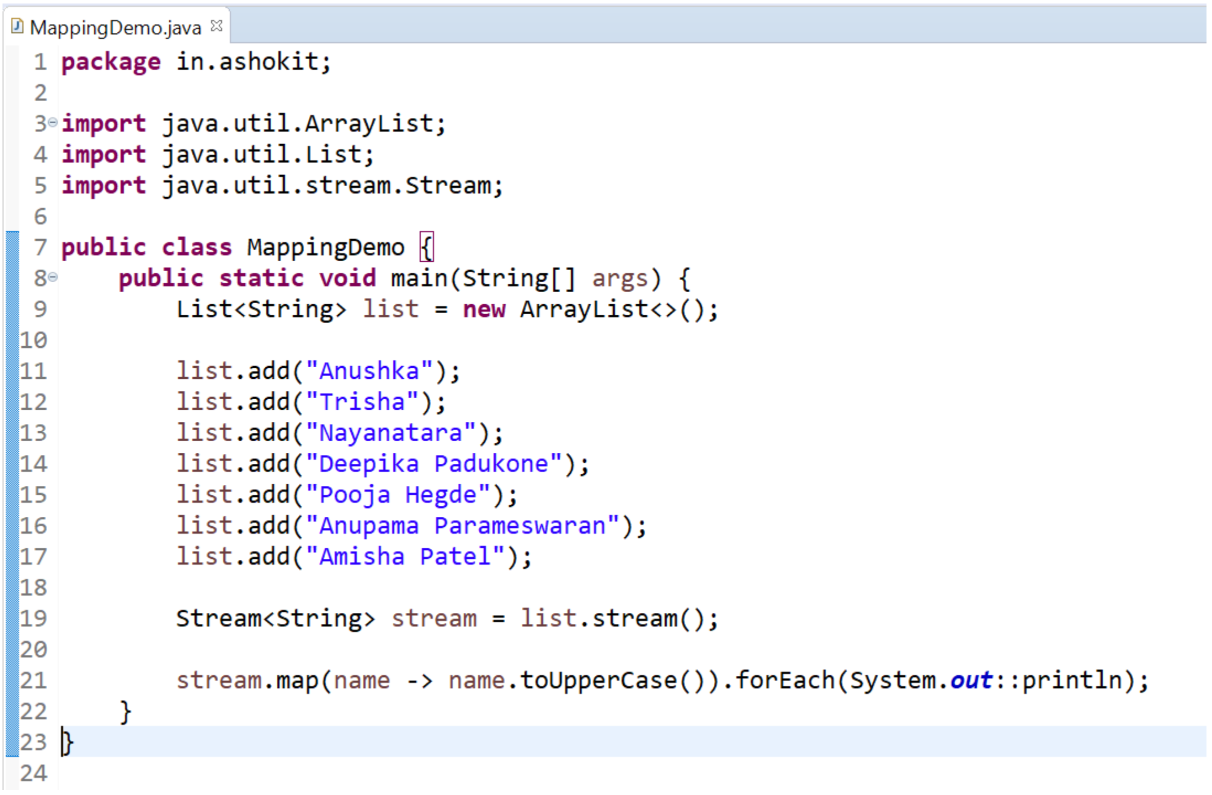


```java
public class Demo {
    public static void main(String[] args) {

        Person p1 = new Person("Raju", 28, "Software");
        Person p2 = new Person("Mahesh", 29, "Driver");
        Person p3 = new Person("Ashok", 30, "Teacher");
        Person p4 = new Person("Sunil", 27, "Mechanic");
        Person p5 = new Person("Bharat", 30, "Chef");

        List<Person> persons = Arrays.asList(p1, p2, p3, p4, p5);

                    persons.stream()
                            .filter(p -> p.getAge() > 21 && p.getAge() < 30 && p.getJob().equals("Software"))
                            .forEach(System.out::println);
    }
}
```

Mapping Operations In Streams

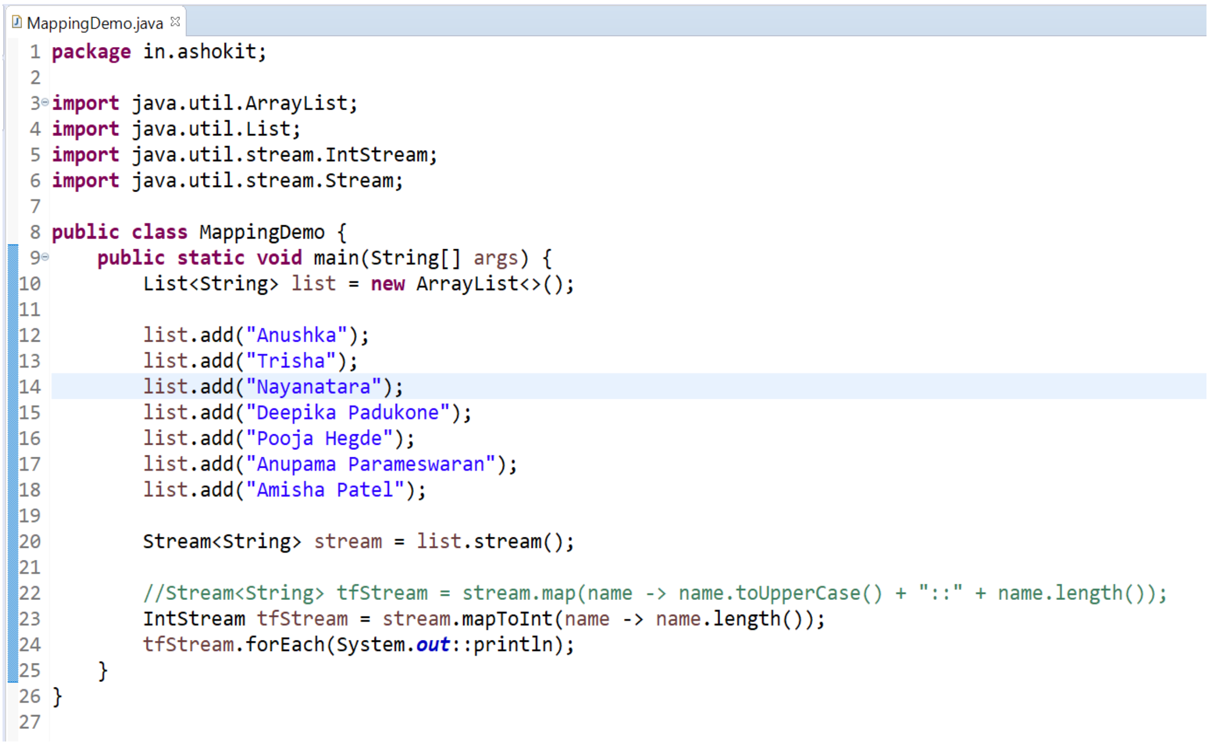
-> Mapping operations are belonging to Intermediate operations

-> Mapping operations are those operations that transform the elements of a stream and return a new stream with transformed elements.



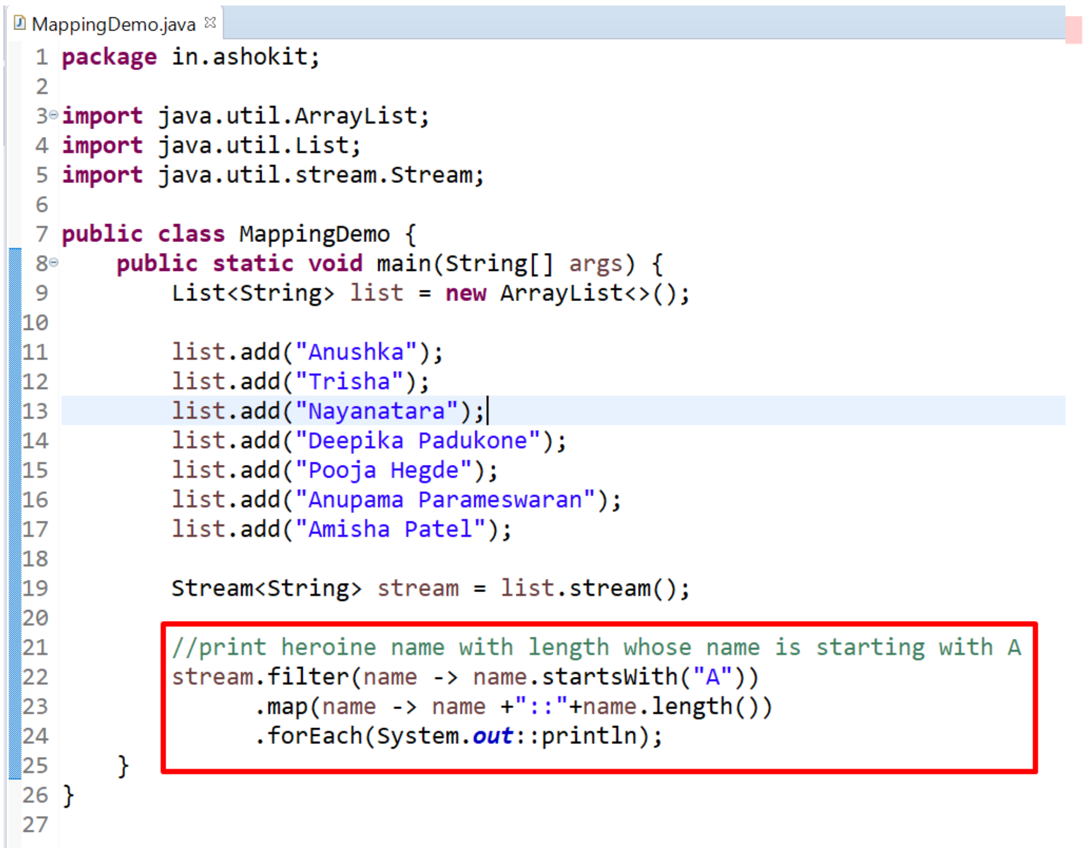
MappingDemo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

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

        list.add("Anushka");
        list.add("Trisha");
        list.add("Nayanatara");
        list.add("Deepika Padukone");
        list.add("Pooja Hegde");
        list.add("Anupama Parameswaran");
        list.add("Amisha Patel");

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

        stream.map(name -> name.toUpperCase()).forEach(System.out::println);
    }
}
```



MappingDemo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.IntStream;
import java.util.stream.Stream;

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

        list.add("Anushka");
        list.add("Trisha");
        list.add("Nayanatara");
        list.add("Deepika Padukone");
        list.add("Pooja Hegde");
        list.add("Anupama Parameswaran");
        list.add("Amisha Patel");

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

        //Stream<String> tfStream = stream.map(name -> name.toUpperCase() + "::" + name.length());
        IntStream tfStream = stream.mapToInt(name -> name.length());
        tfStream.forEach(System.out::println);
    }
}
```

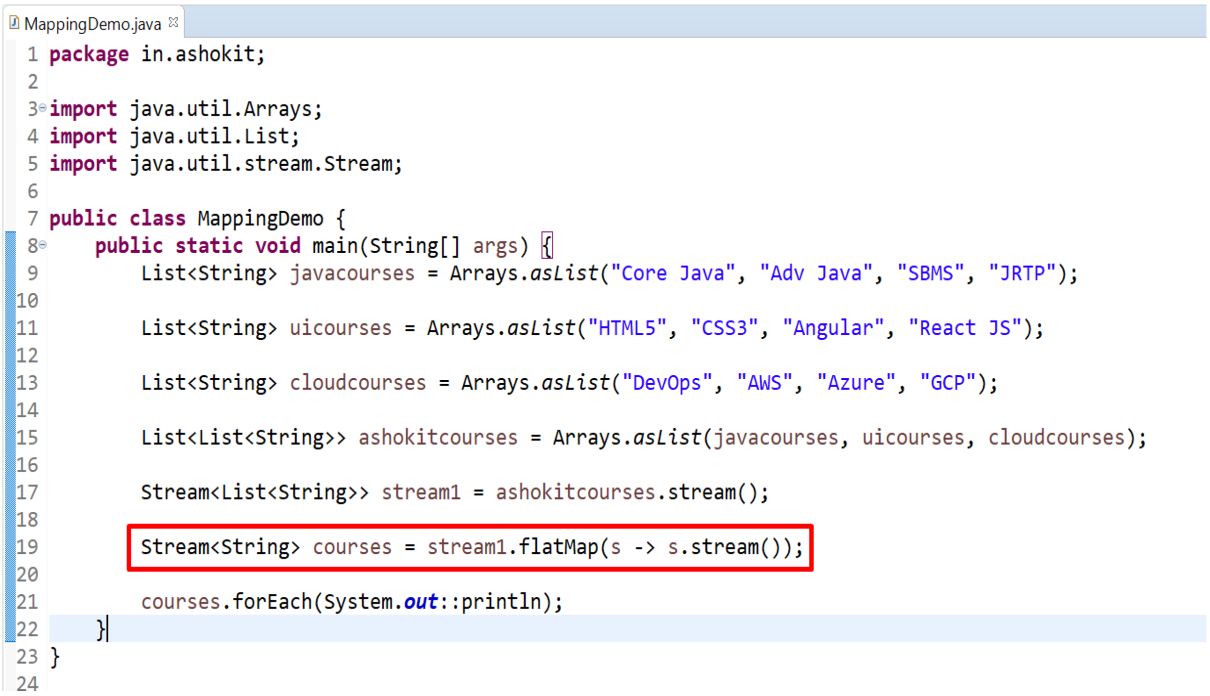


MappingDemo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

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

        list.add("Anushka");
        list.add("Trisha");
        list.add("Nayanatara");
        list.add("Deepika Padukone");
        list.add("Pooja Hegde");
        list.add("Anupama Parameswaran");
        list.add("Amisha Patel");

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

        //print heroine name with length whose name is starting with A
        stream.filter(name -> name.startsWith("A"))
              .map(name -> name +"::"+name.length())
              .forEach(System.out::println);
    }
}
```

```java
Person p1 = new Person("Raju", "Software", 6301921083l);
Person p2 = new Person("Gopi", "Chef", 7897678991l);
Person p3 = new Person("Mahesh", "PhotoGrapher", 6686210831l);
Person p4 = new Person("Sunil", "Driver", 47575721083l);
Person p5 = new Person("David", "Teacher", 5678921083l);
Person p6 = new Person("Ashok", "Software", 630192178l);

//Display person name with phn number who is doing Software job

List<Person> persons = Arrays.asList(p1, p2, p3, p4, p5, p6);

persons.stream()
        .filter(person -> person.getJob().equals("Software"))
        .map(person -> person.getName()+"::"+person.getPhno())
        .forEach(System.out::println);
```

-> When we have Collection inside another collection then to flaten that stream we will use flatMap( ) method.


MappingDemo.java

```java
package in.ashokit;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class MappingDemo {
    public static void main(String[] args) {
        List<String> javacourses = Arrays.asList("Core Java", "Adv Java", "SBMS", "JRTP");

        List<String> uicourses = Arrays.asList("HTML5", "CSS3", "Angular", "React JS");

        List<String> cloudcourses = Arrays.asList("DevOps", "AWS", "Azure", "GCP");

        List<List<String>> ashokitcourses = Arrays.asList(javacourses, uicourses, cloudcourses);

        Stream<List<String>> stream1 = ashokitcourses.stream();

        Stream<String> courses = stream1.flatMap(s -> s.stream());

        courses.forEach(System.out::println);
    }
}
```

<u>Slicing Operations In Streams</u>

-> distinct ( ) -> it is used to get unique elements from the Stream

-> limit (long maxSize) -> it is used to get number of elements from the stream based on given size

-> skip (long n) -> it is used to skip elements from starting to given length and returns remaining elements.

Note: The above 3 methods are intermediate methods only. They perform operation and returns new stream.

```
List<String> countries = new ArrayList<>();
countries.add("India");
countries.add("USA");
countries.add("UK");
countries.add("China");
countries.add("India");
countries.add("USA");


//Getting unique values from collection using distinct()
/*countries.stream()
         .distinct()
         .forEach(System.out::println);*/


//Getting specific no.of records from collection using limit()
/*countries.stream()
         .limit(4)
         .forEach(System.out::println);*/

//removing first N no.of recording
countries.stream()
         .skip(3)
         .forEach(System.out::println);
```

Matching Operations in the streams

-> Matching operations are terminal operations that are used to check if elements with certain are present in the stream or not.

-> There are mainley three matching functions available in the Stream. these are

- anyMatch ( )
- allMatch( )
- noneMatch( )

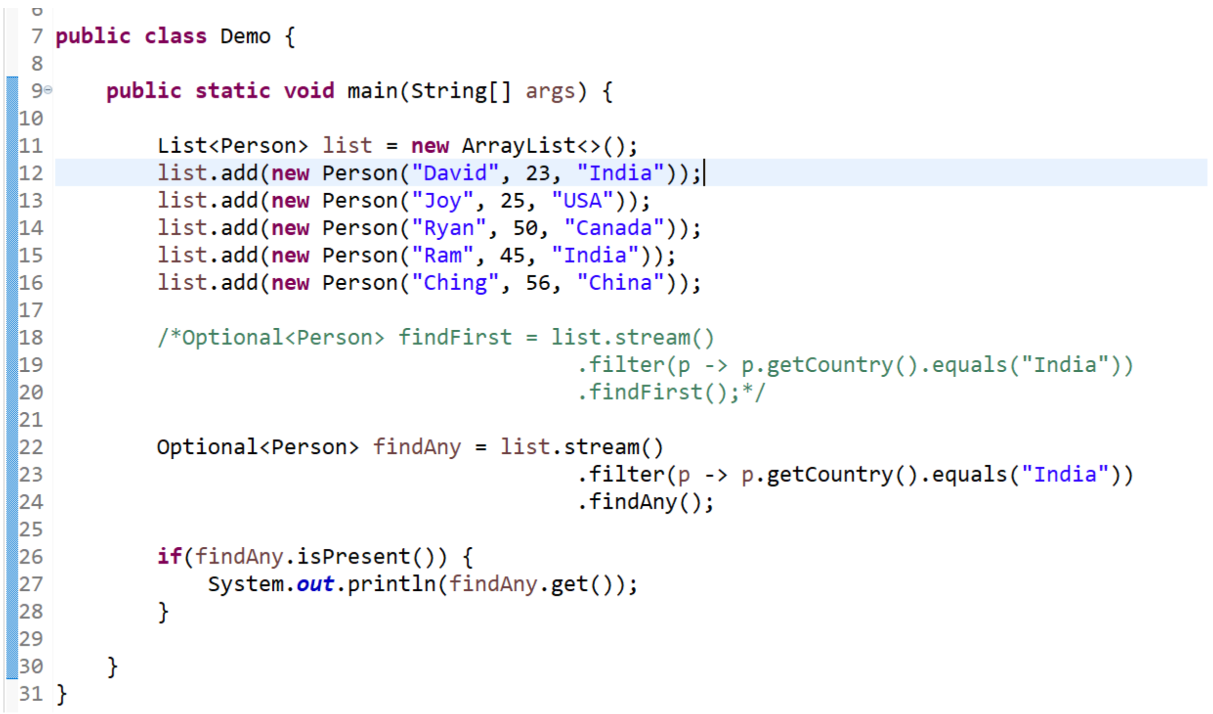
```
public class Demo {

    public static void main(String[] args) {

        List<Person> list = new ArrayList<>();
        list.add(new Person("David", 23, "India"));
        list.add(new Person("Joy", 25, "USA"));
        list.add(new Person("Ryan", 50, "Canada"));
        list.add(new Person("Ram", 45, "India"));
        list.add(new Person("Ching", 56, "China"));

        /*boolean isIndiansAvailable = list.stream().anyMatch(p -> p.getCountry().equals("India"));
        System.out.println("Is Indians Available in Collection :: "+ isIndiansAvailable);*/

        /*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 Germans are available :: "+ noneMatch);

    }
}
```

Finding Operations In Streams

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

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

- findFirst()
- findAny( )



```java
public class Demo {

    public static void main(String[] args) {

        List<Person> list = new ArrayList<>();
        list.add(new Person("David", 23, "India"));
        list.add(new Person("Joy", 25, "USA"));
        list.add(new Person("Ryan", 50, "Canada"));
        list.add(new Person("Ram", 45, "India"));
        list.add(new Person("Ching", 56, "China"));

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

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

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

    }
}
```

Collectors In Streams

-> Collectors operations are used to collect from Streams

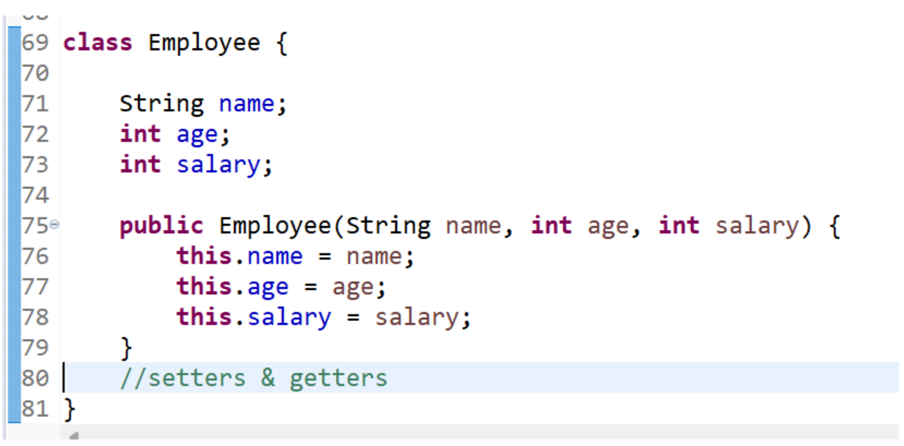
-> We are having below methods to perform Collectors operations

Collectors.toList()

Collectors.toSet()

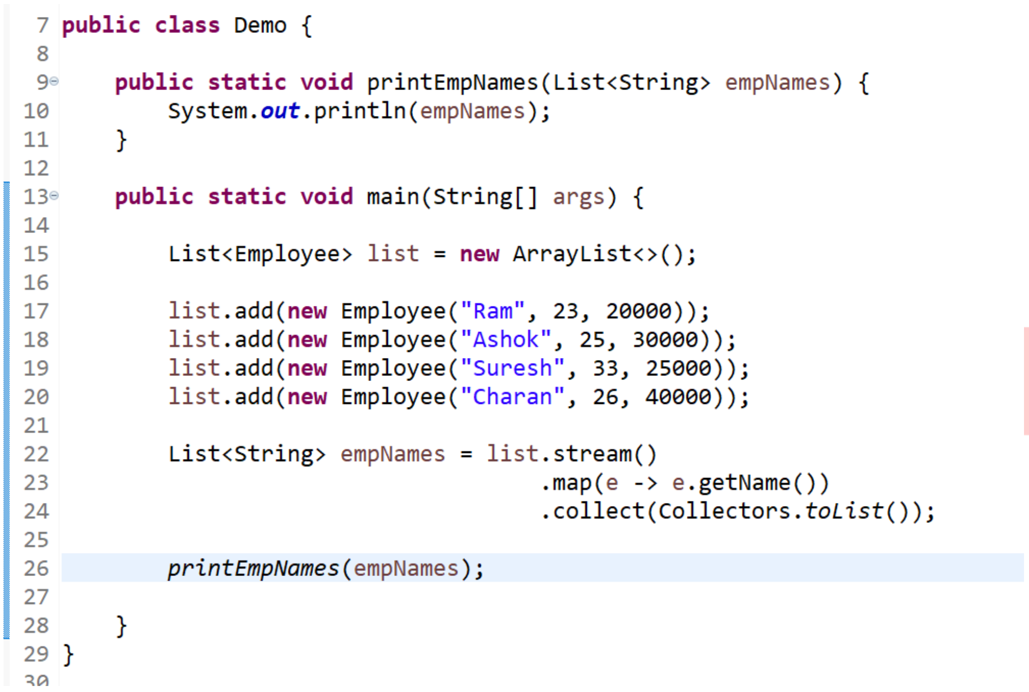
Collectors.toMap()

Collectors.toCollection() etc..



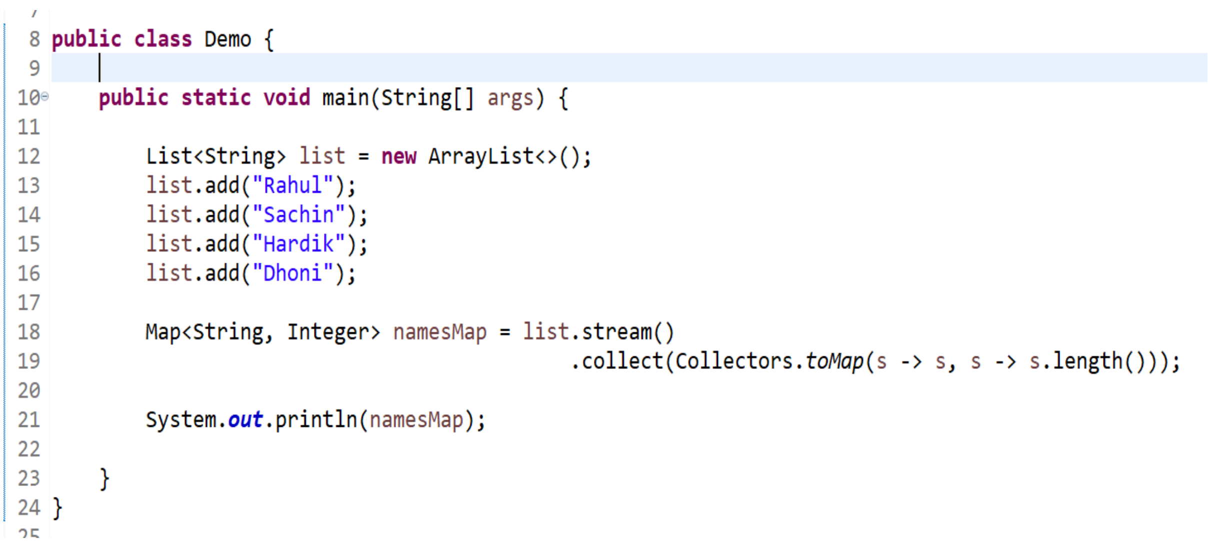
```java
class Employee {

    String name;
    int age;
    int salary;

    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    //setters & getters
}
```

// java program on Collectors with toList( )



```java
public class Demo {

    public static void printEmpNames(List<String> empNames) {
        System.out.println(empNames);
    }

    public static void main(String[] args) {

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

        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());

        printEmpNames(empNames);

    }
}
```

// java program on Collectors with toMap( )



```java
public class Demo {

    public static void main(String[] args) {

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

    }
}
```

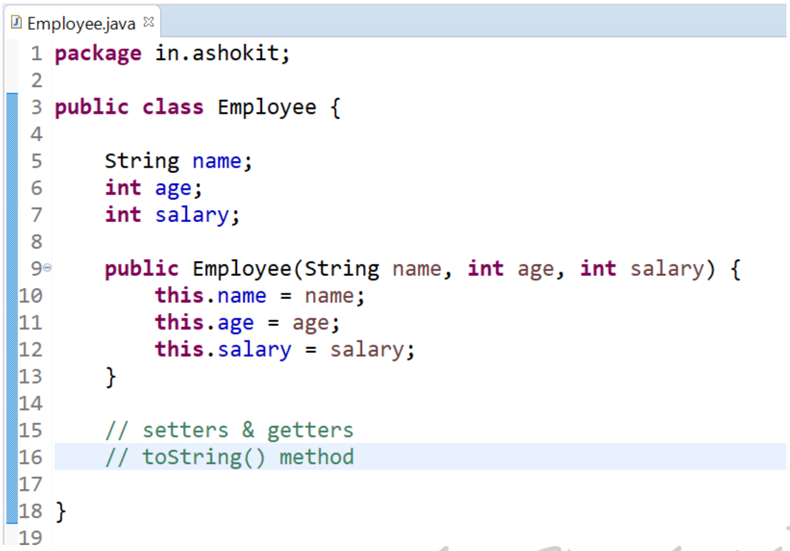
Stream API Interview Questions

1) Getting Min salary emp from collection

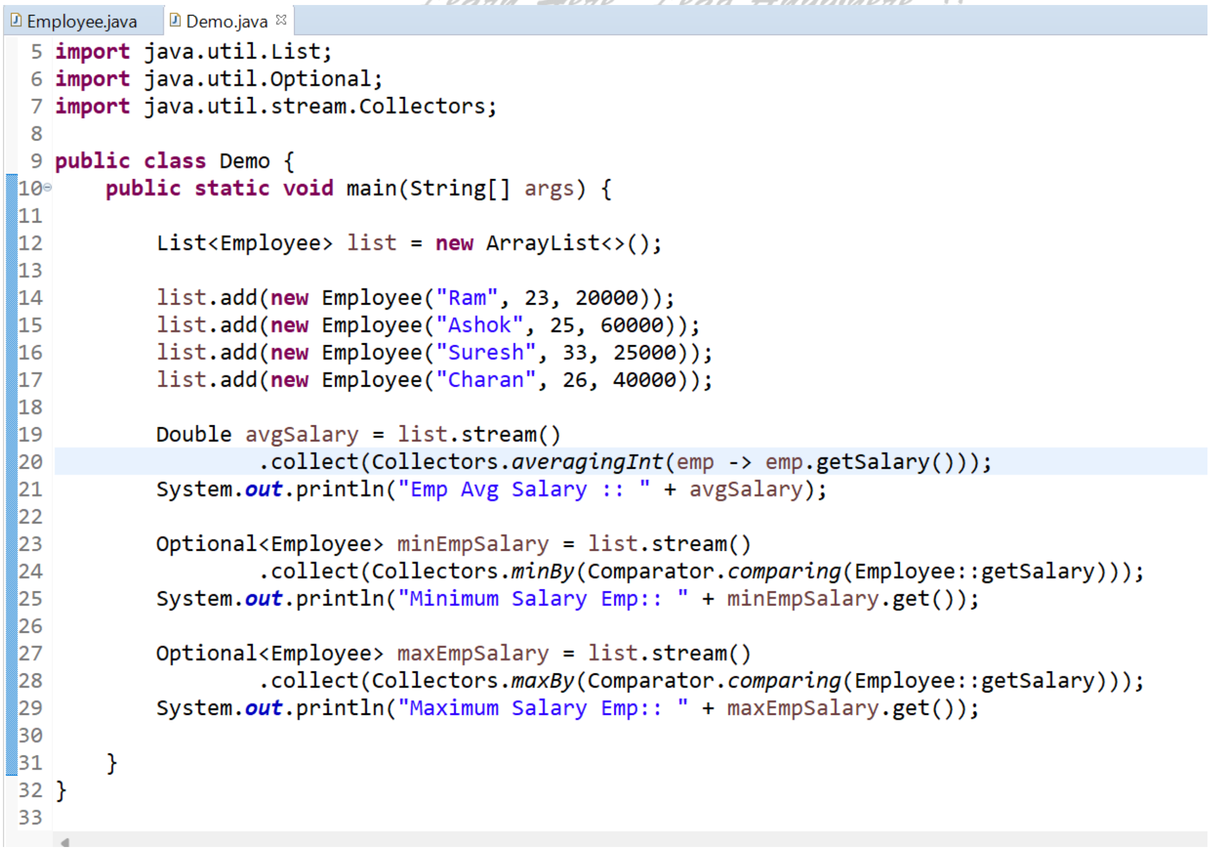
2) Getting Max salary emp from Collection

3) Getting Avg Salary of the employee

4) Group By

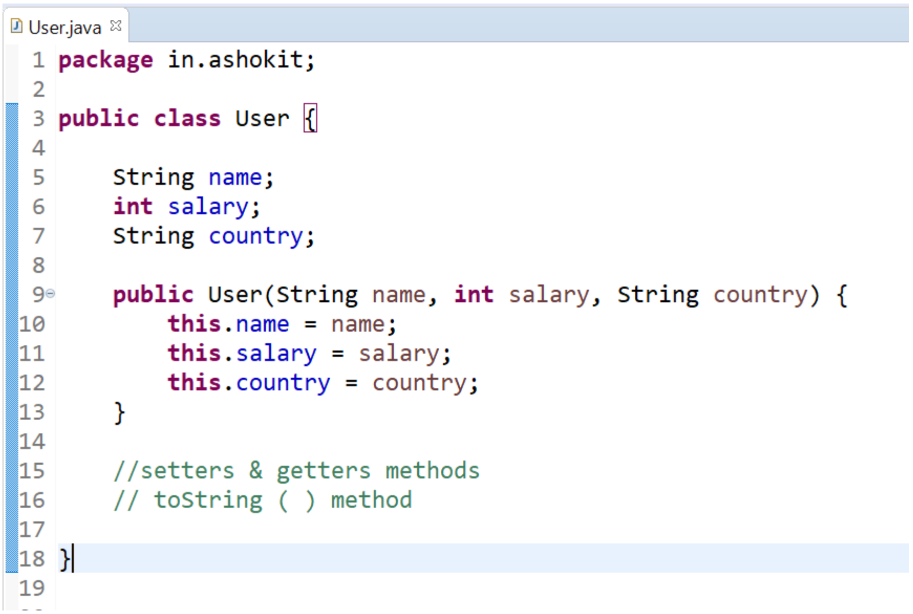

Employee.java

```java
package in.ashokit;

public class Employee {

    String name;
    int age;
    int salary;

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

    // setters & getters
    // toString() method

}
```

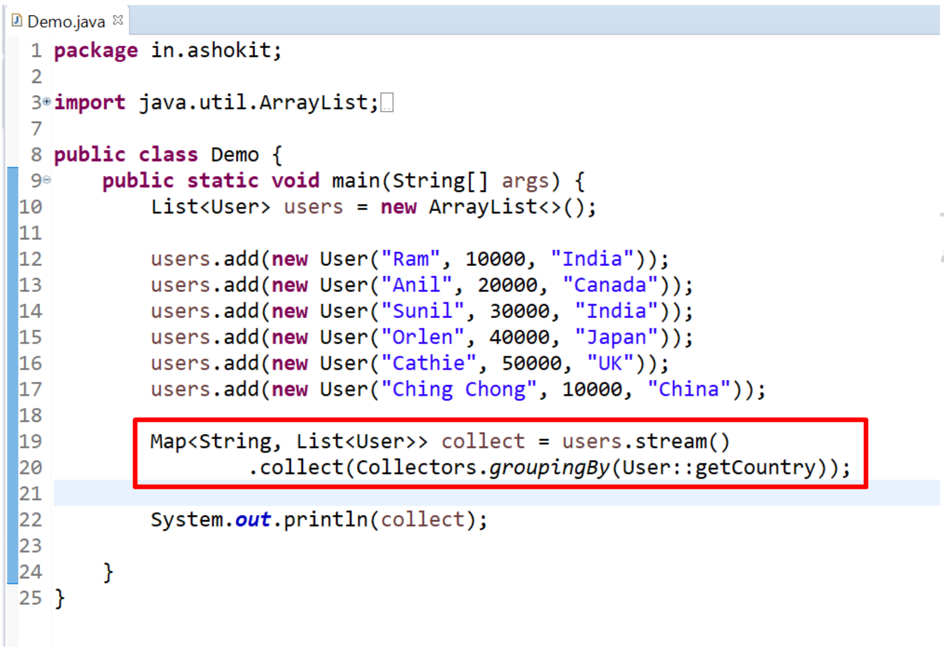

Employee.java Demo.java

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

public class Demo {
    public static void main(String[] args) {

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

        list.add(new Employee("Ram", 23, 20000));
        list.add(new Employee("Ashok", 25, 60000));
        list.add(new Employee("Suresh", 33, 25000));
        list.add(new Employee("Charan", 26, 40000));

        Double avgSalary = list.stream()
                .collect(Collectors.averagingInt(emp -> emp.getSalary()));
        System.out.println("Emp Avg Salary :: " + avgSalary);

        Optional<Employee> minEmpSalary = list.stream()
                .collect(Collectors.minBy(Comparator.comparing(Employee::getSalary)));
        System.out.println("Minimum Salary Emp:: " + minEmpSalary.get());

        Optional<Employee> maxEmpSalary = list.stream()
                .collect(Collectors.maxBy(Comparator.comparing(Employee::getSalary)));
        System.out.println("Maximum Salary Emp:: " + maxEmpSalary.get());

    }
}
```

Group By Operation


User.java

```java
package in.ashokit;

public class User {

    String name;
    int salary;
    String country;

    public User(String name, int salary, String country) {
        this.name = name;
        this.salary = salary;
        this.country = country;
    }

    //setters & getters methods
    // toString ( ) method

}
```


Demo.java

```java
package in.ashokit;

import java.util.ArrayList;

public class Demo {
    public static void main(String[] args) {
        List<User> users = new ArrayList<>();

        users.add(new User("Ram", 10000, "India"));
        users.add(new User("Anil", 20000, "Canada"));
        users.add(new User("Sunil", 30000, "India"));
        users.add(new User("Orlen", 40000, "Japan"));
        users.add(new User("Cathie", 50000, "UK"));
        users.add(new User("Ching Chong", 10000, "China"));

        Map<String, List<User>> collect = users.stream()
                .collect(Collectors.groupingBy(User::getCountry));

        System.out.println(collect);

    }
}
```

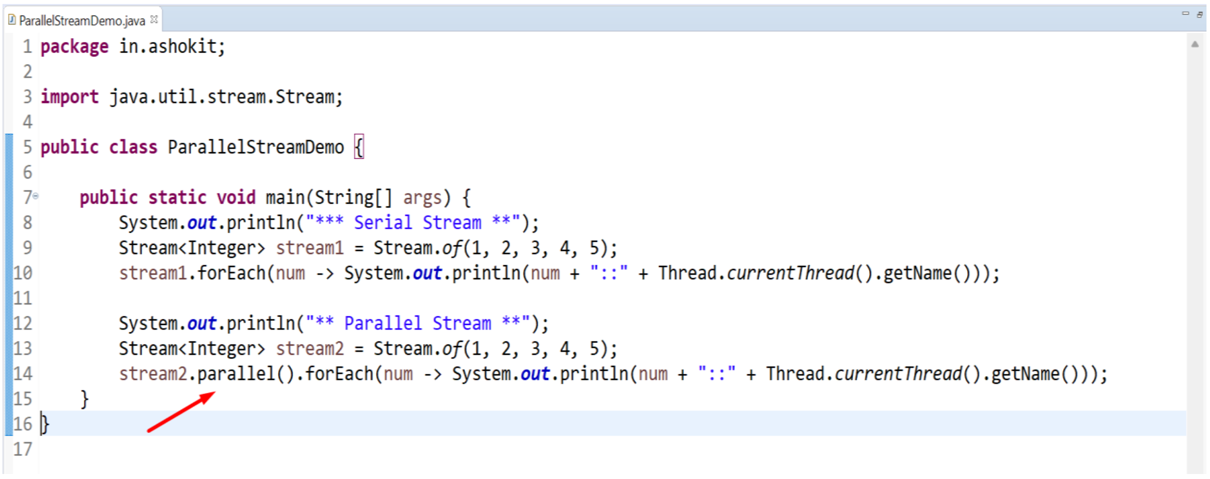
Parallel Streams

-> Streams is one of the major change added in java 1.8 version

-> Generally Streams will execute in Sequential manner

-> We can use parallel streams also to execute program faster by utilizing system resources efficiently.

-> Parallel Streams introduced to improve performance of the program.



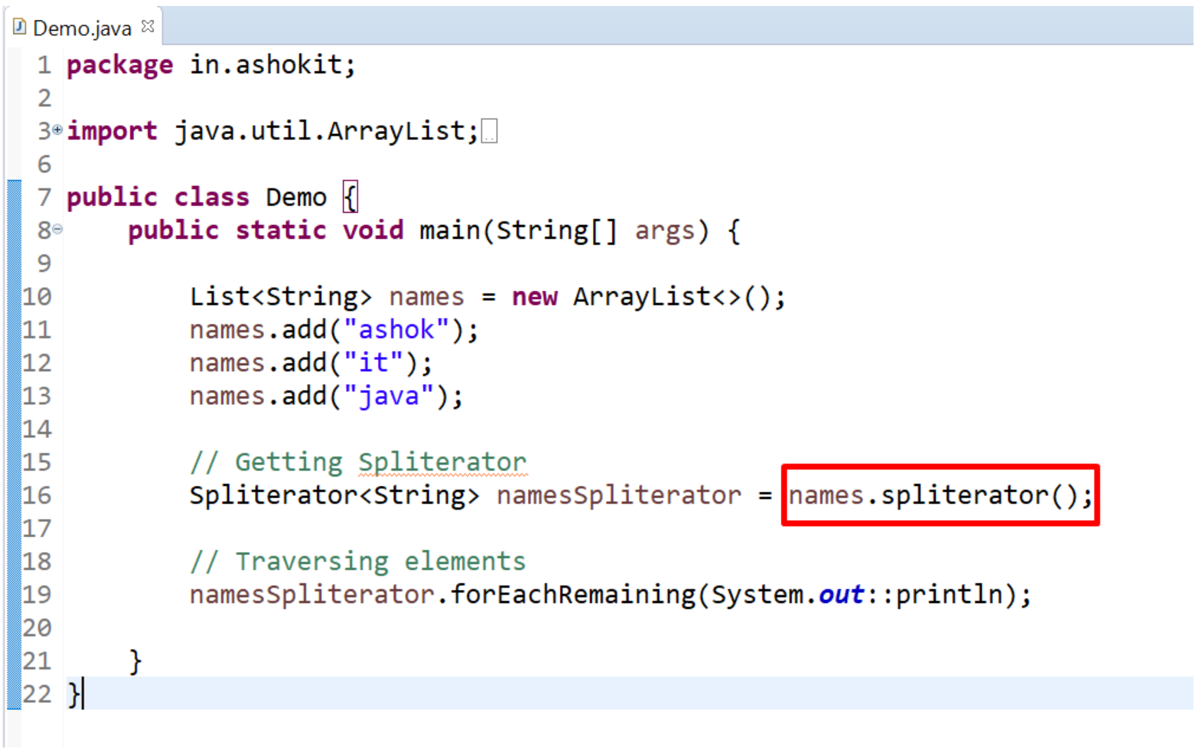
ParallelStreamDemo.java

```java
package in.ashokit;

import java.util.stream.Stream;

public class ParallelStreamDemo {

    public static void main(String[] args) {
        System.out.println("*** Serial Stream **");
        Stream<Integer> stream1 = Stream.of(1, 2, 3, 4, 5);
        stream1.forEach(num -> System.out.println(num + "::" + Thread.currentThread().getName()));

        System.out.println("** Parallel Stream **");
        Stream<Integer> stream2 = Stream.of(1, 2, 3, 4, 5);
        stream2.parallel().forEach(num -> System.out.println(num + "::" + Thread.currentThread().getName()));
    }
}

```

Java Spliterator

Like Iterator and ListIterator, Spliterator is a Java Iterator, which is used to iterate elements one-by-one from a List implemented object. Some important points about Java Spliterator are:

- Java Spliterator is an interface in Java Collection API.
- Spliterator is introduced in Java 8 release in java.util package.
- It supports Parallel Programming functionality.
- We can use it for both Collection API and Stream API classes.
- It provides characteristics about Collection or API objects.
- We can NOT use this Iterator for Map implemented classes.
- It uses tryAdvance() method to iterate elements individually in multiple Threads to support Parallel Processing.
- It uses forEachRemaining() method to iterate elements sequentially in a single Thread.
- It uses trySplit() method to divide itself into Sub-Spliterators to support Parallel Processing.
- Spliterator supports both Sequential and Parallel processing of data.


Demo.java

```
package in.ashokit;

import java.util.ArrayList;

public class Demo {
    public static void main(String[] args) {

        List<String> names = new ArrayList<>();
        names.add("ashok");
        names.add("it");
        names.add("java");

        // Getting Spliterator
        Spliterator<String> namesSpliterator = names.spliterator();

        // Traversing elements
        namesSpliterator.forEachRemaining(System.out::println);

    }
}
```

<u>Date API changes in java 1.8 version</u>

-> In java we have java.util.Date class to work with date related functionality in our application.

Date date = new Date();

-> When we create object for Date class it will give both date and time (current date & time).

-> If we want to get only date without time or only time without date then we have to write our own logic using Date class object.

-> To overcome this problem, in java 1.8 v changes happened for Date API.

-> In Java 1.8v new classes provided to work with date related functionality

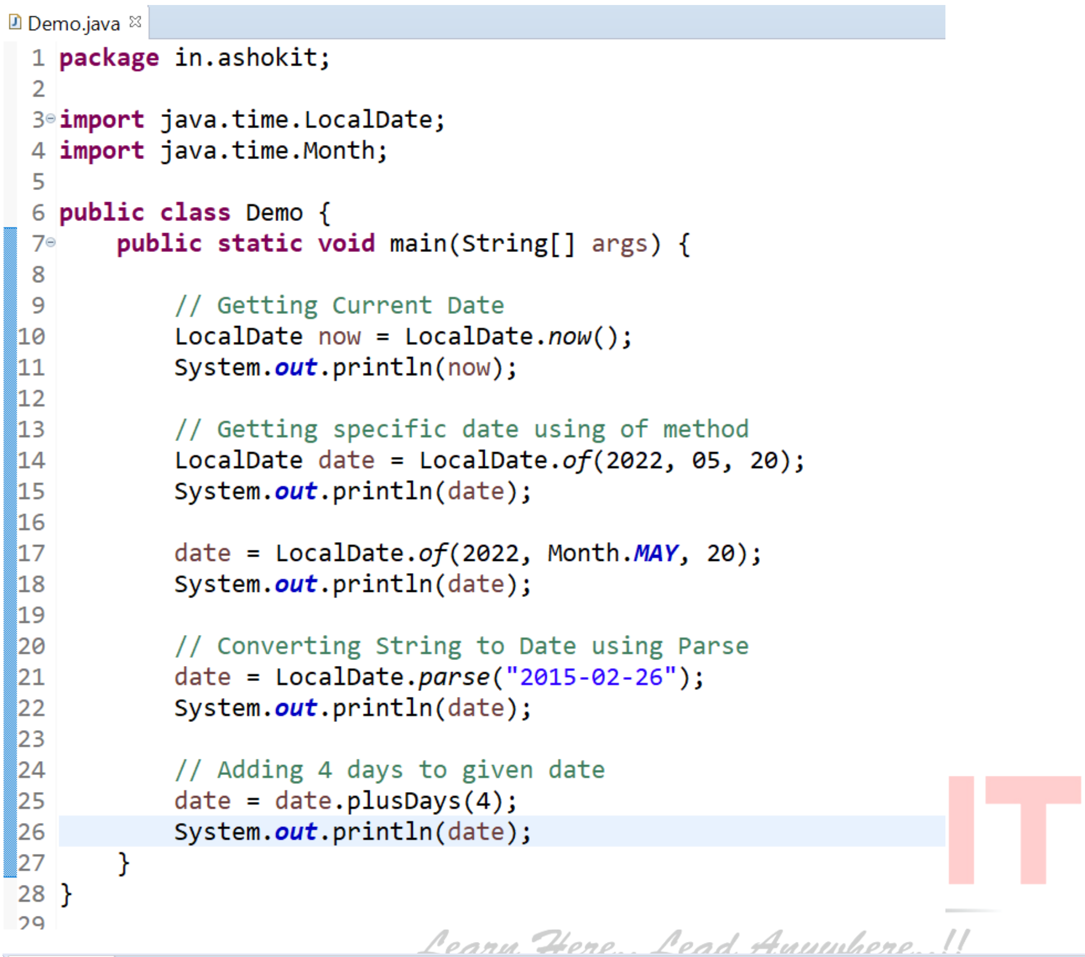
java.time.LocalDate (It will deal with only date)

java.time.LocalTime (It will deal with only time)

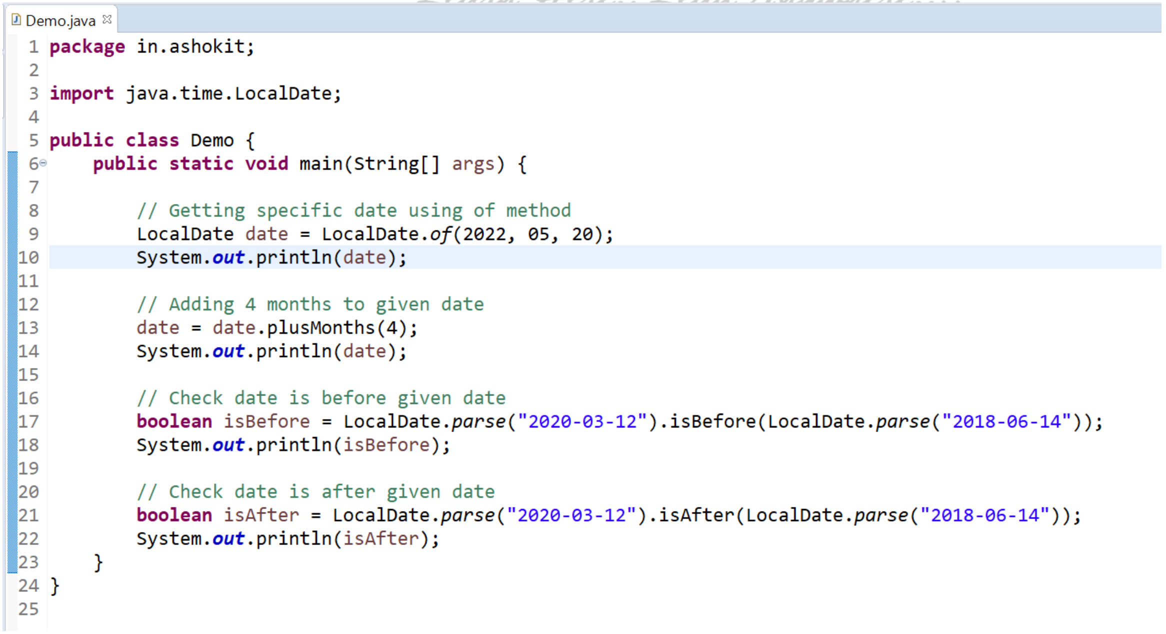
java.time.LocalDateTime (It will deal with both date & time)

-> The above classes got introduced in java.time package.

// java program LocalDate class

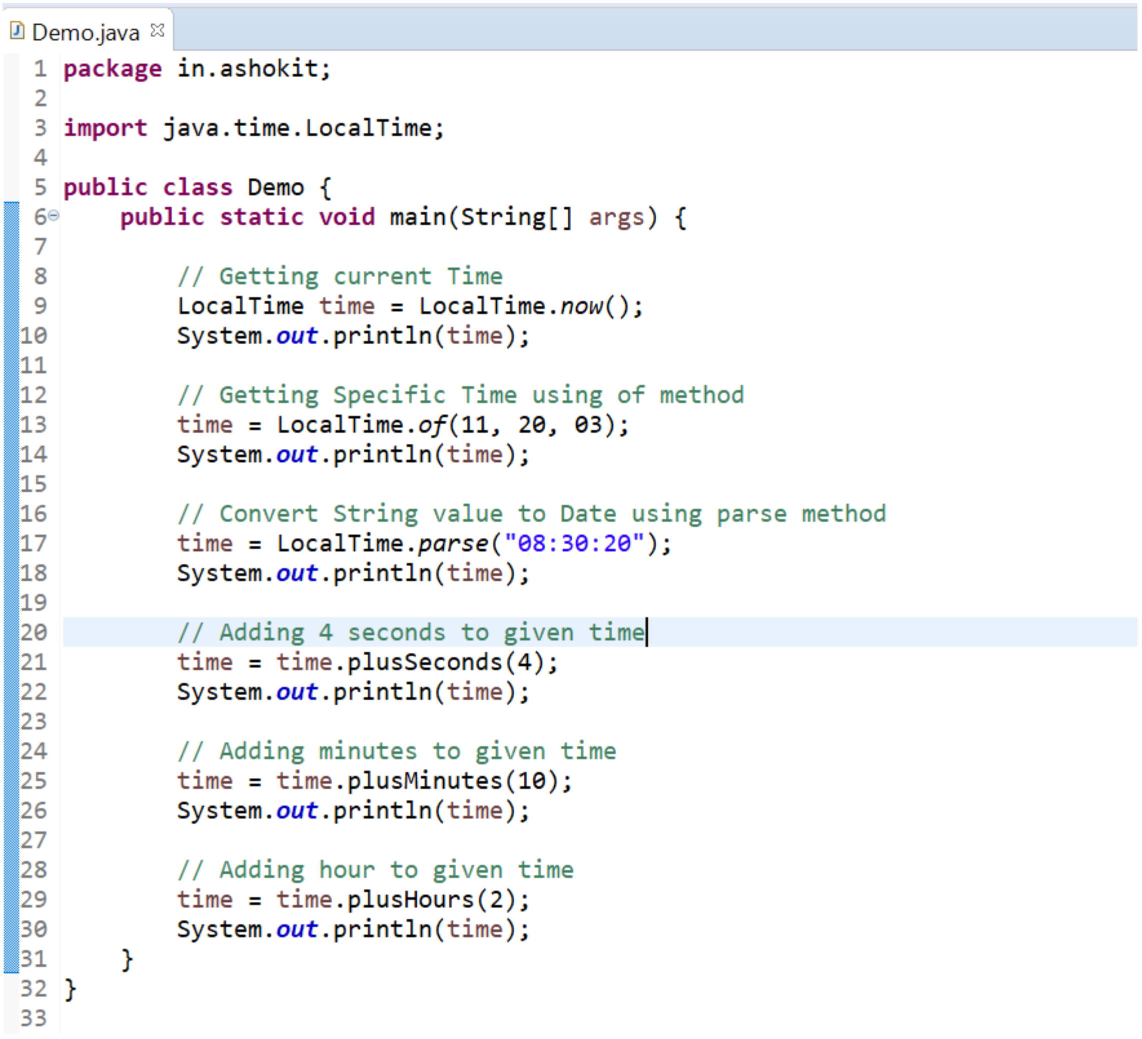

Demo.java

```java
package in.ashokit;

import java.time.LocalDate;
import java.time.Month;

public class Demo {
    public static void main(String[] args) {

        // Getting Current Date
        LocalDate now = LocalDate.now();
        System.out.println(now);

        // Getting specific date using of method
        LocalDate date = LocalDate.of(2022, 05, 20);
        System.out.println(date);

        date = LocalDate.of(2022, Month.MAY, 20);
        System.out.println(date);

        // Converting String to Date using Parse
        date = LocalDate.parse("2015-02-26");
        System.out.println(date);

        // Adding 4 days to given date
        date = date.plusDays(4);
        System.out.println(date);
    }
}
```


Demo.java

```java
package in.ashokit;

import java.time.LocalDate;

public class Demo {
    public static void main(String[] args) {

        // Getting specific date using of method
        LocalDate date = LocalDate.of(2022, 05, 20);
        System.out.println(date);

        // Adding 4 months to given date
        date = date.plusMonths(4);
        System.out.println(date);

        // Check date is before given date
        boolean isBefore = LocalDate.parse("2020-03-12").isBefore(LocalDate.parse("2018-06-14"));
        System.out.println(isBefore);

        // Check date is after given date
        boolean isAfter = LocalDate.parse("2020-03-12").isAfter(LocalDate.parse("2018-06-14"));
        System.out.println(isAfter);
    }
}
```

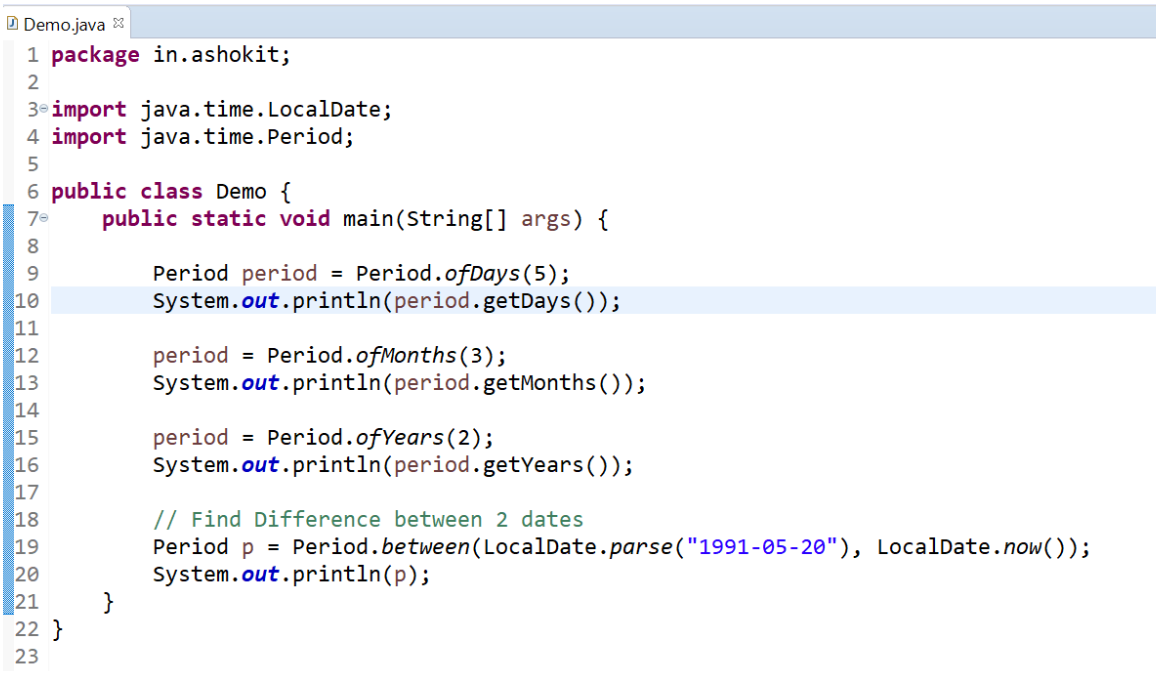
// java program on LocalTime class


Demo.java

```java
package in.ashokit;

import java.time.LocalTime;

public class Demo {
    public static void main(String[] args) {

        // Getting current Time
        LocalTime time = LocalTime.now();
        System.out.println(time);

        // Getting Specific Time using of method
        time = LocalTime.of(11, 20, 03);
        System.out.println(time);

        // Convert String value to Date using parse method
        time = LocalTime.parse("08:30:20");
        System.out.println(time);

        // Adding 4 seconds to given time
        time = time.plusSeconds(4);
        System.out.println(time);

        // Adding minutes to given time
        time = time.plusMinutes(10);
        System.out.println(time);

        // Adding hour to given time
        time = time.plusHours(2);
        System.out.println(time);
    }
}

```

-> Period class is used to check difference between 2 dates

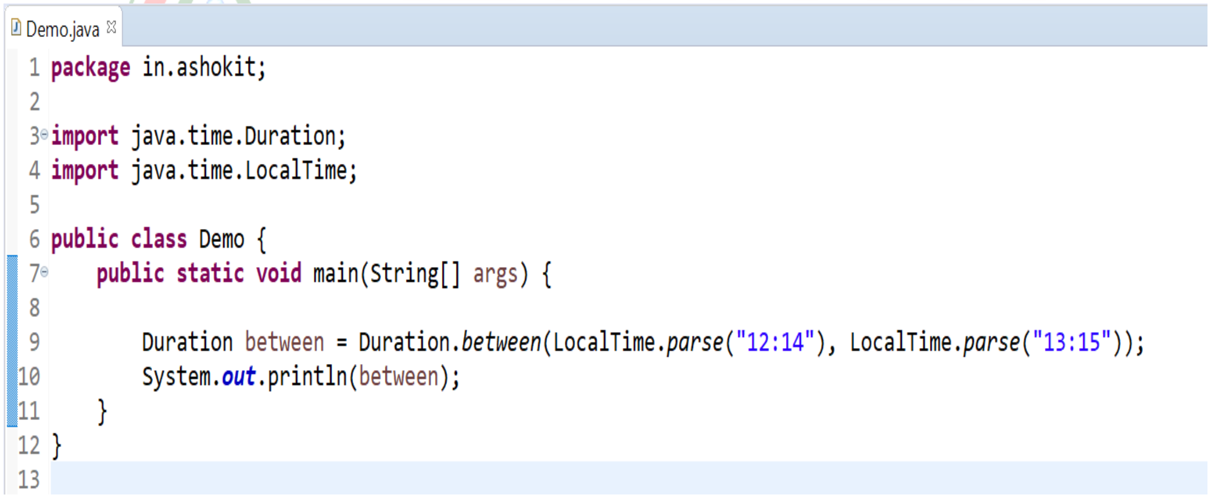
// java program on Period class


Demo.java

```java
package in.ashokit;

import java.time.LocalDate;
import java.time.Period;

public class Demo {
    public static void main(String[] args) {

        Period period = Period.ofDays(5);
        System.out.println(period.getDays());

        period = Period.ofMonths(3);
        System.out.println(period.getMonths());

        period = Period.ofYears(2);
        System.out.println(period.getYears());

        // Find Difference between 2 dates
        Period p = Period.between(LocalDate.parse("1991-05-20"), LocalDate.now());
        System.out.println(p);
    }
}
```

-> Duration class is used to check difference between 2 times.

// java program on Duration class


Demo.java

```java
package in.ashokit;

import java.time.Duration;
import java.time.LocalTime;

public class Demo {
    public static void main(String[] args) {

        Duration between = Duration.between(LocalTime.parse("12:14"), LocalTime.parse("13:15"));
        System.out.println(between);
    }
}
```

Java – 8 Knowledge – Check

1) What are the new features added in Java 8?
2) What is Default method and why we need it?
3) What is static method and why we need it?
4) Default methods vs Static Methods in interfaces?
5) What is Functional Interface?
6) What is Lambda expression?
7) How to write Lambda expression?
8) How to invoke Lambda expressions?
9) What are predefined Functional Interfaces available in java 8?
10) What is Predicate?
11) What is Supplier?
12) What is Consumer?
13) What is Function?
14) What is Optional class and why we need it?
15) What is Stream API?
16) How to create Stream object in java?
17) How to filter data using Stream API?
18) What is map ( ) operation in java 8?
19) What is flatMap ( ) in java 8 ?
20) What is the difference between intermediate and terminal operations in Stream?
21) Write java a program to get names of the employees whose salary is greater than 1 lakh.
22) What are date changes implemented in Java 8?
23) What is forEach ( ) method in java ?
24) What are the new changes introduced in java 8 w.r.t Collection Framework?
25) What is SplIterator?

=== Ashok IT : Lean Here .. Lead Anywhere...!! ===