
Java™

INDEX

Java 8 Features ----------

Java 8 Features

Introduction

- After Java 1.5 version, Java 8 is the next major version.
- Before Java 8, Sun Microsystem people gave importance only for objects but in 1.8 version oracle people gave the importance for functional aspects of programming to bring its benefits to Java i.e. it doesn't mean Java is functional oriented programming language.
- Java 8 New Features:
    a. Lambda (λ) Expression
    b. Functional Interfaces
    c. Default methods in interfaces
    d. Static methods in interfaces
    e. Predicate
    f. Function
    g. Consumer
    h. Supplier

    } Predefine functional interfaces

    i. Double colon operator (: :) – Method reference & Constructor reference
    j. Stream API
    k. Date and Time API (Joda API)
    Etc.

Lambda (λ) Expression

- Lambda calculus is a big change in mathematical world which has been introduced in 1930. Because of benefits of Lambda calculus slowly this concept started using in programming world. "LISP" is the first programming which uses Lambda Expression.
- The other languages which use lambda expressions are:
    - C#.Net
    - Objective C
    - C
    - C++
    - Scala
    - Python
    - Ruby etc.
    - and finally in Java also.

- The main objective of Lambda expression is
    - To bring benefits of functional programming into Java.
    - To write more readable, maintainable and concise code.

- To use APIs very easily & effectively.
- To enable parallel processing.

What is Lambda Expression (λ):

- Lambda expression is just an anonymous (nameless) function. That means the function which doesn't have the name, any return type and any access modifiers.
- Lambda expression also known as anonymous functions or closures.

Ex: 1
```
public void m1() {
  sop("hello");
}

() -> {
  sop("hello");
}
()-> {sop("hello");};
()-> sop("hello");
```

Ex: 2
```
public void add (int a, int b) {
  sop (a + b);
}

(int a, int b) -> {sop (a + b);};
```

- If the type of the parameter can be decided by compiler automatically based on the context, then we can remove types also.
- The above Lambda expression we can rewrite as
  (a, b) -> sop (a + b);

Ex: 3
```
public int getLength (String str) {
  return str.length();
}

(String str) -> {return str.length();};
(str) -> str.length();
str -> str.length();
```

Conclusions:

- A lambda expression can have zero or more number of parameters (arguments).

  Ex:
  () -> sop(“hello”);
  (int a) -> sop(a);
  (int a, int b) -> return a + b;

- Usually, we can specify type of parameter. If the compiler expects the type based on the context, then we can remove type [type inference]. i.e., programmer is not required.

  Ex:
  (int a, int b) -> sop (a + b);
  (a, b) -> sop (a + b);

- If multiple parameters present then these parameters should be separated with comma (,).

- If only one parameter is available and if the compiler can expect the type, then we can remove the type and parenthesis also.

  Ex:
  (int a) -> sop(a);
  (a) -> sop(a); a -> sop(a);

- Similar to method body lambda expression body also can contain multiple statements. If more than one statements present then we have to enclose inside within curly braces. If one statement present then curly braces are optional.

- If lambda expression only returns something then we can remove the return keyword also.

  Ex:
  str -> str.length();

- Once we write lambda expression, we can call/ invoke that expression just like a method, for this functional interface are required.

Functional Interfaces

- If an interface contains only one abstract method, such type of interfaces is called as functional interfaces and the method is called as functional method or single abstract method (SAM).

  Ex:
  a. Runnable - It contains only run () method
  b. Comparable - It contains only compareTo () method
  c. ActionListener - It contains only actionPerformed()
  d. Callable - It contains only call () method

- Inside functional interface in addition to single abstract method (SAM) we write any number of default and static methods.

  Ex:
```
interface Interf {
        public abstract void m1();
        default void m2() {
                System.out.println ("hello");
        }
}
```

- In Java 8, Sun Microsystem introduced @FunctionalInterface annotation to specify that the interface is a Functional Interface.

  Ex:
```
@FunctionalInterface ✓
interface Interf {.
        public void m1();
        default void m2() {
        }
        Public static void m3() {
        }
}
```

- Inside Functional Interface we can take only one abstract method, if we take more than one abstract method then compiler raise an error message that is
  CE: Unexpected @FunctionalInterface annotation,
  Multiple non-overriding abstract methods present is interface Interf

Ex:

```
@FunctionalInterface {❌
interface Interf {
        public void m1();
        public void m2();
}
```

- Inside Functional Interface we have to take exactly only one abstract method. If we are not declaring that abstract method then compiler gives an error message

  CE: Unexpected @FunctionalInterface annotation,
  No abstract method found in interface Interf

  Ex:

```
@Functional Interface {
interface Interf {❌
}
```

Functional Interface with respect to Inheritance

Case 1: If an interface extends Functional Interface and child interface doesn't contain any abstract method then child interface is also Functional Interface.

Ex: ✓

```
@FunctionalInterface
interface A {
        public void m1();
}
@FunctionalInterface
interface B extends A {
}
```

Case 2: In the child interface we can define exactly same parent interface abstract method.

Ex: ✓

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

```
interface C extends P {
        public void m1();
}
```

Case 3: In the child interface we can't define any new abstract methods otherwise we will get compile time error and by mistake if we are trying to declare child interface as @FunctionalInterface annotation then immediately compiler will give an error message

CE: Unexpected @FunctionalInterface annotation,
No abstract method found in interface C

Ex 1: ❌

```
@FunctionalInterface
interface P {
        public void m1();
}
@FunctionalInterface
interface C extends P {
        public void m2();
}
```

Ex 2: ✓

```
@Functional Interface
interface P {
        public void m1();
}
interface C extends P {
        public void m2();
}
```

- No compile time error and C is Normal interface so that code compiles without any error.
- In the all above examples in both parent & child interface we can write any number of default and static methods and there are no restrictions. Restrictions are applicable only for abstract methods only.

Functional Interface vs Lambda Expressions

- Once we write Lambda expressions to invoke its functionality, then Functional Interface is required. We can use Functional Interface reference to refer Lambda Expression.

- Where ever Functional Interface concept is applicable there we can use Lambda Expressions.

Ex 1:

Without Lambda (λ) Expression

```
interface Interf {
        public void m1();
}
public class Demo implements Interf {
        public void m1() {
                System.out.println("m1() method implementation");
        }
}
public class Test {
        public static void main(String[] args) {
                Interf i = new Demo();
                i.m1();
        }
}
```

With Lambda (λ) Expression

```
interface Interf {
        public void m1();
}
public class Test {
        public static void main(String[] args) {
                Interf i = () -> System.out.println("m1() method
                                                        implementation");
                i.m1();
        }
}
```

Ex 2:

Without Lambda (λ) Expression

```
interface Interf {
        public void add(int a, int b);
}
public class Demo implements Interf {
        public void add(int a, int b) {
                System.out.println("The sum "+(a+b));
```

```
        }
}
public class Test {
        public static void main(String[] args) {
                Interf i = new Demo();
                i.add(10, 20);
                i.add(100, 200);
        }
}
```

With Lambda (λ) Expression

```
interface Interf {
        public void add(int a, int b);
}
public class Test {
        public static void main(String[] args) {
                Interf i = (int a, int b) ->
                        System.out.println("The sum "+(a+b));
                i.add(10, 20);
                i.add(100, 200);
        }
}
```

Ex 3:

Without Lambda (λ) Expression

```
interface Interf {
        public int getLength(String s);
}
public class Demo implements Interf {
        public int getLength(String s) {
                return s.length();
        }
}
public class Test {
        public static void main(String[] args) {
                Interf i = new Demo();
                System.out.println(i.getLength("Hello"));
                System.out.println(i.getLength("without Lambda"));
        }
}
```

With Lambda (λ) Expression

```
interface Interf {
        public int getLength(String s);
}
public class Test {
        public static void main(String[] args) {
                Interf i = (String s) -> {return s.length();};
                Interf i = s -> return s.length();
                System.out.println(i.getLength("Hello"));
                System.out.println(i.getLength("with Lambda"));
        }
}
```

Ex 4:

Without Lambda (λ) Expression

```
interface Interf {
        public int squareIt(int x);
}
public class Demo implements Interf {
        public int squareIt(int x) {
                return x * x;
        }
}
public class Test {
        public static void main(String[] args) {
                Interf i = new Demo();
                System.out.println(i.squareIt (4));
                System.out.println(i.squareIt(5));
        }
}
```

With Lambda (λ) Expression

```
interface Interf {
        public int squareIt(int x);
}
public class Test {
        public static void main(String[] args) {
                Interf i = x -> return x*x;
                System.out.println(i.squareIt(4));
                System.out.println(i.squareIt(4));
```

```
        }
}
```

Ex 5:

Without Lambda (λ) Expression

```
public class MyRunnable implements Runnable {
        public void run() {
                for(int i=0; i<10; i++) {
                        System.out.prinlnt("Child Thread");
                }
        }
}
public class ThreadDemo {
        public static void main(String[] args) {
                Runnable r = new MyRunnable();
                Thread th = new Thread(r);
                t.start();

                for(int i=0; i<10; i++) {
                        System.out.prinlnt("Main Thread");
                }
        }
}
```

With Lambda (λ) Expression

```
public class ThreadDemo {
        public static void main(String[] args) {
                Runnable r = () -> {
                        for(int i=0; i<10; i++) {
                                System.out.prinlnt("Child Thread");
                        }
                };
                Thread th = new Thread(r);
                t.start();

                for(int i=0; i<10; i++) {
                        System.out.prinlnt("Main Thread");
                }
        }
}
```

Functional interface & Lambda expression summary:

- It should contain exactly one abstract method (SAM – Single abstract method).
- It can contain any number of default and static methods.
- It acts as a type for Lambda (λ) expression.
  e.g.: Interf i = ()->System.out.println("Hello");
- It can be used to invoke Lambda (λ) expression.
  e.g.: i.m1();

Q. Why functional interface should contain only one abstract method?
Ans. Functional interface act as type for Lambda expression and used to invoke lambda expression. Lambda expression should map to some method of interface, so if interface contains multiple abstract method, then in the mapping there is a problem going come. That's why these functional interfaces should compulsory contain a single abstract method only.

Note:

- ✓ Without abstract method or more than one abstract method its is always not treated as functional interface and we can't user Lambda expression.
- ✓ For this reason, only Java people given @FunctionalInterface annotation to make interface as functional interface.

Ex:

```
interface Interf {
        public void m1(int x);
}
Interf i = x -> System.out.println(i*i); ✓

interface Interf {
        public void m1(int x);
        public void m2(int x);
}
Interf i = x -> System.out.println(i*i); ❌
```

CE: Incompatible types: Interf is not a functional interface.
Multiple non-overriding abstract methods in interface Interf

Q. What is the advantage of @FunctionalInterface annotation?
Ans. @FunctionalInterface is used specify explicitly, a functional interface is used for lambda expression don't add any new abstract method.

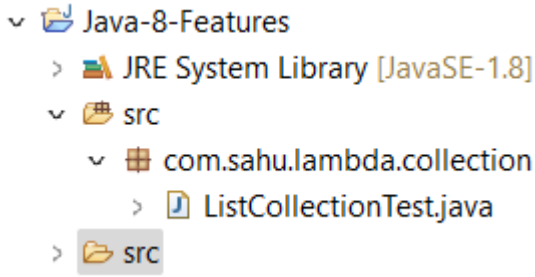
Lambda Expressions with Collections

- Collection is nothing but a group of objects represented as a single entity. The important Collection types are:
  - List (I)
  - Set (I)
  - Map (I)

List (I):

- If we want to represent a group of objects as a single entity where duplicate objects are allowed and insertion order is preserved then we should go for List.
- Insertion order is preserved.
- Duplicate objects are allowed.
- The main implementation classes of List interface are,
  - ArrayList
  - LinkedList
  - Vector
  - Stack

Directory Structure of Java-8-Features:



- Develop the above directory structure using Java Project option and create the packages and java files.
- Then place the following code with in their respective files.

ListCollectionTest.java

```
package com.sahu.lambda.collection;

import java.util.ArrayList;
import java.util.List;

public class ListCollectionTest {
```

```
        public static void main(String[] args) {
                List<String> nameList = new ArrayList<>();
                nameList.add("Sunny");
                nameList.add("Bunny");
                nameList.add("Chinny");
                nameList.add("Sunny");
                System.out.println(nameList);
        }
}
```

Output: [Sunny, Bunny, Chinny, Sunny]

Note: List (may be ArrayList, LinkedList, Vector or Stack) never talks about sorting order. If we want sorting for the list then we should use Collections class sort () method.

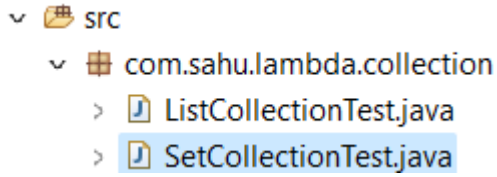
- Collections.sort(list) - meant for Default natural sorting order.
- Collections.sort(list, Comparator) - meant for customized sorting order.

Set (I):

- If we want to represent a group of individual objects as a single entity where duplicate objects are not allowed and insertion order is not preserved then we should go for Set.
- Insertion order is not preserved.
- Duplicates are not allowed. If we are trying to add duplicates then we won't get any error, just add() method returns false.
- The following are important Set implementation classes
    - HashSet
    - TreeSet

      Etc.

Note: In the case of Set, if we want sorting order then we should go for: TreeSet.



- Add the selected java files, then places the following code with in their respective file.

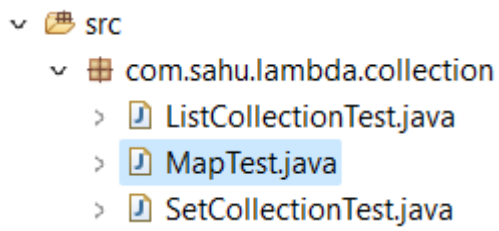
SetCollectionTest.java

```
package com.sahu.lambda.collection;

import java.util.HashSet;
import java.util.Set;

public class SetCollectionTest {

	public static void main(String[] args) {
		Set<String> nameList = new HashSet<>();
		nameList.add("Sunny");
		nameList.add("Bunny");
		nameList.add("Chinny");
		nameList.add("Sunny");
		System.out.println(nameList);
	}

}
```

Output: [Chinny, Bunny, Sunny]

Map (I):

- If we want to represent objects as key-value pairs then we should go for Map.
  E.g.:
  - Roll no - Name
  - Mobile number - address
- The important implementation classes of Map are
  - HashMap
  - TreeMap
    Etc.



- Add the selected java files, then places the following code with in their respective file.

MapTest.java

```
package com.sahu.lambda.collection;

import java.util.HashMap;
import java.util.Map;

public class MapTest {

	public static void main(String[] args) {
		Map<String, String> alphaMap = new HashMap<>();
		alphaMap.put("A","Apple");
		alphaMap.put("Z","Zebra");
		alphaMap.put("Durga","Java");
		alphaMap.put("B","Boy");
		alphaMap.put("T","Tiger");
		System.out.println(alphaMap);
	}

}
```

Output: {A=Apple, B=Boy, T=Tiger, Z=Zebra, Durga=Java}

Comparator (I):

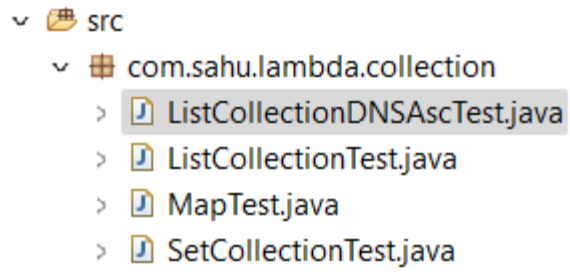
- Instead of Default natural sorting order if we want customized sorting order then we should go for Comparator interface.
- Where ever Comparator (I) is there we should go for lambda expression, because this interface contains only one abstract method i.e. compare () method.
- This metho is used to define our own sorting (custom sorting).
- Method signature

  public int compare (Object obj1, Object obj2);
  - Returns -ve, if obj1 has to come before obj2
  - Return +ve, if obj1 has to come after obj2
  - Return 0, if obj1 and obj2 are equal

Sorted List:

- List (may be ArrayList, LinkedList, Vector or Stack) never talks about sorting order. If we want sorting for the list then we should use Collections class sort () method.

- Collecttions.sort(list) - meant for Default Natural Sorting Order
    - For String objects: Alphabetical Order
    - For Numbers: Ascending order



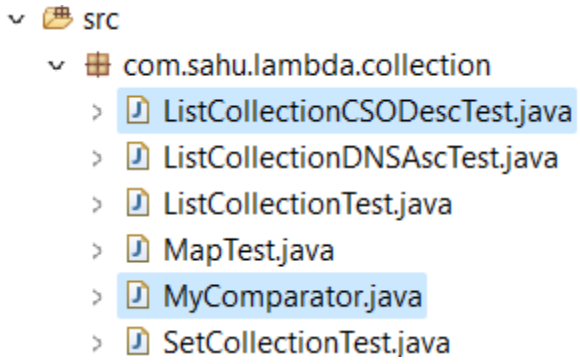
- Add the selected java files, then places the following code with in their respective file.

ListCollectionDNSAscTest.java

```
package com.sahu.lambda.collection;

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

public class ListCollectionDNSAscTest {

	public static void main(String[] args) {
		List<Integer> numList = new ArrayList<>();
		numList.add(10);
		numList.add(0);
		numList.add(15);
		numList.add(5);
		numList.add(20);
		System.out.println("Before Sorting: "+numList);
		Collections.sort(numList);
		System.out.println("After Sorting: "+numList);
	}

}
```

Output:
Before Sorting: [10, 0, 15, 5, 20]
After Sorting: [0, 5, 10, 15, 20]

If we want our own customised sorting order then we should go for comparator concept.



- Add the selected java files, then places the following code with in their respective file.

MyComparator.java

```
package com.sahu.lambda.collection;

import java.util.Comparator;

public class MyComparator implements Comparator<Integer> {

	@Override
	public int compare(Integer i1, Integer i2) {
		if (i1 > i2) {
			return -1;
		} else if (i1 < i2) {
			return 1;
		} else {
			return 0;
		}
	}

}
```

ListCollectionCSODescTest.java

```
package com.sahu.lambda.collection;

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

```java
import java.util.List;

public class ListCollectionCSODescTest {

	public static void main(String[] args) {
		List<Integer> numList = new ArrayList<>();
		numList.add(10);
		numList.add(0);
		numList.add(15);
		numList.add(5);
		numList.add(20);
		System.out.println("Before Sorting: "+numList);
		Collections.sort(numList, new MyComparator());
		System.out.println("After Sorting: "+numList);
	}

}
```

Output:
Before Sorting: [10, 0, 15, 5, 20]
After Sorting: [20, 15, 10, 5, 0]

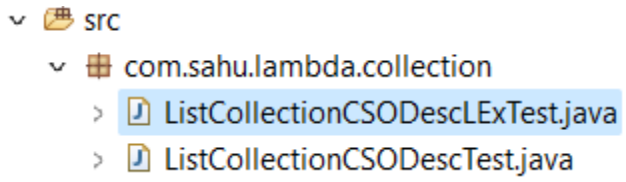
The above Comparator class we can write as below

MyComparator.java

```java
package com.sahu.lambda.collection;

import java.util.Comparator;

public class MyComparator implements Comparator<Integer> {

	@Override
	public int compare(Integer i1, Integer i2) {
		return (i1 > i2) ? -1 : (i1 < i2) ? 1 : 0;
	}

}
```

Sorting with Lambda expression:

- As Comparator is Functional interface, we can replace its implementation with Lambda expression.
  Collections.sort(nameList, (i1, i2) -> (i1 > i2) ? -1 : (i1 < i2) ? 1 : 0);



- Add the selected java files, then places the following code with in their respective file.

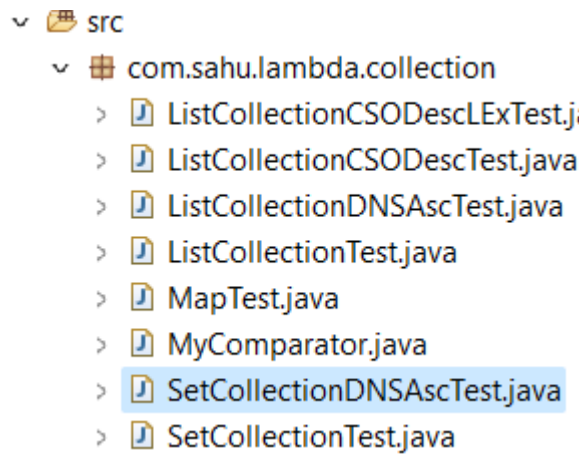
ListCollectionCSODescLExTest.java

```
package com.sahu.lambda.collection;

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

public class ListCollectionCSODescLExTest {

	public static void main(String[] args) {
		List<Integer> numList = new ArrayList<>();
		numList.add(10);
		numList.add(0);
		numList.add(15);
		numList.add(5);
		numList.add(20);
		System.out.println("Before Sorting: "+numList);
		Collections.sort(numList, (i1, i2) -> (i1 > i2) ? -1 : (i1 < i2) ? 1 : 0);
		System.out.println("After Sorting: "+numList);
	}

}
```

Output:
Before Sorting: [10, 0, 15, 5, 20]
After Sorting: [20, 15, 10, 5, 0]

Sorted Set:

- In the case of Set, if we want sorting order then we should go for TreeSet.
- TreeSet t = new TreeSet (); -This TreeSet object meant for default natural sorting order.
- TreeSet t = new TreeSet (Comparator c); This TreeSet object meant for customized sorting order.



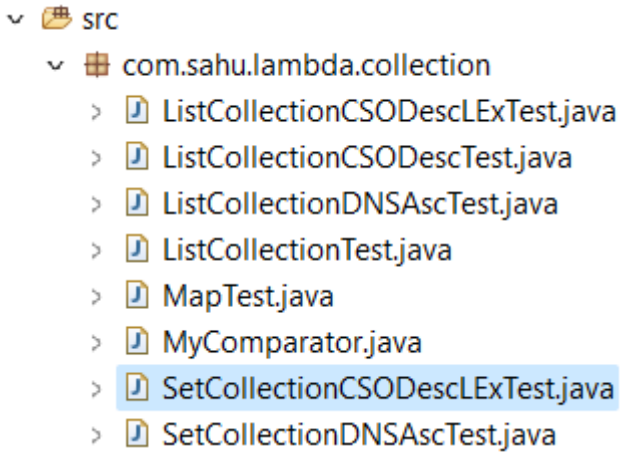
- Add the selected java files, then places the following code with in their respective file.

SetCollectionDNSAscTest.java

```
package com.sahu.lambda.collection;

import java.util.Set;
import java.util.TreeSet;

public class SetCollectionSDNSAscTest {
	public static void main(String[] args) {
		Set<Integer> numList = new TreeSet<>();
		numList.add(10);
		numList.add(0);
		numList.add(15);
		numList.add(25);
		numList.add(5);
		numList.add(20);
		System.out.println(numList);
	}
}
```

Output: [0, 5, 10, 15, 20, 25]

Custom sorting using lambda expression.



- Add the selected java files, then places the following code with in their respective file.

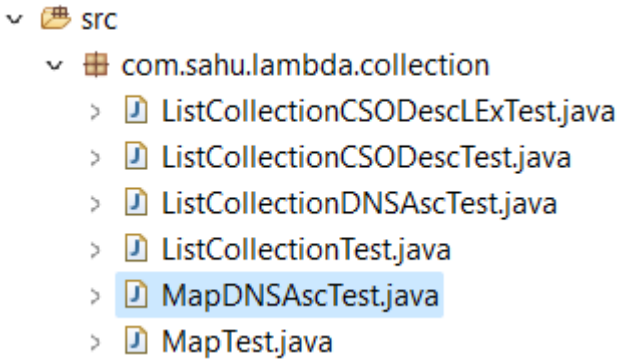
SetCollectionCSODescLExTest.java

```
package com.sahu.lambda.collection;

import java.util.Set;
import java.util.TreeSet;

public class SetCollectionCSODescLExTest {

	public static void main(String[] args) {
		Set<Integer> numList = new TreeSet<>((i1, i2) -> (i1 > i2) ? -1 :
(i1 < i2) ? 1 : 0);
		numList.add(10);
		numList.add(0);
		numList.add(15);
		numList.add(25);
		numList.add(5);
		numList.add(20);
		System.out.println(numList);
	}

}
```

Output: [25, 20, 15, 10, 5, 0]

Sorted Map:

- In the case of Map, if we want default natural sorting order based on keys then we should go for TreeMap.
- TreeMap map = new TreeMap(); - This TreeMap object meant for default natural sorting order of keys.
- TreeMap t = new TreeMap(Comparator c); - This TreeMap object meant for Customized sorting order of keys.



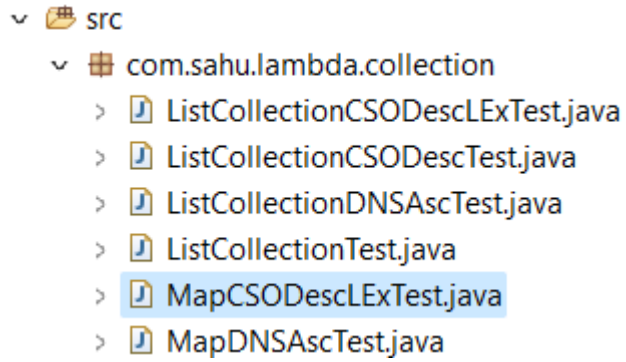
- Add the selected java files, then places the following code with in their respective file.

MapDNSAscTest.java

```
package com.sahu.lambda.collection;

import java.util.Map;
import java.util.TreeMap;

public class MapDNSAscTest {

	public static void main(String[] args) {
		Map<Integer, String> nameMap = new TreeMap<>();
		nameMap.put(100, "Durga");
		nameMap.put(600, "Sunny");
		nameMap.put(300, "Bunny");
		nameMap.put(200, "Chinny");
		nameMap.put(700, "Vinny");
		nameMap.put(400, "Vinny");
		System.out.println(nameMap);
	}

}
```

Output: {100=Durga, 200=Chinny, 300=Bunny, 400=Vinny, 600=Sunny, 700=Vinny}

Customized sorting order using Lambda expression



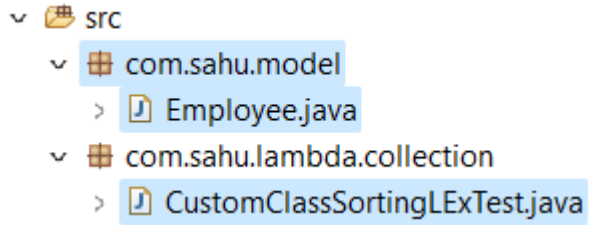
- Add the selected java files, then places the following code with in their respective file.

MapDNSAscTest.java

```
package com.sahu.lambda.collection;

import java.util.Map;
import java.util.TreeMap;

public class MapCSODescLExTest {

	public static void main(String[] args) {
		Map<Integer, String> nameMap = new TreeMap<>((i1, i2) -> (i1 > i2) ? -1 : (i1 < i2) ? 1 : 0);
		nameMap.put(100, "Durga");
		nameMap.put(600, "Sunny");
		nameMap.put(300, "Bunny");
		nameMap.put(200, "Chinny");
		nameMap.put(700, "Vinny");
		nameMap.put(400, "Vinny");
		System.out.println(nameMap);
	}

}
```

Output: {700=Vinny, 600=Sunny, 400=Vinny, 300=Bunny, 200=Chinny, 100=Durga}

Sorting for Customized class objects by using Lambda Expressions



- Create the package and add the selected java files, then places the following code with in their respective file.

Employee.java

```
package com.sahu.model;

public class Employee {
	private Integer empNo;
	private String ename;

	public Integer getEmpNo() {
		return empNo;
	}
	public void setEmpNo(Integer empNo) {
		this.empNo = empNo;
	}
	public String getEname() {
		return ename;
	}
	public void setEname(String ename) {
		this.ename = ename;
	}
	public Employee(Integer empNo, String ename) {
		super();
		this.empNo = empNo;
		this.ename = ename;
	}
	@Override
	public String toString() {
		return empNo + " : " + ename;
	}

}
```

Employee.java

```
package com.sahu.lambda.collection;

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

import com.sahu.model.Employee;

public class CustomClassSortingLExTest {

	public static void main(String[] args) {
		List<Employee> employees = new ArrayList<Employee>();
		employees.add(new Employee(200, "Deepika"));
		employees.add(new Employee(400, "Sunny"));
		employees.add(new Employee(300, "Mallika"));
		employees.add(new Employee(100, "Katrina"));
		System.out.println("Before Sorting : " + employees);
		Collections.sort(employees,
				(e1, e2) -> e1.getEmpNo() < e2.getEmpNo() ? -1 :
e1.getEmpNo() > e2.getEmpNo() ? 1 : 0);
		System.out.println("After Sorting : " + employees);
	}

}
```

Output:
Before Sorting : [200 : Deepika, 400 : Sunny, 300 : Mallika, 100 : Katrina]
After Sorting : [100 : Katrina, 200 : Deepika, 300 : Mallika, 400 : Sunny]

Anonymous inner classes vs Lambda Expressions

- Wherever we are using anonymous inner classes there may be a chance of using Lambda expression, to reduce length of the code and to resolve complexity.
- Nameless inner class is by default consider as inner class.

Ex: With anonymous inner class
public class ThreadDemo {
	public static void main(String[] args) {

```
                Runnable r = new Runnable () {
                        public void run () {
                                for (int i=0; i<10; i++) {
                                        System.out.println("Child Thread");
                                }
                        }
                };
                Thread t = new Thread(r);
                t.start();
                for (int i=0; i<10; i++) {
                        System.out.println("Main thread");
                 }
        }
}
```

Ex: With Lambda expression

```
public class ThreadDemo {
        public static void main(String[] args) {
                Runnable r = () -> {
                        for (int i=0; i<10; i++) {
                                System.out.println("Child Thread");
                        }
                };
                Thread t = new Thread(r);
                        OR
                Thread t = new Thread (() -> {
                        for (int i=0; i<10; i++) {
                                System.out.println("Child Thread");
                        }
                });
                t.start();
                for (int i=0; i<10; i++) {
                        System.out.println("Main thread");
                 }
        }
}
```

Note: We can't replace every anonymous inner class with Lambda expression. Some anonymous inner classes only we can replace with lambda expression because both are not same.

Ex1: How both are not same

```
class Test {

}
Test t = new Test () {

}
```

Anonymous inner class that extends concrete class

```
abstract class Test {

}
Test t = new Test () {
        //abstract method…
}
```

Anonymous inner class that extends abstract class

```
interface Test {
        public void m1;
        public void m2;
        public void m3;
}
Test t = new Test () {
        Public void m1() {}
        Public void m2() {}
        Public void m3() {}
}
```

Anonymous inner class that implements an interface which contains multiple methods

```
interface Test {
        public void m1;
}
Test t = new Test () {
        Public void m1() {}
}
```

Anonymous inner class that implements an interface which contains only one abstract method

- An anonymous inner class which contains only one abstract method that anonymous inner class only replaced with Lambda expression.
- Hence in this particular case only we can replace with lambda expressions.
- Anonymous inner class is more powerful than lambda expression.
- Anonymous inner class! = Lambda Expression.

Ex2:

```
interface Interf {
        public void m1();
}
class Test {
        int x = 888;
        public void m2() {
                Interf i = new Interf() {
                        int x = 999; // instance variables
                        public void m1() {
                                System.out.println(this.x); //999
                        }
                }
                i.m1();
        }
        public static void main (String[] args) {
                Test t = new Test();
                t.m2();
        }
}
```

- Inside anonymous inner class we can declare instance variables.
- Inside anonymous inner class “this” always refers current inner class object (anonymous inner class) but not related outer class object.
- If we wan to access Outer class “X” variable inside the inner class then we have to write <Outer class name>.this.<variable name>
  Ex: Test.this.x;

```
interface Interf {
        public void m1();
}
class Test {
        int x = 888;
        public void m2() {
                Interf i = () -> {
                        int x = 999;
                        System.out.println(this.x); //888
                }
                i.m1();
        }
```

```
        public static void main (String[] args) {
                Test t = new Test();
                t.m2();
        }
}
```

- Inside lambda expression we can't declare instance variables. Whatever the variables declare inside lambda expression are simply acts as local variables.
- Within lambda expression "this" keyword represents outer class object reference (that is current enclosing class reference in which we declare lambda expression).

Differences between anonymous inner classes and Lambda expression

Anonymous Inner class	Lambda Expression
It's a class without name.	It's a function method without name (anonymous function).
Anonymous inner class can extend abstract and concrete classes.	Lambda expression can't extend abstract and concrete classes.
Anonymous inner class can implement an interface that contains any number of abstract methods.	Lambda expression can implement an interface which contains single abstract method (functional interface).
Inside anonymous inner class we can declare instance variables.	Inside lambda expression we can't declare instance variables, whatever variables declared are consider as local variables.
Anonymous inner classes can be Instantiated.	Lambda expressions can't be instantiated.
Inside anonymous inner class "this" always refers current anonymous inner class object but not outer class object.	Inside lambda expression "this" always refers current outer class object. That is enclosing class object.
Anonymous inner class is the best choice if we want to handle multiple methods.	Lambda expression is the best choice if we want to handle interface with single abstract method (Functional Interface).

For the anonymous inner class at the time of compilation, a separate .class file will be generated (outerclass$1.class).	For the lambda expression at the time of compilation, no separate .class file will be generated.
Memory will be allocated on demand whenever we are creating an object.	Lambda expression will reside in permanent memory of JVM (Method Area).

Note:

- ✓ From lambda expression we can access enclosing class variables and enclosing method variables directly.
- ✓ The local variables referenced from lambda expression are implicitly final and hence we can't perform re-assignment for those local variables otherwise we get compile time error.
  CE: Local variables referenced from a lambda expression must be final, or effectively final.

```
interface Interf {
        public void m1();
}
class Test {
        int x = 10;
        public void m2() {
                int y = 20;
                Interf i = () -> {
                        System.out.println(x); //10
                        System.out.println(y); //20
                        x = 888;
                        y = 999; //CE
                }
                i.m1();
                y = 777; //CE
        }
        public static void main (String[] args) {
                Test t = new Test();
                t.m2();
        }
}
```

Q. From lambda expression, is it possible to access class level variable or not?

Ans. Yes, enclosing class level variable happily we can access.

Q. From lambda expression is it possible to access local variables of enclosing method or not?

Ans. Yes, we can access, but local variable which are referenced from lambda expression must be final or effectively final. Hence with in the lambda expression or outside of lambda expression we can't change the value of local variable which are referenced from Lambda expression. If we are not using local variable inside the lambda expression happily, we can change its value because it's not final.

Advantages of Lambda Expression

a. We can enable functional programming in java.
b. We can reduce length of the code so that readability will be improved.
c. We can resolve complexity of anonymous inner class until some extent.
d. We can handle procedures/ functions just like values.
e. We can pass procedures/ function as arguments.
f. Easier to use updated APIs and libraries.
g. Enable support for parallel processing.

Default Methods

- Until 1.7 version onwards inside interface we can take only public abstract methods and public static final variables.
- Every method present inside interface is always public and abstract whether we are declaring or not.
- Every variable declared inside interface is always public static final whether we are declaring or not.
- But from 1.8 version onwards in addition to these, we can declare concrete methods also inside interface. The concreate methods which we can allow to declare inside of interface their methods are by default consider as default method.
- We can declare default method with the keyword "default" as follows
  Ex:

```
Interface Interf {
        default void m1() {
                System.out.println("Default Method");
        }
}
```

- Interface default methods are by-default available to all implementation

classes. Based on requirement, implementation class can use these default methods directly or can override also.

Ex:

```
class Test implements Interf {
        public static void main (String[] args) {
                Test t = new Test ();
                t.m1();
        }
}

class Test implements Interf {
        public void m1() {
                System.out.println("My own implementation");
        }

        public static void main (String[] args) {
                Test t = new Test ();
                t.m1();
        }
}
```

- Default methods also known as defender methods or virtual extension methods.
- The main advantage of default methods is without effecting implementation classes we can add new functionality to the interface (backward compatibility).

Note: We can't override object class methods as default methods inside interface otherwise we get compile time error.

```
interface Interf {
        default int hashCode () {
                return 10;
        }
}
```

CE: A default method cannot override a method from java.lang.Object

Reason: Object class methods are by-default available to every Java class hence it's not required to bring through default methods.

Default method WRT multiple inheritance

Problem: If two interfaces can contain a default method with same signature then there may be a chance of ambiguity problem (diamond problem) to the implementation class.

Ex:

```
interface Left {
        default void m1() {
                System.out.println("Left Default Method");
        }
}
interface Right {
        default void m1() {
                System.out.println("Right Default Method");
        }
}
class Test implements Left, Right {}
```

CE: class Test inherits unrelated defaults for Left and Right.

Solution: To overcome this problem compulsory, we should override default method in the implementation class otherwise we get compile time error.

Q. How to override default method in the implementation class?

Ans. In the implementation class we can provide completely new implementation or we can call any interface method as follows

<Interface name>.super.<method name>

```
class Test implements Left, Right {
        public void m1() {
                System.out.println("Test Class Method");
                        OR
                Left.super.m1();
        }

        public static void main (String[] args) {
                Test t = new Test ();
                t.m1();
        }
}
```

Differences between Interface with Default methods and Abstract class

- Even though we can add concrete methods in the form of default methods to the interface, it never equals to abstract class.
- Interface with default method! = abstract class

Interface with Default methods	Abstract Class
Inside interface every variable is always public static and final, we cannot declare instance variables.	Inside abstract class we can declare instance variables which are required to the child class.
Interface never talks about state of object.	Abstract class can talk about state of object.
Inside interface we can't declare constructors.	Inside abstract class we can declare constructors.
Inside interface we can't declare instance and static blocks.	Inside abstract class we can declare instance and static blocks.
Functional interface with default methods can refer lambda expression.	Abstract class can't refer lambda expressions.
Inside interface we can't override Object class methods.	Inside abstract class we can override Object class methods.

Static Methods inside Interface

- From 1.8 version onwards in addition to default methods we can write static methods also inside interface.
- To define general utility methods for that purpose we can use static method.

  Ex:

```
interface Interf {
        public static void m1() {
        }
}
```

- These methods are not meant only for implementation class, any class can call this method by using interface name.
- Interface static methods by-default not available to the implementation classes, so we can't call the static method by using implementation class reference/ object or implementation class name, if we are trying to access then we will get compile time error.

- We should call interface static methods always we can call by using interface name only.

Ex:

```
interface Interf {
        public static void m1() {
                System.out.println("interface static met hod");
        }
}
class Test implements Interf {
        public static void main(String[] args) {
                Test t = new Test();
                t.m1();  //CE ❌
                Test.m1(); //CE ❌
                Interf.m1(); ✓
        }
}
```

Interface static method WRT Overriding

- Interface static methods by default not available to the implementation class, that's way overriding concept is not applicable.
- Based on our requirement we can define exactly same method in the implementation class, it's valid but not overriding.

Ex 1: Valid but not overriding ✓

```
interface Interf {
        public static void m1() {

        }
}
class Test implements Interf {
        public static void m1() {

        }
}
```

Ex 2: Valid but not overriding ✓

In normal classes, if parent class a method is static and child class method is non-static then immediately, we will get compile time error because we can't override a static method as non-static method but in the case of interface

static method, we never going to get any compile time error because it is not overriding concept.

```
interface Interf {
        public static void m1() {

        }
}
class Test implements Interf {
        public void m1() {

        }
}
```

Ex 3: Valid but not overriding ✓
In normal java class parent and child classes, while overriding we can't reduce scope of access modifier then immediately, we will get compile time error but in the case of interface static method, it is valid because it is not overriding concept.

```
interface Interf {
        public static void m1() {

        }
}
class Test implements Interf {
        private static void m1() {

        }
}
```

Note: From 1.8 version onwards, we can declare static method inside interface. If we are allowed to declare static methods inside interface then we can declare main () method also inside interface and we can run interface directly from the command prompt.

```
interface Interf {
        public static void main() {
                System.out.println("Interface main method");
        }
}
```

Note: In Java 1.8 there are some new predefine functional interfaces provided by Java. These are present in java.util.function package.

- Predicate
- Function
- Consumer
- Supplier

  Etc.

Predicate

- Normally predicate means perform some conditional check and returns true and false based on that condition.
- A predicate is a boolean-valued function with a single argument to check a particular condition and the returns boolean value (true/ false).
- This Predicate interface introduced in 1.8 version (i.e., Predicate<T>).
- Predicate interface present in java.util.function package.
- It's a functional interface and it contains only one abstract method i.e., test ().

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

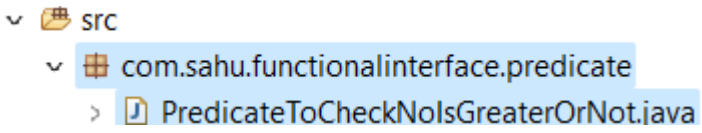
- Predicate is a functional interface that's why we can user Predicate to refer lambda expression.

Ex 1: Write a predicate to check the given integer is greater than 10 or not.

```
public boolean test (Integer i) {
        if (i >10) {
                return true;
        }
        else {
                return false;
        }
}
```

```
(Integer i) -> {
        if (i > 10)
                return true;
        else
                return false;
};
```

```
Predicate<Integer> p = i -> (i >10);
System.out.println (p.test(100)); true
System.out.println(p.test(7)); false
```



- Create the package and add the selected java files, then places the following code with in their respective file.

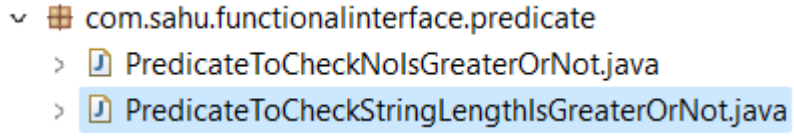
PredicateToCheckNoIsGreaterOrNot.java

```
package com.sahu.functionalinterface.predicate;

import java.util.function.Predicate;

public class PredicateToCheckNoIsGreaterOrNot {

	public static void main(String[] args) {
		Predicate<Integer> predicate = i -> i>10;
		System.out.println(predicate.test(100));
		System.out.println(predicate.test(5));
	}

}
```

Note: The above predicate takes Integer as type compulsory we have to pass int or integer values as argument to the test method. By mistake if we are providing any other type immediately, we will get compile time error.
If we give predicate.test("Durga") then below compile time error we will get
CE: The method test(Integer) in the type Predicate<Integer> is not applicable for the arguments (String)

Ex 2: Write a predicate to check the length of given string is greater than 5 or not.

Predicate<String> p = s -> s.length() > 3;

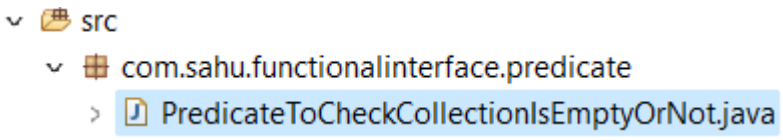


- Add the selected java files, then places the following code with in their respective file.

PredicateToCheckStringLengthIsGreaterOrNot.java

```
package com.sahu.functionalinterface.predicate;

import java.util.function.Predicate;

public class PredicateToCheckStringLengthIsGreaterOrNot {

	public static void main(String[] args) {
		Predicate<String> predicate = s -> s.length() > 5;
		System.out.println(predicate.test("abcdef"));
		System.out.println(predicate.test("abc"));
	}

}
```

Ex 3: write a predicate to check whether the given collection is empty or not.



- Add the selected java files, then places the following code with in their respective file.

PredicateToCheckCollectionIsEmptyOrNot.java

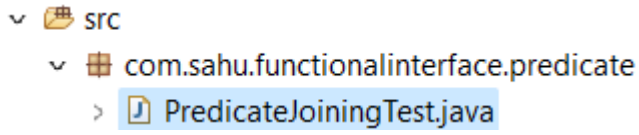
```
package com.sahu.functionalinterface.predicate;

import java.util.ArrayList;
import java.util.Collection;
import java.util.function.Predicate;

public class PredicateToCheckCollectionIsEmptyOrNot {

	public static void main(String[] args) {
		Predicate<Collection<?>> predicate = c -> c.isEmpty();
		ArrayList<String> l1 = new ArrayList<>();
		l1.add("A");
		System.out.println(predicate.test(l1));
```

```
                ArrayList<String> l2 = new ArrayList<>();
                System.out.println(predicate.test(l2));
        }

}
```

Predicate Joining

- It's possible to join predicates into a single predicate by using the following methods.
  - and ()
  - or ()
  - negate ()
- These are exactly same as logical AND, OR complement operators.

Ex:



- Add the selected java files, then places the following code with in their respective file.

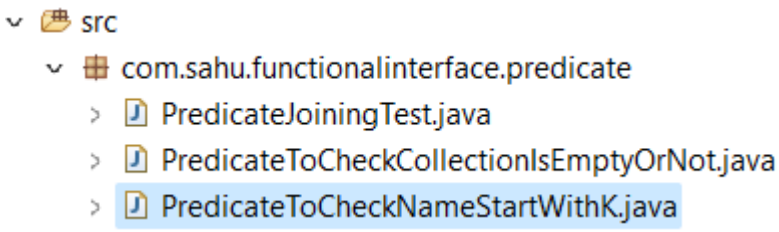
PredicateJoiningTest.java

```
package com.sahu.functionalinterface.predicate;

import java.util.function.Predicate;

public class PredicateJoiningTest {

        public static void main(String[] args) {
                int[] x = {0,5,10,15,20,25,30};
                Predicate<Integer> predicate1 = i -> i>10;
                Predicate<Integer> predicate2 = i -> i%2==0;
                System.out.println("The numbes greater than 10 are: ");
                checkTheNumher(predicate1, x);
                System.out.println("The even numbers are: ");
                checkTheNumher(predicate2, x);
                System.out.println("The numbes not greater than 10 are: ");
                checkTheNumher(predicate1.negate(), x);
```

```
                System.out.println("The numbes greater than 10 and even are:
");
                checkTheNumher(predicate1.and(predicate2), x);
                System.out.println("The numbes greater than 10 or even are:
");
                checkTheNumher(predicate1.or(predicate2), x);
        }

        public static void checkTheNumher(Predicate<Integer> predicate,
int[] x) {
                for (int i : x) {
                        if(predicate.test(i)) {
                                System.out.println(i);
                        }
                }
        }

}
```

Practice Examples

Ex 1: Program to display names starts with 'K' by using predicate



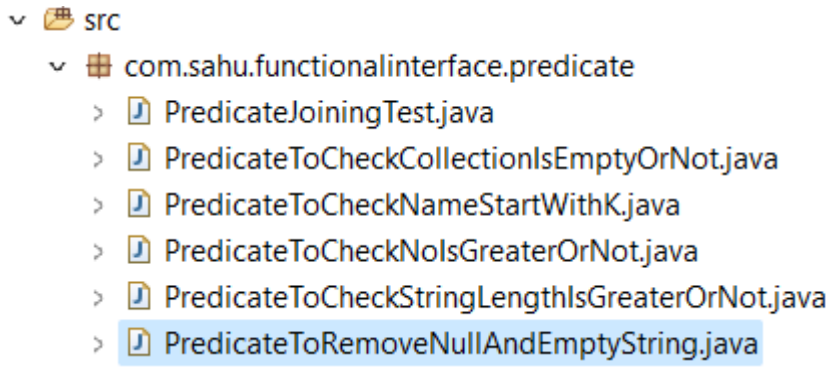
- Add the selected java files, then places the following code with in their respective file.

PredicateToCheckNameStartWithK.java

```
package com.sahu.functionalinterface.predicate;

import java.util.function.Predicate;

public class PredicateToCheckNameStartWithK {

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

```
                String[] names = { "Sunny", "Kajal", "Malika", "Katrina",
"Kareena" };
                Predicate<String> startsWithK = s -> s.charAt(0) == 'K';
                System.out.println("The names start with K are: ");
                for (String name : names) {
                        if (startsWithK.test(name)) {
                                System.out.println(name);
                        }
                }
        }

}
```

Ex 2: Predicate example to remove null values and empty strings from the given list



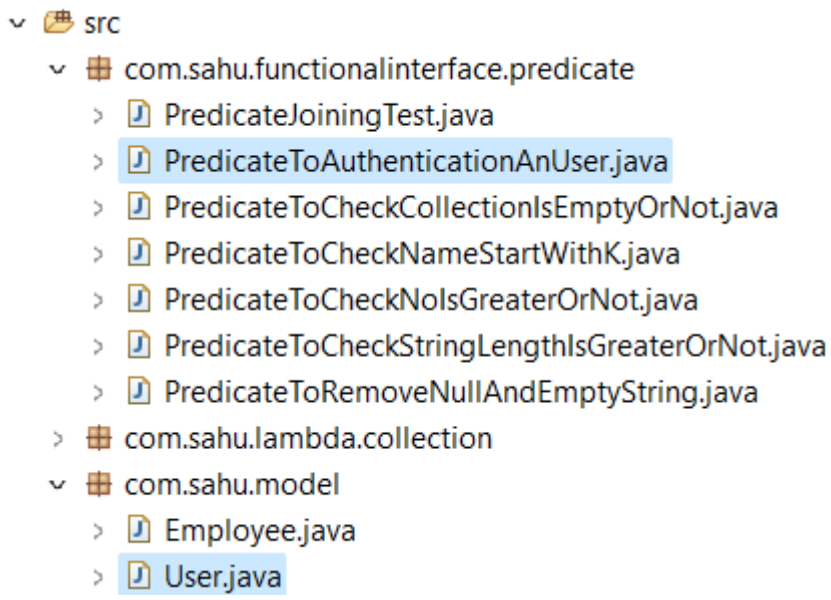
- Add the selected java files, then places the following code with in their respective file.

PredicateToRemoveNullAndEmptyString.java

```
package com.sahu.functionalinterface.predicate;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class PredicateToRemoveNullAndEmptyString {

        public static void main(String[] args) {
                String[] names = { "Durga", " ", null, "Ravi", "", "Shiva", null };
```

```
                Predicate<String> predicate = s -> s!=null && !s.isEmpty();
                List<String> nameList = new ArrayList<>();
                for (String name : names) {
                        if(predicate.test(name)) {
                                nameList.add(name);
                        }
                }
                System.out.println("The list of valid names");
                System.out.println(nameList);
        }

}
```

Ex 3: Program for User Authentication by using Predicate



- Add the selected java files, then places the following code with in their respective file.

User.java

```
package com.sahu.model;

public class User {
        private String userName;
        private String password;
```

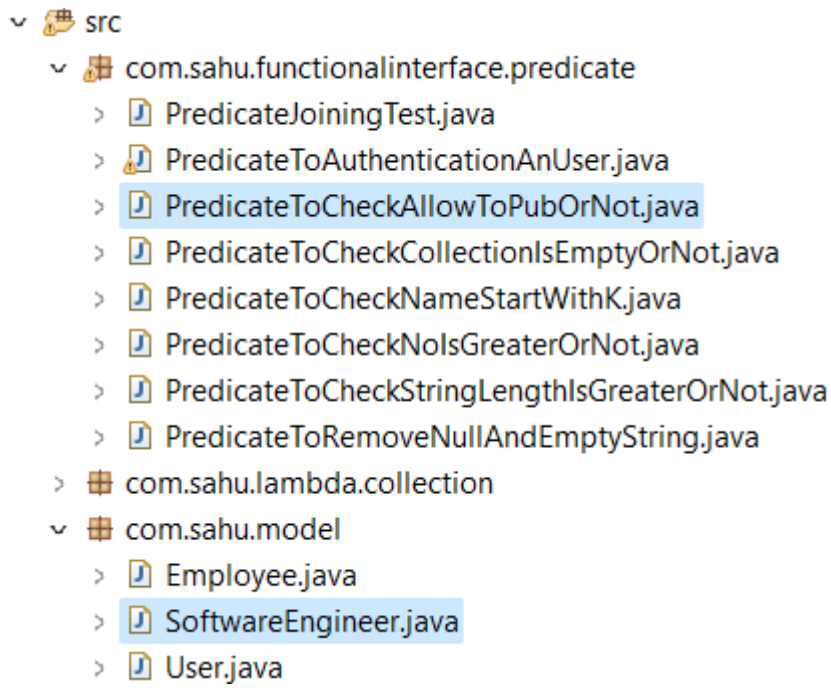
```
	public User(String userName, String password) {
		this.userName = userName;
		this.password = password;
	}

	public String getUserName() {
		return userName;
	}

	public void setUserName(String userName) {
		this.userName = userName;
	}

	public String getPassword() {
		return password;
	}

	public void setPassword(String password) {
		this.password = password;
	}
}
```

PredicateToAuthenticationAnUser.java

```
package com.sahu.functionalinterface.predicate;

import java.util.Scanner;
import java.util.function.Predicate;

import com.sahu.model.User;

public class PredicateToAuthenticationAnUser {

	public static void main(String[] args) {
		Predicate<User> predicateUser = user ->
user.getUserName().equals("durga") && user.getPassword().equals("java");
		Scanner scanner = new Scanner(System.in);
		System.out.print("Enter username: ");
		String userName = scanner.next();
		System.out.print("Enter password: ");
		String password = scanner.next();
```

```
                String password = scanner.next();
                User user = new User(userName, password);
                if (predicateUser.test(user)) {
                        System.out.println("Valid user, you can get all services");
                } else {
                        System.out.println("Invalid user, please login again");
                }
        }

}
```

Ex 4: Program to check whether Software Engineer is allowed into pub or not by using Predicate?



- Add the selected java files, then places the following code with in their respective file.

SoftwareEngineer.java

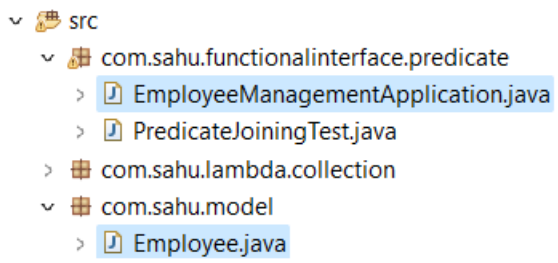
```
package com.sahu.model;

public class SoftwareEngineer {
        private String name;
        private Integer age;
        private Boolean isHavingGF;
```

```
	public SoftwareEngineer(String name, Integer age, Boolean
isHavingGF) {
		this.name = name;
		this.age = age;
		this.isHavingGF = isHavingGF;
	}

	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 Boolean getIsHavingGF() {
		return isHavingGF;
	}

	public void setIsHavingGF(Boolean isHavingGF) {
		this.isHavingGF = isHavingGF;
	}

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

}
```

PredicateToCheckAllowToPubOrNot.java

```
package com.sahu.functionalinterface.predicate;

import java.util.function.Predicate;

import com.sahu.model.SoftwareEngineer;

public class PredicateToCheckAllowToPubOrNot {

	public static void main(String[] args) {
		SoftwareEngineer[] softwareEngineers = { new
SoftwareEngineer("Durga", 60, false),
				new SoftwareEngineer("Sunil", 25, true), new
SoftwareEngineer("Sayan", 26, true),
				new SoftwareEngineer("Subbu", 28, false), new
SoftwareEngineer("Ravi", 19, true) };

		Predicate<SoftwareEngineer> allowed = softwareEngineer ->
softwareEngineer.getAge() >= 18
				&& softwareEngineer.getIsHavingGF();
		for (SoftwareEngineer softwareEngineer : softwareEngineers) {
			if (allowed.test(softwareEngineer)) {
				System.out.println(softwareEngineer);
			}
		}
	}

}
```

Ex: Employee Management Application



- Add the selected java files, then places the following code with in their respective file.

Employee.java

```
package com.sahu.model;

public class Employee {
	private String name;
	private String designation;
	private Double salary;
	private String city;

	public Employee(String name, String designation, Double salary,
String city) {
		this.name = name;
		this.designation = designation;
		this.salary = salary;
		this.city = city;
	}

	public String getName() {
		return name;
	}

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

	public String getDesignation() {
		return designation;
	}

	public void setDesignation(String designation) {
		this.designation = designation;
	}

	public Double getSalary() {
		return salary;
	}

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

```
	public String getCity() {
		return city;
	}

	public void setCity(String city) {
		this.city = city;
	}

	@Override
	public String toString() {
		return String.format("(%s, %s, %.2f, %s)", name, designation,
salary, city);
	}
}
```

SoftwareEngineer.java

```
package com.sahu.functionalinterface.predicate;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

import com.sahu.model.Employee;

public class EmployeeManagementApplication {

	public static void main(String[] args) {
		List<Employee> employees = new ArrayList<>();
		populateEmployees(employees);
		System.out.println(employees);

		Predicate<Employee> managerDesgPredicate = employee ->
employee.getDesignation().equals("Manager");
		System.err.println("\nEmployees who are Manager");
		displayEmployee(managerDesgPredicate, employees);

		Predicate<Employee> bangaloreCityPredicate = employee ->
employee.getCity().equals("Bangalore");
		System.err.println("\nEmployees who are from Bangalore");
```

```
                displayEmployee(bangaloreCityPredicate, employees);

                Predicate<Employee> salaryLessThan20000Predicate =
employee -> employee.getSalary() < 20000.0;
                System.err.println("\nEmployees whose salary is less than
20000");
                displayEmployee(salaryLessThan20000Predicate, employees);

                Predicate<Employee> pinkSlipPredicate =
managerDesgPredicate.and(bangaloreCityPredicate);
                System.err.println("\nEmployees who can get Pink slip");
                displayEmployee(pinkSlipPredicate, employees);

                System.err.println("\nEmployees who are manger or salary is
less than 20000");

        displayEmployee(managerDesgPredicate.or(salaryLessThan20000Pred
icate), employees);

                System.err.println("\nEmployees who are not Manager");
                displayEmployee(managerDesgPredicate.negate(), employees);
        }

        public static void populateEmployees(List<Employee> employees) {
                employees.add(new Employee("Durga", "CEO", 300000.0,
"Hyderbad"));
                employees.add(new Employee("Sunny", "Manager", 20000.0,
"Hyderbad"));
                employees.add(new Employee("Mallika", "Manager", 20000.0,
"Bangalore"));
                employees.add(new Employee("Kareena", "Lead", 15000.0,
"Hyderbad"));
                employees.add(new Employee("Katrina", "Lead", 15000.0,
"Bangalore"));
                employees.add(new Employee("Anushka", "Developer",
10000.0, "Hyderbad"));
                employees.add(new Employee("Kanushka", "Developer",
10000.0, "Hyderbad"));
                employees.add(new Employee("Soumya", "Developer",
10000.0, "Bangalore"));
```

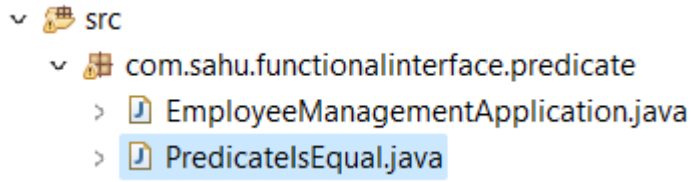
```
                employees.add(new Employee("Ramya", "Developer", 10000.0,
"Bangalore"));
        }

        public static void displayEmployee(Predicate<Employee>
empPredicate, List<Employee> employees) {
                for (Employee employee : employees) {
                        if (empPredicate.test(employee)) {
                                System.out.println(employee);
                        }
                }
        }
}
```

Predicate isEqual()

- Predicate having a static method call isEqual().
- It returns a predicate that tests if two arguments are equal or not.

```
static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
                ? Objects::isNull
                : object -> targetRef.equals(object);
}
```

Ex:



- Add the selected java files, then places the following code with in their respective file.

PredicateIsEqual.java

```
package com.sahu.functionalinterface.predicate;

import java.util.function.Predicate;

import com.sahu.model.Employee;
```

```
public class PredicateIsEqual {

	public static void main(String[] args) {
		Predicate<String> predicateIsEqual =
Predicate.isEqual("WEBTECHSOLUTION");

	System.out.println(predicateIsEqual.test("WEBTECHSOLUTION"));
		System.out.println(predicateIsEqual.test("MALLIKA"));

		Predicate<Employee> isCEO = Predicate.isEqual(new
Employee("Durga", "CEO", 300000.0, "Hyderbad"));
		Employee employee1 = new Employee("Durga", "CEO",
300000.0, "Hyderbad");
		Employee employee2 = new Employee("Sunny", "Manager",
20000.0, "Hyderbad");
		System.out.println(isCEO.test(employee1));
		System.out.println(isCEO.test(employee2));
	}

}
```

But to work the employee with isEqual we have to override the equals method in Employees class as like below.

Employee.java

```
	@Override
	public boolean equals(Object obj) {
		Employee3 employee = (Employee3) obj;
		return (name.equals(employee.getName()) &&
designation.equals(employee.getDesignation())
				&& salary.equals(employee.getSalary()) &&
city.equals(employee.getCity())) ? true : false;
	}
```

Predicate Practice Bits

Q. Which abstract method present in Predicate interface?

Ans. Predicate functional interface contains only one abstract method: test().

Q. Which static method present in Predicate interface?
Ans. Predicate functional interface contains only one static method: isEqual().

Q. Which default methods present in Predicate interface?
Ans. Predicate Functional interface contains the following 3 default methods: and(), or(), not()

Q. What is Predicate interface declaration?
Ans. Predicate interface can take only one Type parameter which represents only input type. We are not required to specify return type because return type is always boolean type.

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

Q. Write a Predicate to check whether the given Integer is divisible by 10 or not?
Ans. Predicate<Integer> p = i -> i%10 == 10;

Q. Explain about Predicate functional interface?
Ans. Following points are valid for Predicate functional interface,

a. Predicate Functional interface present in java.util.function package.
b. It is introduced in java 1.8 version.
c. We can use Predicate to implement conditional checks.
d. It is possible to join 2 predicates into a single predicate also.

Q. Write a Predicate to check whether the given user is admin or not?
Ans. Predicate<User> p = user -> user.getRole().equals("Admin");

Q. Is negate() method taking any argument?
Ans. negate() method won't take any argument

Function

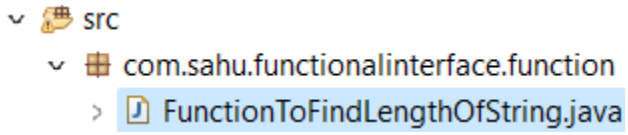
- Function is exactly same as predicate except but only difference is it can return any type of value need not to be boolean value.
- To implement function oracle people introduced Function interface in 1.8 version.
- Function interface present in java.util.function package.
- Function interface contains only one method i.e., apply ()

- This method accepts one argument and produces a result.
- T is input type and R is return type.

```
@FunctionalInterface
public interface Function<T, R> {
        R apply(T t);
}
```

Note: Function is a functional interface and hence it can refer lambda expression.

Ex: Write a function to find length of given input string.

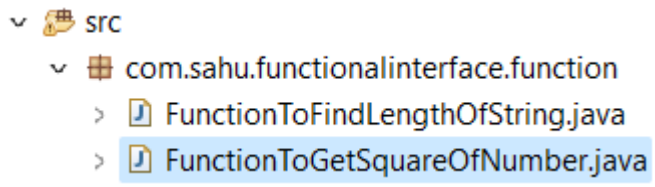


- Add the selected java files, then places the following code with in their respective file.

FunctionToFindLengthOfString.java

```
package com.sahu.functionalinterface.function;

import java.util.function.Function;

public class FunctionToFindLengthOfString {

        public static void main(String[] args) {
                Function<String, Integer> findLengthOfString = str ->
str.length();
                System.out.println(findLengthOfString.apply("durga"));

        System.out.println(findLengthOfString.apply("WEBTECHSOLUTION"));
        }

}
```

Ex: Write a function to return the square of the given number by using lambda expression.



- Add the selected java files, then places the following code with in their respective file.

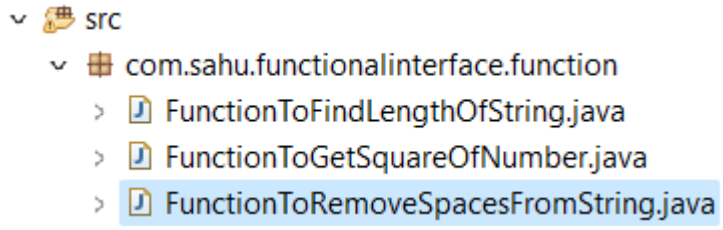
FunctionToGetSquareOfNumber.java

```
package com.sahu.functionalinterface.function;

import java.util.function.Function;

public class FunctionToGetSquareOfNumber {

	public static void main(String[] args) {
		Function<Integer, Integer> squareOfNumber = num -> num *
num;
		System.out.println(squareOfNumber.apply(5));
		System.out.println(squareOfNumber.apply(10));
	}

}
```

Differences between predicate and function

Predicate	Function
To implement conditional checks, we should go for predicate.	To perform certain operation and to return some result we Should go for function.
Predicate can take one type parameter which represents input argument type. Predicate<T>	Function can take 2 type Parameters. First one represents input argument type and second one represents return Type. Function<T,R>
Predicate interface definesonly 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.

Ex 1: Program to remove spaces present in the given String by using Function.

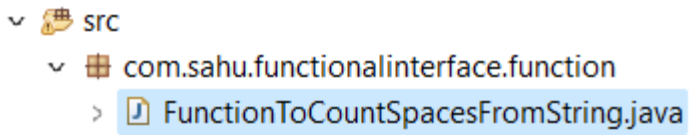


- Add the selected java files, then places the following code with in their respective file.

FunctionToRemoveSpacesFromString.java

```
package com.sahu.functionalinterface.function;

import java.util.function.Function;

public class FunctionToRemoveSpacesFromString {

	public static void main(String[] args) {
		Function<String, String> removeSpacesFunction = str ->
str.replaceAll(" ", "");
		System.out.println(removeSpacesFunction.apply("Webtech
solution Hyderbad"));
	}

}
```

Ex 2: Program to find number of spaces present in the given String by using Function.



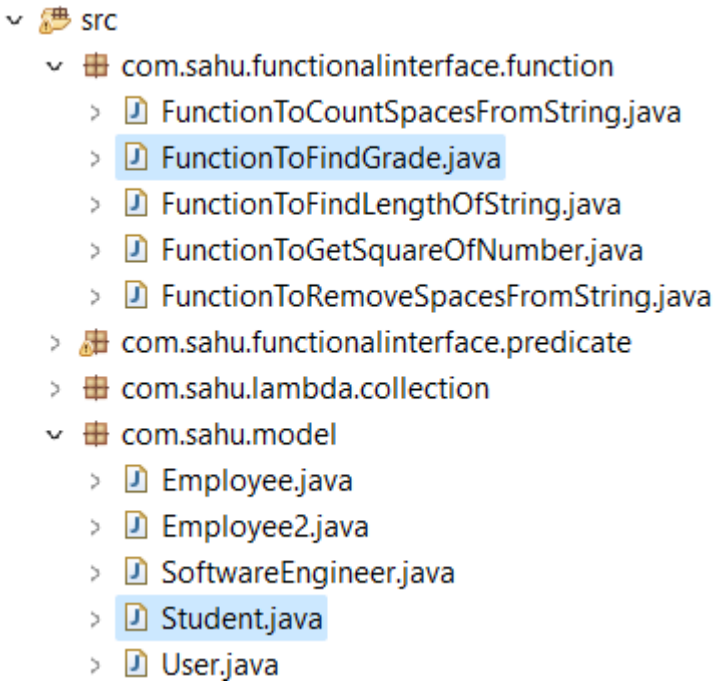
- Add the selected java files, then places the following code with in their respective file.

FunctionToRemoveSpacesFromString.java

```
package com.sahu.functionalinterface.function;
```

```java
import java.util.function.Function;

public class FunctionToCountSpacesFromString {

	public static void main(String[] args) {
		Function<String, Integer> countSpacesFunction = str ->
str.length() - str.replaceAll(" ", "").length();
		System.out.println(countSpacesFunction.apply("Webtech
solution Hyderbad"));
	}

}
```

Ex 3: Program to find student grade by using Function.



- Add the selected java files, then places the following code with in their respective file.

Student.java

```java
package com.sahu.model;

public class Student {
	private String name;
	private Integer marks;
```

```
	public Student(String name, Integer marks) {
		this.name = name;
		this.marks = marks;
	}

	public String getName() {
		return name;
	}

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

	public Integer getMarks() {
		return marks;
	}

	public void setMarks(Integer marks) {
		this.marks = marks;
	}

}
```

FunctionToFindGrade.java

```
package com.sahu.functionalinterface.function;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

import com.sahu.model.Student;

public class FunctionToFindGrade {

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

		Function<Student, String> togetGradeFunction = student -> {
```

```
                int marks = student.getMarks();
                if (marks >= 80) {
                        return "A [Distinction]";
                } else if (marks >= 60) {
                        return "B [First class]";
                } else if (marks >= 50) {
                        return "C [Second class]";
                } else if (marks >= 35) {
                        return "D [Third class]";
                } else {
                        return "E [Fail]";
                }
        };

        for (Student stud : students) {
                System.out.println("Student Name: "+stud.getName());
                System.out.println("Student Marks: "+stud.getMarks());
                System.out.println("Student Grade:
"+togetGradeFunction.apply(stud));
                System.out.println();
        }
    }

    private static void populateStudent(List<Student> students) {
        students.add(new Student("Sunny", 100));
        students.add(new Student("Bunny", 65));
        students.add(new Student("Chinny", 55));
        students.add(new Student("Vinny", 45));
        students.add(new Student("Pinny", 25));
    }

}
```

Ex 4: Program to find Students grade using function and whose marks are >=60 using Predicate.

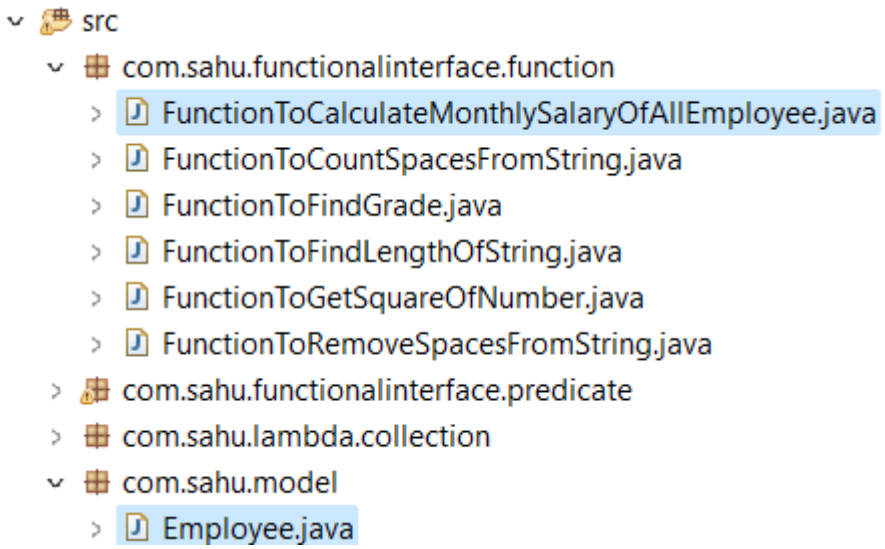
FunctionToFindGrade.java

```
package com.sahu.functionalinterface.function;

import java.util.ArrayList;
```

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

import com.sahu.model.Student;

public class FunctionToFindGrade {

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

		Function<Student, String> togetGradeFunction = student -> {
			int marks = student.getMarks();
			if (marks >= 80) {
				return "A [Distinction]";
			} else if (marks >= 60) {
				return "B [First class]";
			} else if (marks >= 50) {
				return "C [Second class]";
			} else if (marks >= 35) {
				return "D [Third class]";
			} else {
				return "E [Fail]";
			}
		};

		Predicate<Student> markPredicate = student ->
student.getMarks() >= 60;

		for (Student stud : students) {
			if (markPredicate.test(stud)) {
				System.out.println("Student Name: " +
stud.getName());
				System.out.println("Student Marks: " +
stud.getMarks());
				System.out.println("Student Grade: " +
togetGradeFunction.apply(stud));
				System.out.println();
			}
```

```
                }
        }

        private static void populateStudent(List<Student> students) {
                students.add(new Student("Sunny", 100));
                students.add(new Student("Bunny", 65));
                students.add(new Student("Chinny", 55));
                students.add(new Student("Vinny", 45));
                students.add(new Student("Pinny", 25));
        }

}
```

Ex 5: Program to find total monthly salary of all employees by using Function.



- Add the selected java files, then places the following code with in their respective file.

Employee.java

```
package com.sahu.model;

public class Employee {
        private String name;
        private Double salary;

        public Employee(String name, Double salary) {
                this.name = name;
```

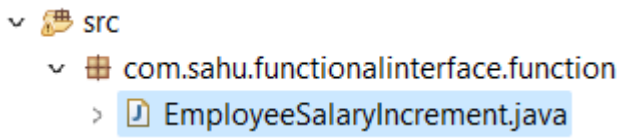
```
		this.salary = salary;
	}

	public String getName() {
		return name;
	}

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

	public Double getSalary() {
		return salary;
	}

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

	@Override
	public String toString() {
		return name+" : "+salary;
	}
}
```

FunctionToCalculateMonthlySalaryOfAllEmployee.java

```
package com.sahu.functionalinterface.function;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;

import com.sahu.model.Employee;

public class FunctionToCalculateMonthlySalaryOfAllEmployee {

	public static void main(String[] args) {
		List<Employee> employees = new ArrayList<>();
		populateemployees(employees);
		Function<List<Employee>, Double> totalSalaryFunction =
```

```
employeeList -> {
                        double total = 0;
                        for (Employee employee : employeeList) {
                                total = total + employee.getSalary();
                        }
                        return total;
                };
                System.out.println("The total salry of this month : " +
totalSalaryFunction.apply(employees));
        }

        private static void populateemployees(List<Employee> employees) {
                employees.add(new Employee("Sunny", 1000.0));
                employees.add(new Employee("Bunny", 2000.0));
                employees.add(new Employee("Chinny", 3000.0));
                employees.add(new Employee("Pinny", 4000.0));
                employees.add(new Employee("Vinny", 5000.0));
        }

}
```

Ex: Program to perform salary increment for employees by using Predicate & Function.



- Add the selected java files, then places the following code with in their respective file.

EmployeeSalaryIncrement.java

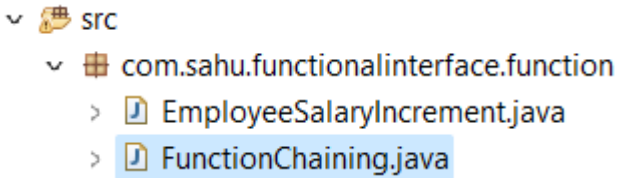
```
package com.sahu.functionalinterface.function;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Function;
import java.util.function.Predicate;

import com.sahu.model.Employee;
```

```
import com.sahu.model.Employee;

public class EmployeeSalaryIncrement {

	public static void main(String[] args) {
		List<Employee> employees = new ArrayList<>();
		populateemployees(employees);
		System.out.println("Before Increment:");
		System.out.println(employees);

		Predicate<Employee> eligiableForIncrement = employee ->
employee.getSalary() < 3500;

		Function<Employee, Employee> updatedEmployeeSalary =
employee -> {
			employee.setSalary(employee.getSalary() + 477.0);
			return employee;
		};

		List<Employee> updatedEmployees = new ArrayList<>();
		for (Employee emp : employees) {
			if (eligiableForIncrement.test(emp)) {
				updatedEmployeeSalary.apply(emp);
				updatedEmployees.add(emp);
			}
		}
		System.out.println("After increment:");
		System.out.println(updatedEmployees);
	}

	private static void populateemployees(List<Employee> employees) {
		employees.add(new Employee("Sunny", 1000.0));
		employees.add(new Employee("Bunny", 2000.0));
		employees.add(new Employee("Chinny", 3000.0));
		employees.add(new Employee("Pinny", 4000.0));
		employees.add(new Employee("Vinny", 5000.0));
	}

}
```

Function Chaining

- We can combine multiple functions into a one function to form more complex functions.
- For this Function interface defines the following 2 default methods,
  - f1.andThen(f2): First f1 will be applied and then for the result f2 will be applied.
  - f1.compose(f2): First f2 will be applied and then for the result f1 will be applied.

Ex: Function Chaining



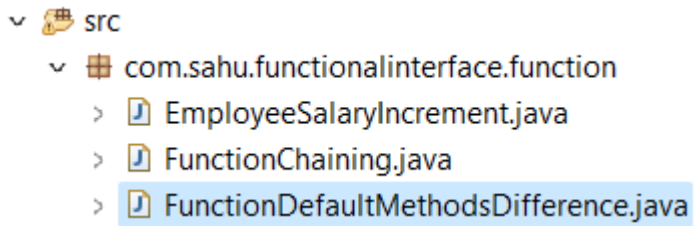
- Add the selected java files, then places the following code with in their respective file.

FunctionChaining.java

```
package com.sahu.functionalinterface.function;

import java.util.function.Function;

public class FunctionChaining {

	public static void main(String[] args) {
		Function<String, String> toMakeUppercase = str ->
str.toUpperCase();
		Function<String, String> toMakeSubString = str ->
str.substring(0, 9);
		System.out.println("The result of toMakeUppercase:
"+toMakeUppercase.apply("Aishwaryaabhi"));
		System.out.println("The result of toMakeSubString:
"+toMakeSubString.apply("Aishwaryaabhi"));
		System.out.println("The result of
toMakeUppercase.andThen(toMakeSubString):
"+toMakeUppercase.andThen(toMakeSubString).apply("Aishwaryaabhi"));
		System.out.println("The result of
toMakeUppercase.compose(toMakeSubString):
```

```
"+toMakeUppercase.compose(toMakeSubString).apply("Aishwaryaabhi"));
        }

}
```

Ex: Program to Demonstrate the difference between andThen() and compose().

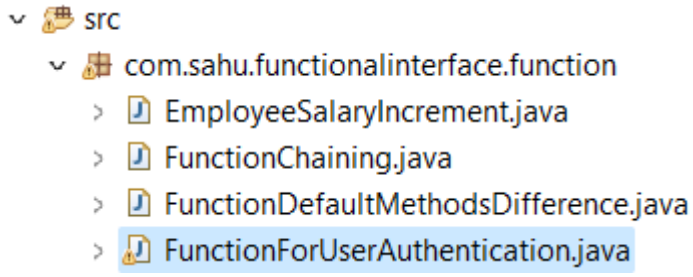


- Add the selected java files, then places the following code with in their respective file.

FunctionDefaultMethodsDifference.java

```
package com.sahu.functionalinterface.function;

import java.util.function.Function;

public class FunctionDefaultMethodsDifference {

        public static void main(String[] args) {
                Function<Integer, Integer> doubleOfNumber = num -> num +
num;
                Function<Integer, Integer> cubeOfNumber = num -> num *
num * num;
                System.out.println(doubleOfNumber.apply(2));
                System.out.println(cubeOfNumber.apply(2));

        System.out.println(doubleOfNumber.andThen(cubeOfNumber).apply(
2));
                System.out.println(doubleOfNumber.compose
(cubeOfNumber).apply(2));
        }

}
```

Ex: Program for user authentication by using Function chaining



- Add the selected java files, then places the following code with in their respective file.

FunctionForUserAuthentication.java

```
package com.sahu.functionalinterface.function;

import java.util.Scanner;
import java.util.function.Function;

public class FunctionForUserAuthentication {

	public static void main(String[] args) {
		Function<String, String> toMakeLowerCase = str ->
str.toLowerCase();
		Function<String, String> toMakeSubString = str ->
str.substring(0, 5);

		Scanner scanner = new Scanner(System.in);
		System.out.print("Enter user name: ");
		String userName = scanner.next();

		System.out.print("Enter password: ");
		String password = scanner.next();


	if(toMakeLowerCase.andThen(toMakeSubString).apply(userName).eq
uals("durga") && password.equals("java")) {
			System.out.println("Valid user");
		}
		else {
			System.out.println("Invalid user");
```

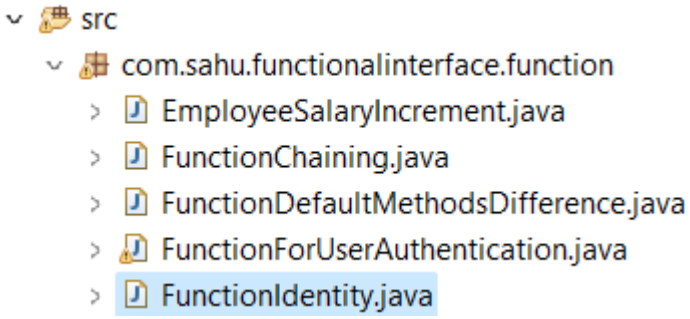
```
        }
    }

}
```

Function interface Static Method: identity()

- Function interface contains a static method identity().

```
static <T> Function<T, T> identity() {
        return t -> t;
}
```

- This method returns a function that always returns its input argument.



- Add the selected java files, then places the following code with in their respective file.

FunctionIdentity.java

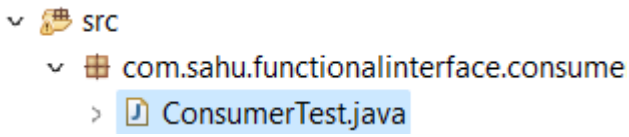
```
package com.sahu.functionalinterface.function;

import java.util.function.Function;

public class FunctionIdentity {

    public static void main(String[] args) {
        Function<String, String> functionIdentity = Function.identity();
        String result = functionIdentity.apply("durga");
        System.out.println(result);
    }

}
```

Consumer

- Sometimes our requirement is we have to provide some input value, perform certain operation, but not required to return anything, then we should go for Consumer. i.e. Consumer can be used to consume object and perform certain operation.
- Consumer Functional Interface contains one abstract method accept().
- This also introduced in java 1.8 version.
- Consumer interface present in java.util.function package.

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

Ex 1: Program for Consumer

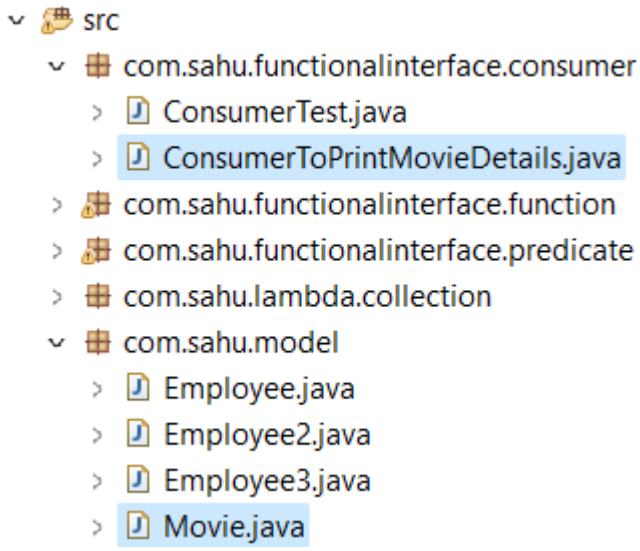


- Add the selected java files, then places the following code with in their respective file.

ConsumerTest.java

```
package com.sahu.functionalinterface.consumer;

import java.util.function.Consumer;

public class ConsumerTest {

    public static void main(String[] args) {
        Consumer<String> consumer = str -> System.out.println(str);
        consumer.accept("hello");
        consumer.accept("Webtech solution");
    }

}
```

Ex 2: Program to display movie information by using Consumer.



- Add the selected java files, then places the following code with in their respective file.

Movie.java

```
package com.sahu.model;

public class Movie {

	private String name;
	private String hero;
	private String heroine;

	public Movie(String name, String hero, String heroine) {
		super();
		this.name = name;
		this.hero = hero;
		this.heroine = heroine;
	}

	public String getName() {
		return name;
	}

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

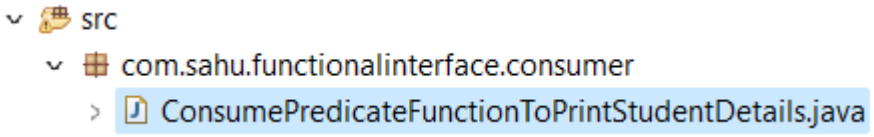
```
	public String getHero() {
		return hero;
	}

	public void setHero(String hero) {
		this.hero = hero;
	}

	public String getHeroine() {
		return heroine;
	}

	public void setHeroine(String heroine) {
		this.heroine = heroine;
	}

}
```

Movie.java

```
package com.sahu.functionalinterface.consumer;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

import com.sahu.model.Movie;

public class ConsumerToPrintMovieDetails {

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

		Consumer<Movie> printDetails = movie -> {
			System.out.println("Movie Name: "+movie.getName());
			System.out.println("Movie Hero: "+movie.getHero());
			System.out.println("Movie Heroine:
"+movie.getHeroine());
			System.out.println();
		};
```

```
                };

                for (Movie movie : movies) {
                        printDetails.accept(movie);
                }
        }

        private static void populateMovies(List<Movie> movies) {
                movies.add(new Movie("Bahubali", "Prabhas", "Anuskha"));
                movies.add(new Movie("Rayees", "Sharukh", "Sunny"));
                movies.add(new Movie("Dangal", "Ameer", "Ritu"));
                movies.add(new Movie("Sultan", "Salman", "Anushka"));
        }

}
```

Ex 3: Program to Display Student information by using Predicate, Function & Consumer.



- Add the selected java files, then places the following code with in their respective file.
- Use previous Student.java class.

ConsumePredicateFunctionToPrintStudentDetails.java

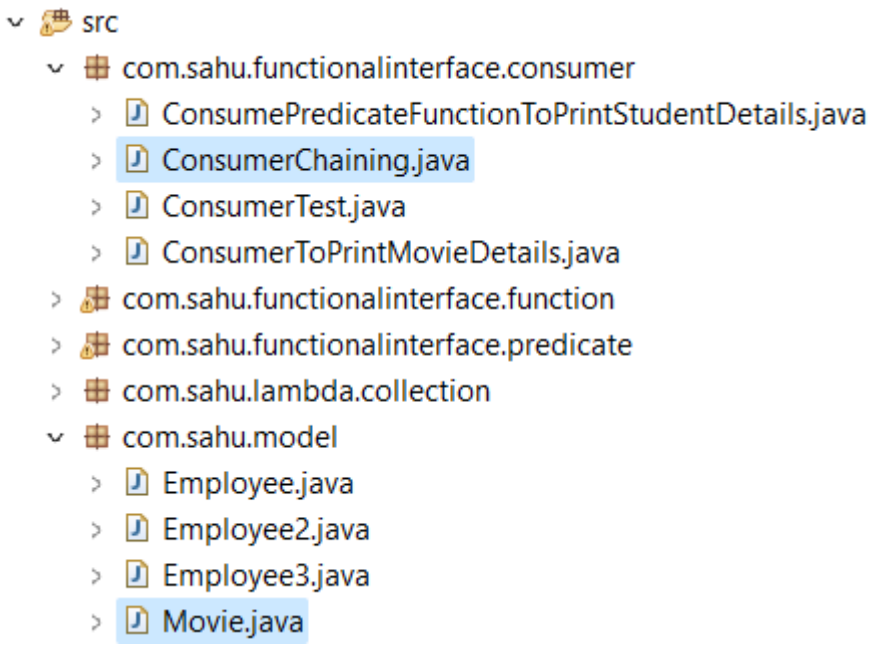
```
package com.sahu.functionalinterface.consumer;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;

import com.sahu.model.Student;

public class ConsumePredicateFunctionToPrintStudentDetails {
```

```
	public static void main(String[] args) {
		List<Student> students = new ArrayList<>();
		populateStudent(students);
		Predicate<Student> getStudent = stud -> stud.getMarks() >=
60;
		Function<Student, String> getGrade = stud -> {
			int marks = stud.getMarks();
			if (marks >= 80) {
				return "A";
			} else if (marks >= 60) {
				return "B";
			} else if (marks >= 50) {
				return "C";
			} else if (marks >= 35) {
				return "D";
			} else {
				return "E";
			}
		};
		Consumer<Student> printStudent = stud -> {
			System.out.println("Name: "+stud.getName());
			System.out.println("Mark: "+stud.getMarks());
			System.out.println("Grade: "+getGrade.apply(stud));
			System.out.println();
		};
		for (Student student : students) {
			if (getStudent.test(student)) {
				printStudent.accept(student);
			}
		}
	}

	private static void populateStudent(List<Student> students) {
		students.add(new Student("Sunny", 100));
		students.add(new Student("Bunny", 65));
		students.add(new Student("Chinny", 55));
		students.add(new Student("Vinny", 45));
		students.add(new Student("Pinny", 25));
	}
}
```

Consumer Chaining

- Just like Predicate Chaining and Function Chaining, Consumer Chaining is also possible.
- For this Consumer Functional Interface contains default method andThen().
- c1.andThen(c2).andThen(c3).accept(s): First Consumer c1 will be applied followed by c2 and c3.

Ex: Program for Consumer Chaining.



- Add the selected java files, then places the following code with in their respective file.

Movie.java

```
package com.sahu.model;

public class Movie {

	private String name;
	private String result;

	public Movie(String name, String result) {
		super();
		this.name = name;
		this.result = result;
```

```
		this.result = result;
	}

	public String getName() {
		return name;
	}

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

	public String getResult() {
		return result;
	}

	public void setResult(String result) {
		this.result = result;
	}

}
```

Movie.java

```
package com.sahu.functionalinterface.consumer;

import java.util.function.Consumer;

import com.sahu.model.Movie;

public class ConsumerChaining {

	public static void main(String[] args) {
		Consumer<Movie> returnName = movie ->
System.out.println("Movie "+movie.getName()+" is ready to release.");
		Consumer<Movie> returnResult = movie ->
System.out.println("Movie "+movie.getName()+" is "+movie.getResult());
		Consumer<Movie> stroingInDB = movie ->
System.out.println("Movie "+movie.getName()+" information storing in
database.");
		Consumer<Movie> chainedCosnumer =
returnName.andThen(returnResult).andThen(stroingInDB);
```

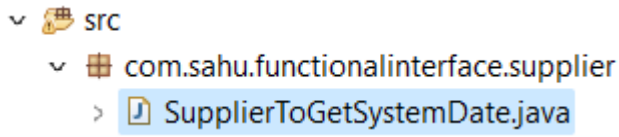
```
                Movie movie = new Movie("Bahubali", "Hit");
                chainedCosnumer.accept(movie);
        }

}
```

Supplier

- Sometimes our requirement is we won't provide any input but we have to get some value based on some operation.
- Like can you supply system date, can you supply Student object, can you supply random name, random OTP, random password, etc.
- For this type of requirements, we should go for Supplier. Supplier can be used to supply items (objects).
- Supplier won't take any input and it will always supply objects.
- Supplier Functional Interface contains only one method get(). This method won't accept any input always it returns R type.
- Supplier Functional interface does not contain any default and static methods.

```
@FunctionalInterface
public interface Supplier<T> {
        T get();
}
```

Ex 1: Program for Supplier to supply System Date.



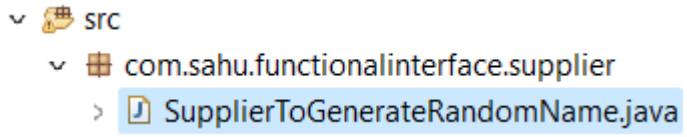
- Add the selected java files, then places the following code with in their respective file.

SupplierToGetSystemDate.java

```
package com.sahu.functionalinterface.supplier;

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

```
public class SupplierToGetSystemDate {

	public static void main(String[] args) {
		Supplier<Date> systemDateSupplier = () -> new Date();
		System.out.println(systemDateSupplier.get());
	}

}
```

Ex 2: Program For Supplier to generate Random name.

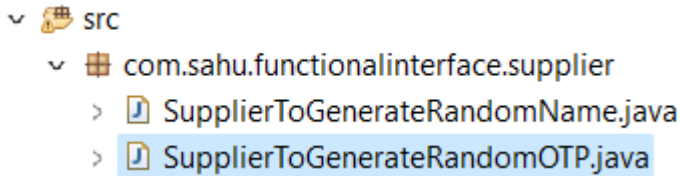


- Add the selected java files, then places the following code with in their respective file.

SupplierToGenerateRandomName.java

```
package com.sahu.functionalinterface.supplier;

import java.util.function.Supplier;

public class SupplierToGenerateRandomName {

	public static void main(String[] args) {
		Supplier<String> randomNameSupplier = () -> {
			String[] names = { "Sunny", "Bunny", "Chinny", "Vinny" };
			int randomNum = (int) (Math.random() * 4);
			return names[randomNum];
		};
		System.out.println(randomNameSupplier.get());
		System.out.println(randomNameSupplier.get());
		System.out.println(randomNameSupplier.get());
		System.out.println(randomNameSupplier.get());
		System.out.println(randomNameSupplier.get());
	}

}
```

Ex 3: Program For Supplier to supply 6-digit Random OTP.

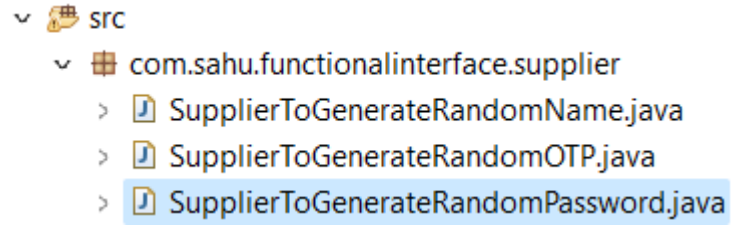


- Add the selected java files, then places the following code with in their respective file.

SupplierToGenerateRandomOTP.java

```
package com.sahu.functionalinterface.supplier;

import java.util.function.Supplier;

public class SupplierToGenerateRandomOTP {

	public static void main(String[] args) {
		Supplier<String> randomOTPSupplier = () -> {
			String otp = "";
			for (int i = 0; i < 6; i++) {
				otp = otp + (int) (Math.random() * 10);
			}
			return otp;
		};
		System.out.println(randomOTPSupplier.get());
		System.out.println(randomOTPSupplier.get());
		System.out.println(randomOTPSupplier.get());
	}

}
```

Ex 4: Program For Supplier to supply Random Passwords.



Rules:

- Length should be 8 characters
- 2, 4, 6, 8 places only digits
- 1, 3, 5, 7 places only Uppercase alphabets and @, #, $ these special characters are allowed

- Add the selected java files, then places the following code with in their respective file.

SupplierToGenerateRandomOTP.java

```
package com.sahu.functionalinterface.supplier;

import java.util.function.Supplier;

public class SupplierToGenerateRandomPassword {

	public static void main(String[] args) {
		Supplier<String> randomPasswordSupplier = () -> {
			StringBuffer password = new StringBuffer();
			Supplier<Integer> getNumber = () -> (int)
(Math.random() * 10);

			String symboles =
"ABCDEFGHIJKLMNOPQRSTUVWXYZ@#$";
			Supplier<Character> getCharacter = () ->
symboles.charAt((int) (Math.random() * 29));

			for (int i = 1; i <= 8; i++) {
				if (i % 2 == 0) {
					password.append(getNumber.get());
				} else {
					password.append(getCharacter.get());
				}

				//Using ternary operator
				//password.append((i % 2 == 0) ? getNumber.get()
: getCharacter.get());
			}
			return password.toString();
		};
```

```
                System.out.println(randomPasswordSupplier.get());
                System.out.println(randomPasswordSupplier.get());
        }

}
```

Comparison table for Predicate, Function, Consumer, Supplier

Property	Predicate	Function	Supplier	Consumer
Purpose	To tale some input an perform some conditional checks	To take some input and perform required operation and returns the result	To consume some input and perform required operation. It won't return the result	To supply some value based on our requirement
Interface declaration	interface Predicate<T> { }	interface Function<T, R> { }	interface Consume<T> { }	interface Supplier<R> { }
Single abstract method	Public boolean test(T t);	public R apply(T t);	public void accept(T t)	public R get()
Default methods	and(), or(), negate()	andThen(), compose()	andThen()	NA
Static method	isEquals()	identity()	NA	NA

Two-Argument (Bi) Functional Interfaces

- Normal Functional interfaces (Predicate, Function and Consumer) can accept only one input argument.
- But sometimes our programming requirement is to accept two input arguments, then we should go for two-argument functional interfaces.
- The following functional interfaces can take two input arguments.
    - BiPredicate
    - BiFunction
    - BiConsumer

Note: For Supplier there is not Bi functional interface because it won't accept any argument.

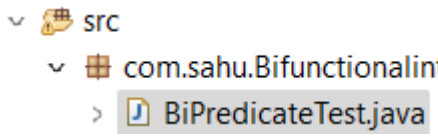
BiPredicate

- Normal Predicate can take only one input argument and perform some conditional check. Sometimes our programming requirement is we have to take two input arguments and perform some conditional check, for this requirement we should go for BiPredicate.
- BiPredicate is exactly same as Predicate except that it will take two input arguments.

```
@FunctionalInterface
public interface BiPredicate<T, U> {
        boolean test(T t, U u);
}
```

- Remaining default methods and(), or() , negate() also there but no static method.

Ex: To check the sum of 2 given integers is even or not by using BiPredicate.



- Add the selected java files, then places the following code with in their respective file.

BiPredicateTest.java

```
package com.sahu.Bifunctionalinterfaces;

import java.util.function.BiPredicate;

public class BiPredicateTest {

        public static void main(String[] args) {
                BiPredicate<Integer, Integer> checkTheSum = (a, b) -> (a + b) %
2 == 0;
                System.out.println(checkTheSum.test(10, 20));
                System.out.println(checkTheSum.test(15, 20));
        }

}
```

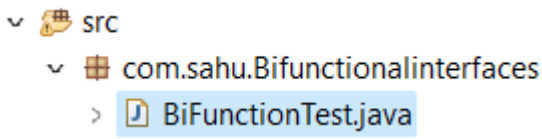
BiFunction

- Normal Function can take only one input argument and perform required operation and returns the result. The result need not be boolean type.
- But sometimes our programming requirement to accept two input values and perform required operation and should return the result. Then we should go for BiFunction.
- BiFunction is exactly same as Function except that it will take two input arguments.

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
        R apply(T t, U u);
}
```

- Only one default method is present i.e. andThen().

Ex 1: To find multiple of 2 given integers by using BiFunction.

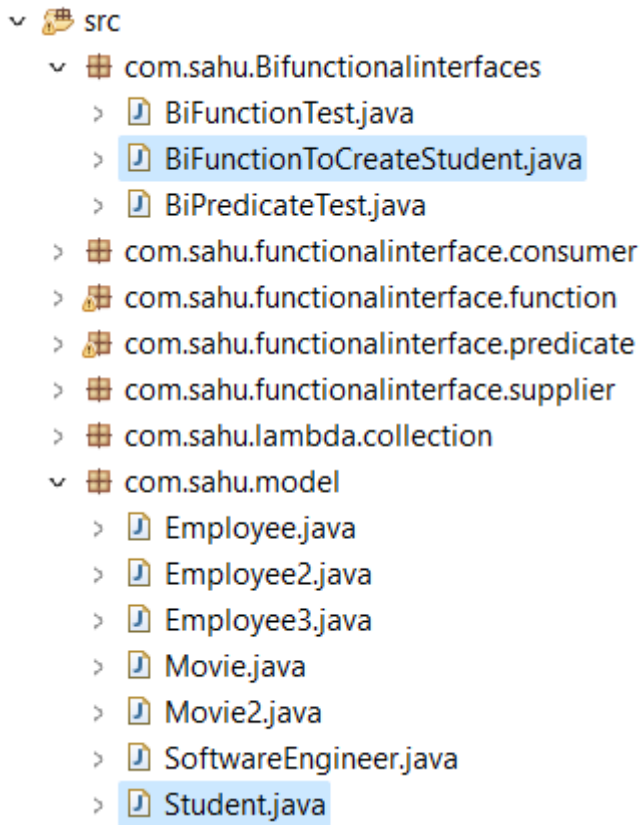


- Add the selected java files, then places the following code with in their respective file.

BiFunctionTest.java

```
package com.sahu.Bifunctionalinterfaces;

import java.util.function.BiFunction;

public class BiFunctionTest {

        public static void main(String[] args) {
                BiFunction<Integer, Integer, Integer> bifunction = (a, b) -> a*b;
                System.out.println(bifunction.apply(10, 20));
                System.out.println(bifunction.apply(100, 200));
        }

}
```

Ex 2: Creation of Student object by taking name and roll number as input with BiFunction.



- Add the selected java files, then places the following code with in their respective file.

Student.java

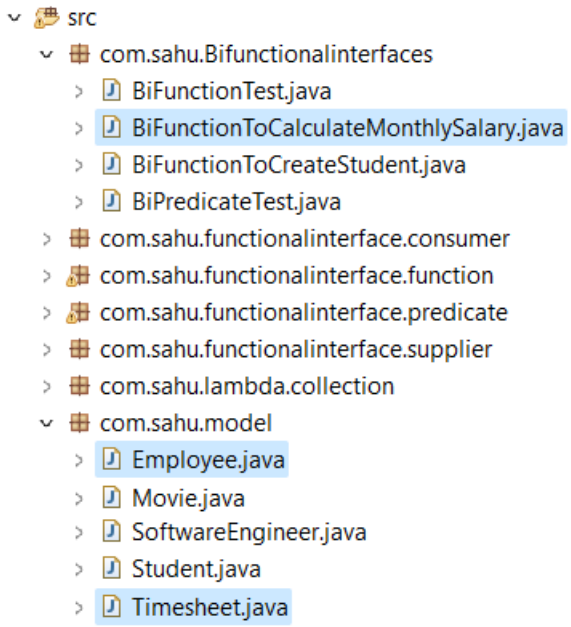
```
package com.sahu.model;

public class Student {
	private String name;
	private Integer rollNo;

	public Student(String name, Integer rollNo) {
		this.name = name;
		this.rollNo = rollNo;
	}

	public String getName() {
		return name;
```

```
	}

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

	public Integer getRollNo() {
		return rollNo;
	}

	public void setRollNo(Integer rollNo) {
		this.rollNo = rollNo;
	}
}
```

BiFunctionToCreateStudent.java

```
package com.sahu.Bifunctionalinterfaces;

import java.util.ArrayList;
import java.util.List;
import java.util.function.BiFunction;

import com.sahu.model.Student;

public class BiFunctionToCreateStudent {

	public static void main(String[] args) {
		List<Student> students = new ArrayList<>();
		BiFunction<String, Integer, Student> createStudent = (name,
rollNo) -> new Student(name, rollNo);
		students.add(createStudent.apply("Durga", 100));
		students.add(createStudent.apply("Ravi", 200));
		students.add(createStudent.apply("Shiva", 300));
		for (Student student : students) {
			System.out.println("Name: " + student.getName());
			System.out.println("RollNo: " + student.getRollNo());
		}
	}

}
```

Ex: To calculate monthly salary with Employee and Timesheet objects as input by using BiFunction.



- Add the selected java files, then places the following code with in their respective file.

Employee.java

```
package com.sahu.model;

public class Employee {
	private Integer eno;
	private String name;
	private Double dailyWage;

	public Employee(Integer eno, String name, Double dailyWage) {
		this.eno = eno;
		this.name = name;
		this.dailyWage = dailyWage;
	}

	public Integer getEno() {
		return eno;
	}

	public void setEno(Integer eno) {
```

```
		this.eno = eno;
	}

	public String getName() {
		return name;
	}

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

	public Double getDailyWage() {
		return dailyWage;
	}

	public void setDailyWage(Double dailyWage) {
		this.dailyWage = dailyWage;
	}
}
```

Timesheet.java

```
package com.sahu.model;

public class Timesheet {
	private Integer eno;
	private Integer dayas;

	public Timesheet(Integer eno, Integer dayas) {
		this.eno = eno;
		this.dayas = dayas;
	}

	public Integer getEno() {
		return eno;
	}

	public void setEno(Integer eno) {
		this.eno = eno;
	}
```

```
	public Integer getDayas() {
		return dayas;
	}

	public void setDayas(Integer dayas) {
		this.dayas = dayas;
	}
}
```

BiFunctionToCalculateMonthlySalary.java

```
package com.sahu.Bifunctionalinterfaces;

import java.util.function.BiFunction;

import com.sahu.model.Employee;
import com.sahu.model.Timesheet;

public class BiFunctionToCalculateMonthlySalary {

	public static void main(String[] args) {
		Employee employee = new Employee(101, "Durga", 1500.0);
		Timesheet timesheet = new Timesheet(101, 25);
		BiFunction<Employee, Timesheet, Double>
calculateMonthlySalary = (emp, times) -> emp.getDailyWage() *
times.getDayas();
		System.out.println(calculateMonthlySalary.apply(employee,
timesheet));
	}

}
```

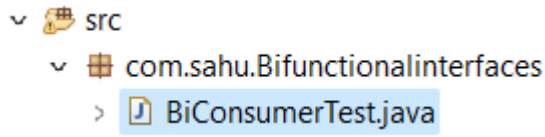
BiConsumer

- Normal Consumer can take only one input argument and perform required operation and won't return any result.
- But sometimes our programming requirement to accept two input values and perform required operation and not required to return any result. Then we should go for BiConsumer.
- BiConsumer is exactly same as Consumer except that it will take two

input arguments. It also have andThen() default method.

```
@FunctionalInterface
public interface BiConsumer<T, U> {
        void accept(T t, U u);
}
```

Ex 1: Program to accept two String values and print result of concatenation by using BiConsumer.

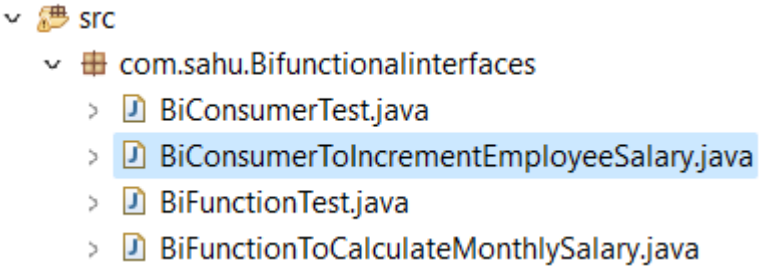


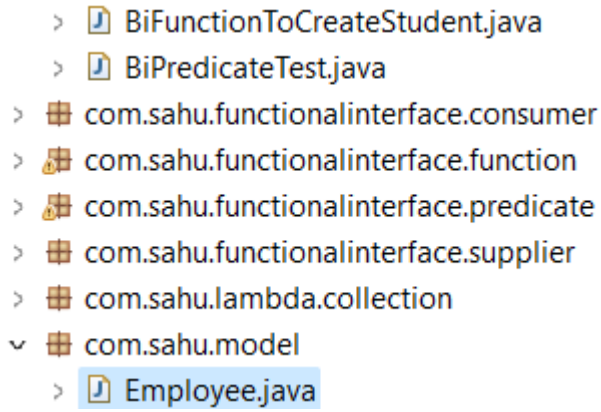
- Add the selected java files, then places the following code with in their respective file.

BiConsumerTest.java

```
package com.sahu.Bifunctionalinterfaces;

import java.util.function.BiConsumer;

public class BiConsumerTest {

    public static void main(String[] args) {
        BiConsumer<String, String> concateString = (str1, str2) ->
System.out.println(str1 + str2);
        concateString.accept("webtech", "solution");
    }

}
```

Ex 2: Program to increment employee salary by using BiFunction, BiConsumer.





- Add the selected java files, then places the following code with in their respective file.

Employee.java

```
package com.sahu.model;
public class Employee {
	private String name;
	private Double salary;

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

	public String getName() {
		return name;
	}

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

	public Double getSalary() {
		return salary;
	}

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

BiConsumerTest.java

```
package com.sahu.Bifunctionalinterfaces;

import java.util.ArrayList;
import java.util.List;
import java.util.function.BiConsumer;
import java.util.function.BiFunction;

import com.sahu.model.Employee;

public class BiConsumerToIncrementEmployeeSalary {

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

		BiFunction<String, Double, Employee> createEmployee =
(name, salary) -> new Employee(name, salary);

		employees.add(createEmployee.apply("Durga", 1000.0));
		employees.add(createEmployee.apply("Sunny", 2000.0));
		employees.add(createEmployee.apply("Bunny", 3000.0));
		employees.add(createEmployee.apply("Chinny", 4000.0));

		BiConsumer<Employee, Double> incrementedSalary =
(employee, incrementSal) -> employee
				.setSalary(employee.getSalary() + incrementSal);

		for (Employee emp : employees) {
			incrementedSalary.accept(emp, 500.0);
		}

		for (Employee emp : employees) {
			System.out.println("Name: " + emp.getName());
			System.out.println("Salary: " + emp.getSalary());
			System.out.println();
		}
	}

}
```

Comparison Table between One argument and Two argument Functional Interfaces

One Argument Functional Interface	Two Argument Functional Interface

interface Predicate<T> {
public boolean test(T t);
default Predicate and(Predicate P) {
}
default Predicate or(Predicate P) {
}
default Predicate negate() {
}
static Predicate isEqual(Object o){
}
}
	interface BiPredicate<T, U> {
	public boolean test(T t, U u);
	default BiPredicate and(
	BiPredicate P) {
	}
	default BiPredicate or(
	BiPredicate P) {
	}
	default BiPredicate negate() {
	}
	}

interface Function<T, R> {
public R apply(T t);
default Function andThen(
Function F) {
}
default Function compose(
Function F) {
}
static Function identify() {
}
}
	interface BiFunction<T, U, R> {
	public R apply(T t, U u);
	default BiFunction andThen(
	Function F) {
	}
	}

interface Consumer<T> {
public void accept(T t);
default Consumer andThen(
Consumer C) {
}
}
	interface BiConsumer<T, U> {
	public void accept(T t, U u);
	default BiConsumer andThen(
	BiConsumer C) {
	}
	}

Primitive type Functional Interfaces

1. Autoboxing

- Automatic conversion from primitive type to Object type by compiler is called autoboxing.
- Introduced in Java 1.5 version.

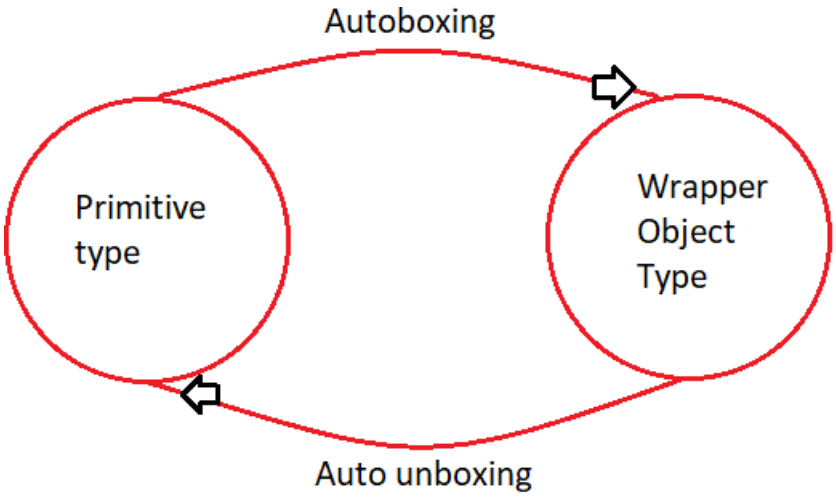
Ex: Integer num = 10; => Integer num = Integer.valueOf(10); //internally

From 1.5 it is valid but before 1.5 i.e. until 1.4 version we you use the above code then we will get compile time error.
CE: incompatible types: int cannot be converted to Integer.

2. Auto unboxing

- Automatic conversion from object type to primitive type by compiler is called auto unboxing.
- Introduced in Java 1.5 version.

Ex:
Integer num = new Integer(10);
Int x = num; => int x = num.intValue(); //internally



3. Generics Type

- In the case of generics, the type parameter is always object type and no chance of passing primitive type.

Ex:
ArrayList<Integer> al = new ArrayList<Integer>();//valid
ArrayList<int> al = new ArrayList<int>(); //invalid

Need of Primitive Functional Interfaces:

- In the case of normal Functional interfaces (like Predicate, Function, etc.) input and return types are always Object types. If we pass primitive values then these primitive values will be converted to Object type (autoboxing), and again according to need again it need to convert from Object type to primitive type (auto unboxing) which creates unnecessary performance problems.

Ex:

```
public class Test {
        public static void main(String[] args) 5) {
                int[] x = {0, 5, 10, 15, 20, 25};
                Predicate<Integer> p = i -> i%2==0;
                for (int x1 : x) {
                        if (p.test(x1)) 11) {
                                System.out.println(x1);
                        }
                }
        }
}
```

In the above examples, 6 times autoboxing and auto unboxing is required to perform unnecessary performance is going to be down. So normal functional interface is not best suitable for the primitive types.

Note: To overcome this problem primitive functional interfaces introduced, which can always take primitive types as input and return primitive types. Hence autoboxing and auto unboxing won't be required, which improves performance.

Primitive type Functional interfaces for Predicate

- The following are various primitive Functional interfaces for Predicate.

1. **IntPredicate:** always accepts input value of int type. It also having and(), or(), and negate() default methods but no static methods.

```
@FunctionalInterface
public interface IntPredicate {
        boolean test(int value);
}
```

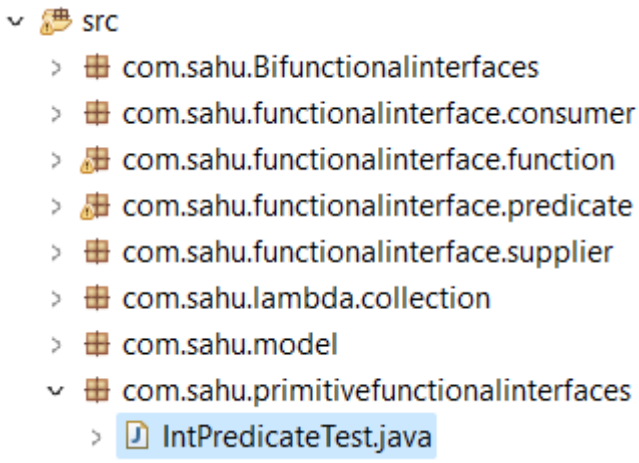
2. **LongPredicate:** always accepts input value of long type. It also having and(), or(), and negate() default methods but no static methods.

```
@FunctionalInterface
public interface LongPredicate {
        boolean test(long value);
}
```

3. **DoublePredicate:** always accepts input value of double type. It also having and(), or(), and negate() default methods but no static methods.

```
@FunctionalInterface
public interface DoublePredicate {
        boolean test(double value);
}
```

Ex: Use of IntPredicate



- Add the selected java files, then places the following code with in their respective file.

IntPredicateTest.java

```
package com.sahu.primitivefunctionalinterfaces;

import java.util.function.IntPredicate;

public class IntPredicateTest {
        public static void main(String[] args) {
                int[] x = { 0, 5, 10, 15, 20, 25 };
                IntPredicate intPredicate = num -> num % 2 == 0;
                for (int i : x) {
                        if (intPredicate.test(i)) {
                                System.out.println(i);
                        }
                }
        }
}
```

Note: In the above example, autoboxing and auto unboxing won't be performed internally. Hence performance wise also improvements are there.

Primitive type Functional interfaces for Function

- The following are various primitive type Functional interfaces for Function.

1. **IntFunction<R>:** can take int type as input and return any type.

```
@FunctionalInterface
public interface IntFunction<R> {
        R apply(int value);
}
```

2. **LongFunction<R>:** can take long type as input and return any type.

```
@FunctionalInterface
public interface LongFunction<R> {
        R apply(long value);
}
```

3. **DoubleFunction<R>:** can take double type as input and return any type.

```
@FunctionalInterface
public interface DoubleFunction<R> {
        R apply(double value);
}
```

4. **ToIntFunction<T>:** It can take any type as input but always returns int type.

```
@FunctionalInterface
public interface ToIntFunction<T> {
        int applyAsInt(T value);
}
```

5. **ToLongFunction<T>:** It can take any type as input but always returns long type.

```
@FunctionalInterface
```

```
public interface ToLongFunction<T> {
        long applyAsLong(T value);
}
```

6. **ToDoubleFunction<T>:** It can take any type as input but always returns double type.

```
@FunctionalInterface
public interface ToDoubleFunction<T> {
        double applyAsDouble(T value);
}
```

7. **IntToLongFunction:** It can take int type as input and returns long type.

```
@FunctionalInterface
public interface IntToLongFunction {
        long applyAsLong(int value);
}
```

8. **IntToDoubleFunction:** It can take int type as input and returns long type.

```
@FunctionalInterface
public interface IntToDoubleFunction {
        double applyAsDouble(int value);
}
```

9. **LongToIntFunction:** It can take long type as input and returns int type.

```
@FunctionalInterface
public interface LongToIntFunction {
        int applyAsInt(long value);
}
```

10. **LongToDoubleFunction:** It can take long type as input and returns double type.

```
@FunctionalInterface
public interface LongToDoubleFunction {
        double applyAsDouble(long value);
}
```

11. **DoubleToIntFunction:** It can take double type as input and returns int type.

```
@FunctionalInterface
public interface DoubleToIntFunction {
        int applyAsDouble(double value);
}
```

12. **DoubleToLongFunction:** It can take double type as input and returns long type.

```
@FunctionalInterface
public interface DoubleToLongFunction {
        long applyAsLong(double value);
}
```

13. **ToIntBiFunction:** taking any types two inputs and return type must be int.

```
@FunctionalInterface
public interface ToIntBiFunction<T, U> {
        int applyAsInt(T t, U u);
}
```

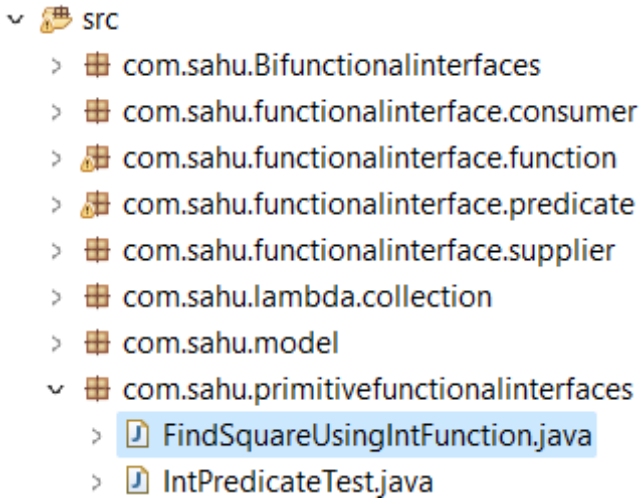
14. **ToLongBiFunction:** taking any types two inputs and return type must be long.

```
@FunctionalInterface
public interface ToLongBiFunction<T, U> {
        long applyAsLong(T t, U u);
}
```

15. **ToDoubleBiFunction:** taking any types two inputs and return type must be double.

```
@FunctionalInterface
public interface ToDoubleBiFunction<T, U> {
        double applyAsDouble(T t, U u);
}
```

Ex 1: Program to find square of given integer by using Function.

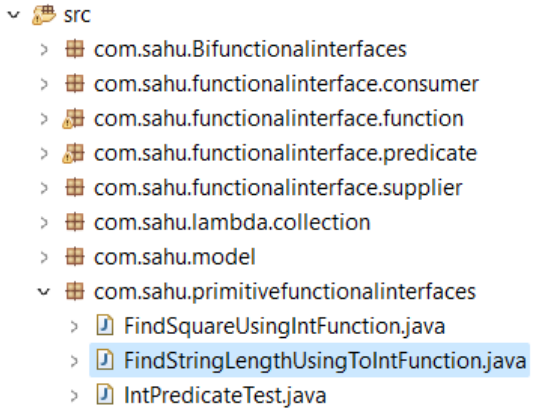


- Add the selected java files, then places the following code with in their respective file.

FindSquareUsingIntFunction.java

```
package com.sahu.primitivefunctionalinterfaces;

import java.util.function.IntFunction;

public class FindSquareUsingIntFunction {
	public static void main(String[] args) {
		IntFunction<Integer> findSquare = i -> i * i;
		System.out.println(findSquare.apply(5));
	}
}
```

Ex 2: Program to find length of the given String by using Function.

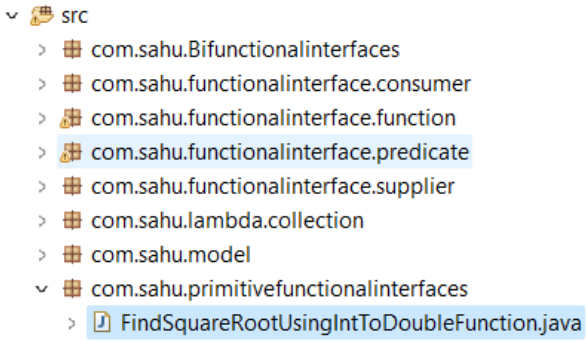


- Add the selected java files, then places the following code with in their respective file.

FindStringLengthUsingToIntFunction.java

```
package com.sahu.primitivefunctionalinterfaces;

import java.util.function.ToIntFunction;

public class FindStringLengthUsingToIntFunction {
	public static void main(String[] args) {
		ToIntFunction<String> getLength = s -> s.length();
		System.out.println(getLength.applyAsInt("durga"));
	}
}
```

Ex 3: Program to find square root of given integer by using Function.



- Add the selected java files, then places the following code with in their respective file.

FindSquareRootUsingIntToDoubleFunction.java

```
package com.sahu.primitivefunctionalinterfaces;

import java.util.function.IntToDoubleFunction;

public class FindSquareRootUsingIntToDoubleFunction {
	public static void main(String[] args) {
		IntToDoubleFunction findSquareRoot = i -> Math.sqrt(i);
		System.out.println(findSquareRoot.applyAsDouble(9));
	}
}
```

Primitive type Functional interfaces for Consumer

- The following 6 primitive versions available for Consumer,

1. **IntConsumer:** If input is int type. And having andThen() default method.

```
@FunctionalInterface
public interface IntConsumer {
        void accept(int value);
}
```

2. **LongConsumer:** If input is long type. And having andThen() default method.

```
@FunctionalInterface
public interface LongConsumer {
        void accept(long value);
}
```

3. **DoubleConsumer:** If input is double type, having same default method.

```
@FunctionalInterface
public interface DoubleConsumer {
        void accept(double value);
}
```

4. **ObjIntConsumer<T>:** If one input is object/ any type and another one is int type. Here no default method.

```
@FunctionalInterface
public interface ObjIntConsumer<T> {
        void accept(T t, int value);
}
```

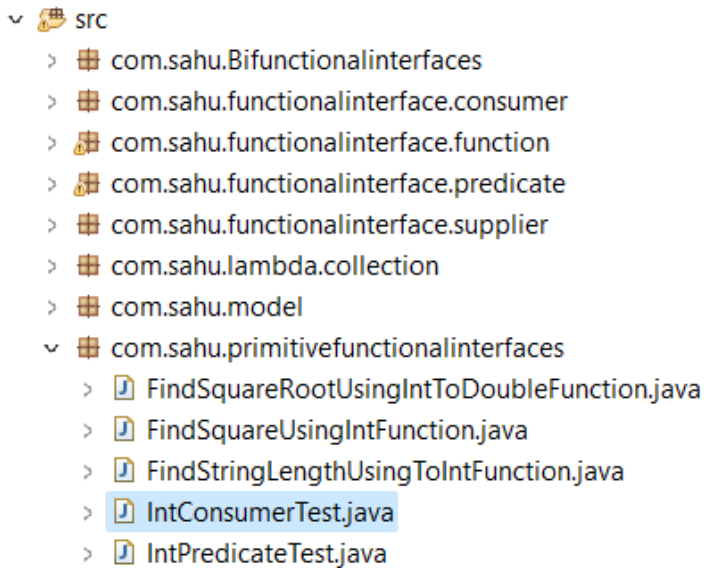
5. **ObjLongConsumer<T>:** If one input is object/ any type and another one is long type. Here no default method.

```
@FunctionalInterface
public interface ObjLongConsumer<T> {
        void accept(T t, long value);
}
```

6. ObjDoubleConsumer<T>: If one input is object/ any type and another one is double type. Here no default method.

```
@FunctionalInterface
public interface ObjLongConsumer<T> {
        void accept(T t, long value);
}
```

Ex 1: Program for IntConsumer

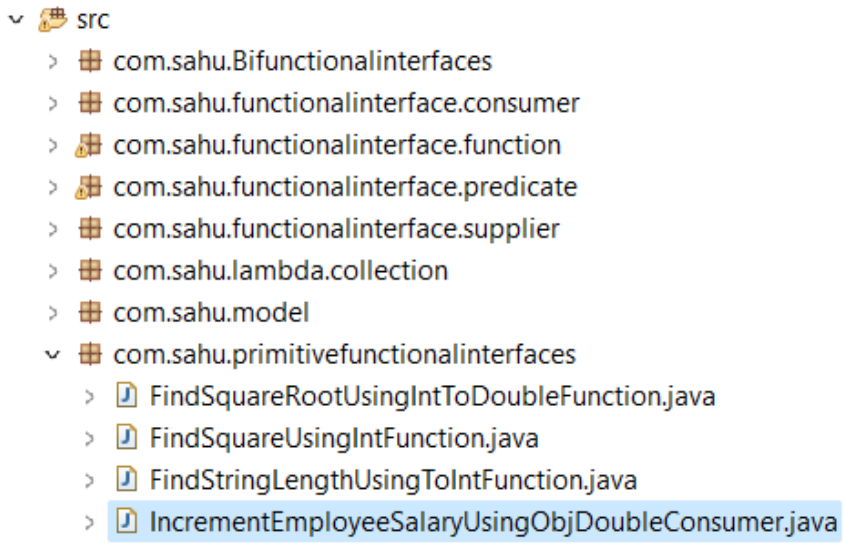


- Add the selected java files, then places the following code with in their respective file.

IntConsumerTest.java

```
package com.sahu.primitivefunctionalinterfaces;

import java.util.function.IntConsumer;

public class IntConsumerTest {

        public static void main(String[] args) {
                IntConsumer displayConsumer = num ->
System.out.println("The square is - "+(num * num));
                displayConsumer.accept(5);
        }

}
```

Ex 2: Program to increment employee Salary by using ObjDoubleConsumer



- Add the selected java files, then places the following code with in their respective file.
- Take old Employee class.

IncrementEmployeeSalaryUsingObjDoubleConsumer.java

```
package com.sahu.primitivefunctionalinterfaces;

import java.util.ArrayList;
import java.util.List;
import java.util.function.ObjDoubleConsumer;

import com.sahu.model.Employee;

public class IncrementEmployeeSalaryUsingObjDoubleConsumer {

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

		ObjDoubleConsumer<Employee> incrementSalary = (employee,
increment) -> employee.setSalary(employee.getSalary() + increment);

		for (Employee emp : employees) {
			incrementSalary.accept(emp, 500.0);
		}
```

```
            for (Employee emp : employees) {
                System.out.println("Employee Name: "+emp.getName());
                System.out.println("Employee Salary: "+emp.getSalary());
                System.out.println();
            }
        }

        public static void populateEmployee(List<Employee> employees) {
            employees.add(new Employee("Durga", 1000.0));
            employees.add(new Employee("Sunny", 2000.0));
            employees.add(new Employee("Bunny", 3000.0));
            employees.add(new Employee("Chinny", 4000.0));
        }

}
```

Primitive type Functional interfaces for Supplier

- The following 4 primitive versions available for Supplier

1. **IntSupplier:** If return type is int type.

```
@FunctionalInterface
public interface IntSupplier {
        int getAsInt();
}
```

2. **LongSupplier:** If return type is long type.

```
@FunctionalInterface
public interface LongSupplier {
        long getAsLong();
}
```

3. **DoubleSupplier:** If return type is double type.

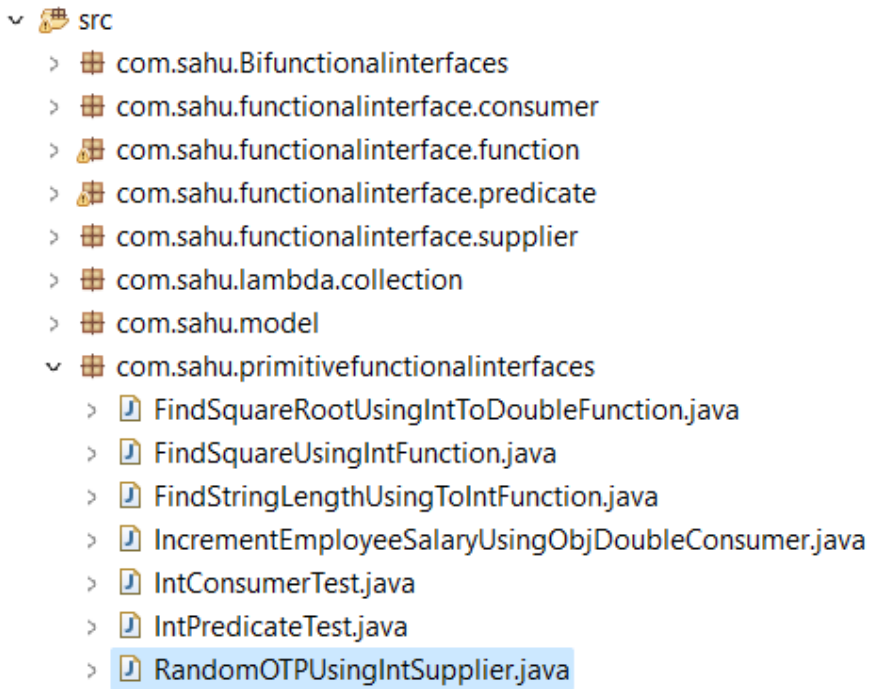
```
@FunctionalInterface
public interface DoubleSupplier {
```

```
        double getAsDouble();
    }
```

4. **BooleanSupplier:** If return type is boolean type.

```
    @FunctionalInterface
    public interface BooleanSupplier {
        boolean getAsBoolean();
    }
```

Ex: Program to generate 6-digit random OTP by using IntSupplier.



- Add the selected java files, then places the following code with in their respective file.

RandomOTPUsingIntSupplier.java

```
package com.sahu.primitivefunctionalinterfaces;

import java.util.function.IntSupplier;

public class RandomOTPUsingIntSupplier {

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

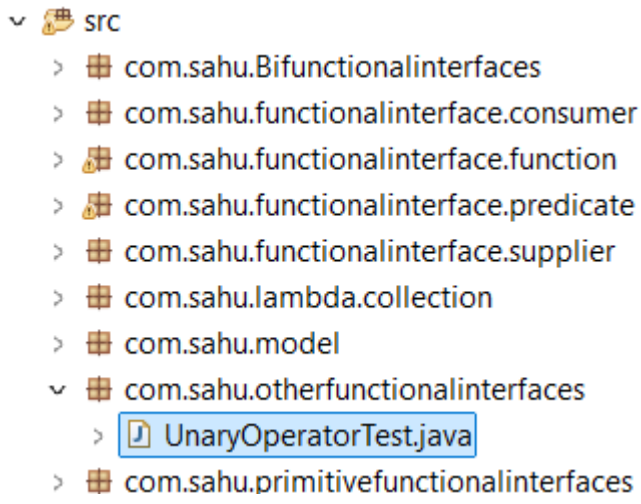
```
        IntSupplier randomNum = () -> (int) (Math.random() * 10);

        StringBuffer otp = new StringBuffer();
        for (int i = 0; i < 6; i++) {
            otp.append(randomNum.getAsInt());
        }
        System.out.println("OTP is - "+otp.toString());
    }

}
```

UnaryOperator<T>

- If input and output are same type then we should go for UnaryOperator.
- It is child of Function<T, T>.
- This functional interface doesn't have direct abstract method so the abstract method is apply() only.
- And also no default method only one static method is there identity().

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {

}
```

Ex: Using of UnaryOperator



- Add the selected java files, then places the following code with in their respective file.

UnaryOperatorTest.java

```
package com.sahu.otherfunctionalinterfaces;

import java.util.function.UnaryOperator;

public class UnaryOperatorTest {

	public static void main(String[] args) {
		UnaryOperator<Integer> sumCal = i -> i * i;
		System.out.println(sumCal.apply(10));
	}

}
```

The primitive versions for UnaryOperator:

1. **IntUnaryOperator:** If input and output both are int type. Here default methods are compose(), andThen() and static method identity().

```
@FunctionalInterface
public interface IntUnaryOperator {
		int applyAsInt(int operand);
}
```

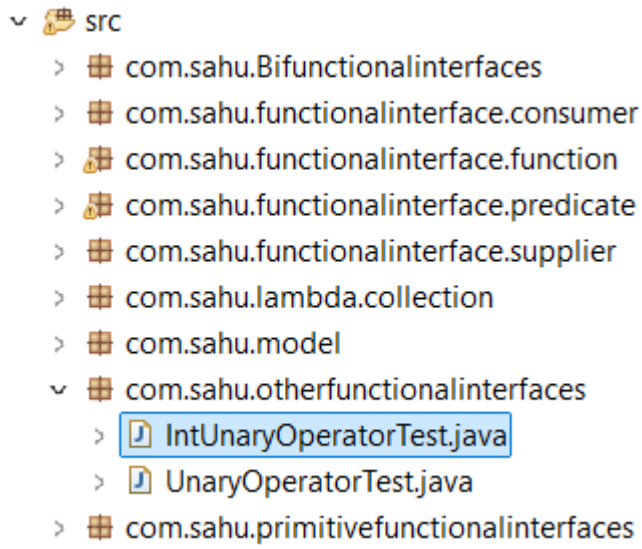
2. **LongUnaryOperator:** If input and output both are long type. Here default methods are compose(), andThen() and static method identity().

```
@FunctionalInterface
public interface LongUnaryOperator {
		long applyAsLong(long operand);
}
```

3. **DoubleUnaryOperator:** If input and output both are double type. Here default methods are compose(), andThen() and static method identity().

```
@FunctionalInterface
public interface DoubleUnaryOperator {
		double applyAsDouble(double operand);
}
```

Ex: Using of IntUnaryOperator



- Add the selected java files, then places the following code with in their respective file.

IntUnaryOperatorTest.java

```
package com.sahu.otherfunctionalinterfaces;

import java.util.function.IntUnaryOperator;

public class IntUnaryOperatorTest {

	public static void main(String[] args) {
		IntUnaryOperator sumCal = i -> i * i;
		System.out.println(sumCal.applyAsInt(10));
	}

}
```

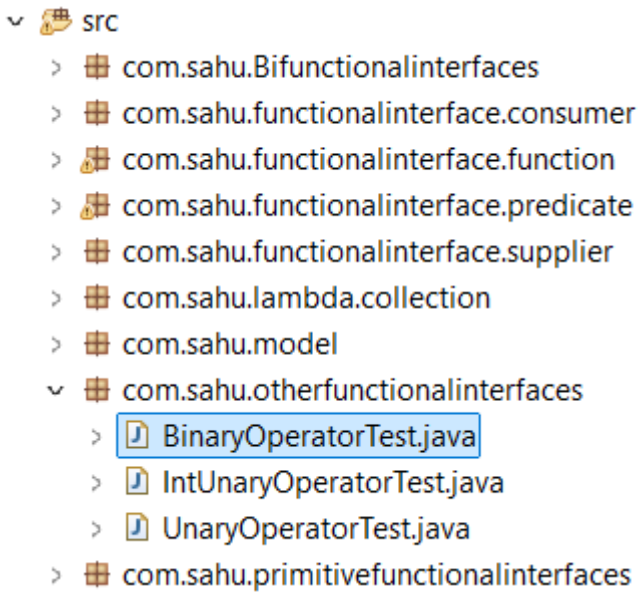
BinaryOperator<T>

- It is the child of BiFunction<T, T, T>.
- If two input and return type is the same i.e. nothing but all the three parameters are same type to handle such type of requirement, we should go for BinaryOperator.
- This functional interface also doesn't have direct abstract method so the abstract method is apply() only.

- And no default methods are there, but two static method are there minBy(), maxBy().

```
@FunctionalInterface
public interface BinaryOperator<T> extends Function<T, T, T> {

}
```

Ex: Using of BinaryOperator



- Add the selected java files, then places the following code with in their respective file.

BinaryOperatorTest.java

```
package com.sahu.otherfunctionalinterfaces;

import java.util.function.BinaryOperator;

public class BinaryOperatorTest {

	public static void main(String[] args) {
		BinaryOperator<String> concate = (str1, str2) -> str1 + str2;
		System.out.println(concate.apply("Webtect", "solution"));
	}

}
```

The primitive versions for BinaryOperator:

1. **IntBinaryOperator:** If two input and output also int then we should go for IntBinaryOperator.

```
@FunctionalInterface
public interface IntBinaryOperator {
        int applyAsInt(int left, int right);
}
```

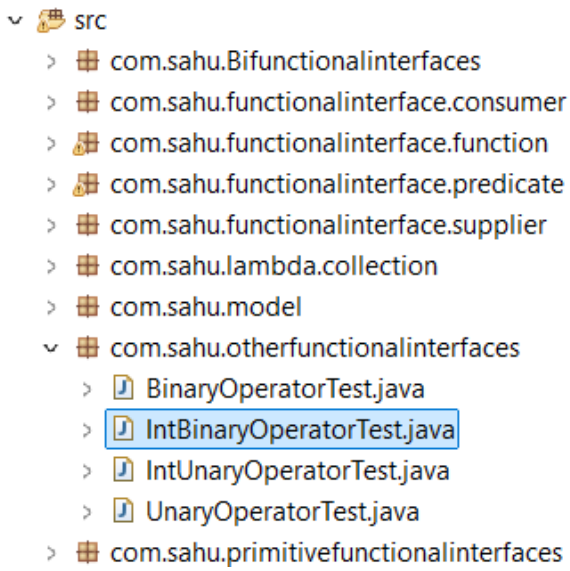
2. **LongBinaryOperator:** If two input and output also long then we should go for LongBinaryOperator.

```
@FunctionalInterface
public interface LongBinaryOperator {
        long applyAsLong(long left, long right);
}
```

3. **DoubleBinaryOperator:** If two input and output also double then we should go for DoubleBinaryOperator.

```
@FunctionalInterface
public interface DoubleBinaryOperator {
        double applyAsDouble(double left, double right);
}
```

Ex: Using of IntBinaryOperator



- Add the selected java files, then places the following code with in their respective file.

IntBinaryOperatorTest.java

```
package com.sahu.otherfunctionalinterfaces;

import java.util.function.IntBinaryOperator;

public class IntBinaryOperatorTest {

	public static void main(String[] args) {
		IntBinaryOperator sumCount = (num1, num2) -> num1 + num2;
		System.out.println(sumCount.applyAsInt(10, 20));
	}

}
```

Method references by using Double colon operator (: :)

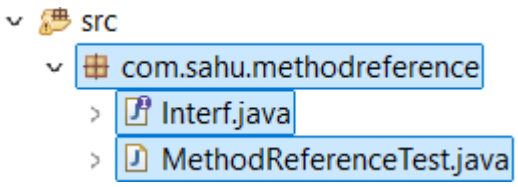
- Functional Interface method can be mapped to our specified method by using double colon (: :) operator. This is called method reference.
- Our specified method can be either static method or instance method.
- Functional Interface method and our specified method should have same argument types, except this the remaining things like return type, method name, modifier set are not required to match.

  Syntax:
  - If our specified method is static method
    Classname::methodName
    Ex: Test::m2

  - If the method is instance method
    Object ref::methodName
    Ex: Test test = new Test();
    test::m2

- Functional Interface can refer lambda expression and Functional Interface can also refer method reference. Hence lambda expression can be replaced with method reference. Hence method reference is alternative syntax to lambda expression.

Ex: Functional interface implementation using Method Reference



- Add the selected java files, then places the following code with in their respective file.

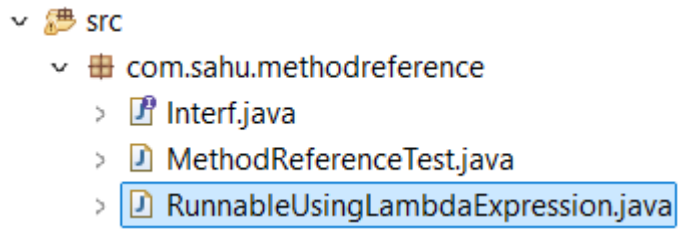
Interf.java

```
package com.sahu.methodreference;

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

MethodReferenceTest.java

```
package com.sahu.methodreference;

public class MethodReferenceTest {

	public static void main(String[] args) {
		// using lambda expression
		Interf interf1 = () -> System.out.println("Implmentation by
Lambda expression");
		interf1.m1();

		Interf interf2 = MethodReferenceTest::m2;
		interf2.m1();
	}

	public static void m2() {
		System.out.println("Implementation by method Reference");
		// other codes
	}

}
```

Ex: Runnable implementation using Lambda Expression

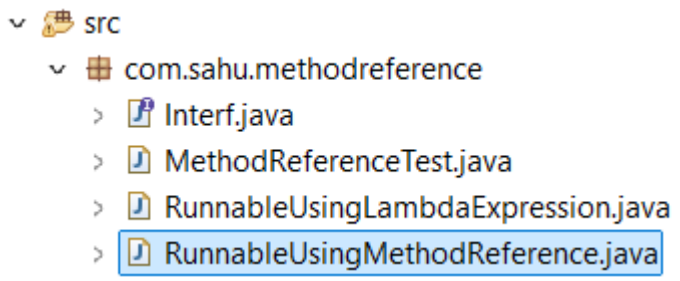


- Add the selected java files, then places the following code with in their respective file.

RunnableUsingLambdaExpression.java

```
package com.sahu.methodreference;

public class RunnableUsingLambdaExpression {

	public static void main(String[] args) {
		Runnable runnable = () -> {
			for (int i = 0; i < 10; i++) {
				System.out.println("Child Thread");
			}
		};
		Thread thread = new Thread(runnable);
		thread.start();

		for (int i = 0; i < 10; i++) {
			System.out.println("Main Thread");
		}
	}

}
```

Ex: Runnable implementation using Method reference



- Add the selected java files, then places the following code with in their respective file.

RunnableUsingMethodReference.java

```
package com.sahu.methodreference;

public class RunnableUsingMethodReference {

	public static void main(String[] args) {
		RunnableUsingMethodReference methodReference = new
RunnableUsingMethodReference();
		Runnable runnable = methodReference::m1;
		Thread thread = new Thread(runnable);
		thread.start();

		for (int i = 0; i < 10; i++) {
			System.out.println("Main Thread");
		}
	}

	public void m1() {
		for (int i = 0; i < 10; i++) {
			System.out.println("Child Thread");
		}
	}

}
```

Note: The main advantage of method reference is we can use already existing code to implement functional interfaces (code reusability).

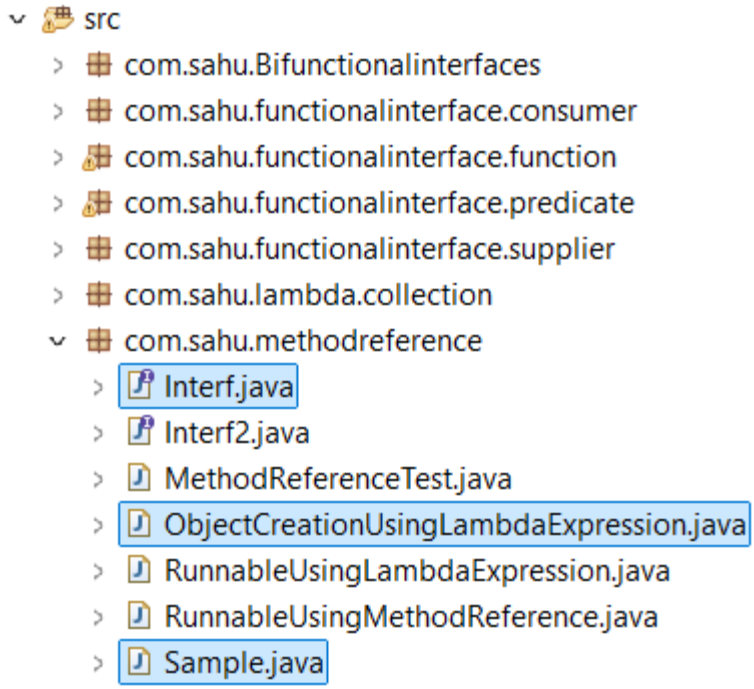
Constructor references by using Double colon operator (: :)

- We can use double colon (: :) operator to refer constructors also.

Syntax: classname::new
Ex: Test::new

Note: In method and constructor references compulsory the argument types must be matched.

Ex: Object creating using Lambda expression



- Add the selected java files, then places the following code with in their respective file.

Sample.java

```
package com.sahu.methodreference;

public class Sample {

	public Sample() {
		System.out.println("Sample constructor execution object creation");
	}

}
```

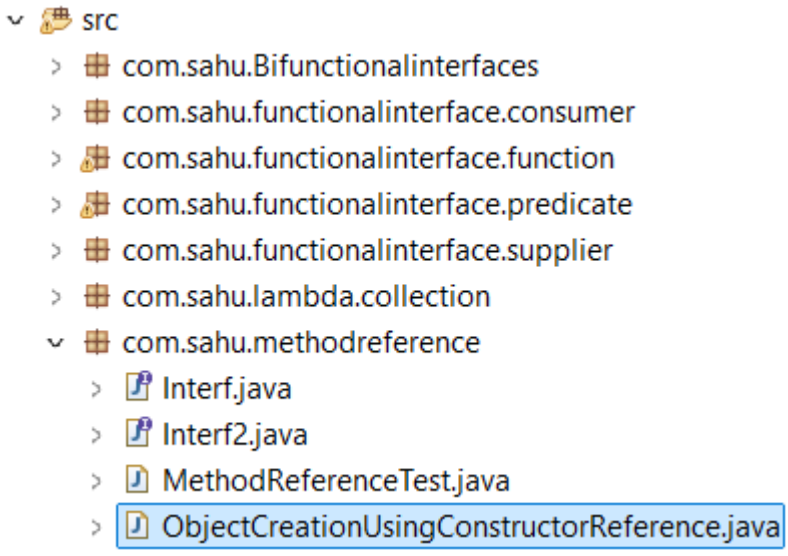
Interf.java

```
package com.sahu.methodreference;
@FunctionalInterface
public interface Interf2 {
	public Sample get();
}
```

ObjectCreationUsingLambdaExpression.java

```
package com.sahu.methodreference;

public class ObjectCreationUsingLambdaExpression {

	public static void main(String[] args) {
		Interf interf = () -> {
			Sample sample = new Sample();
			return sample;
		};
		interf.get();
	}

}
```

Ex: Object creating using Lambda expression



- Add the selected java files, then places the following code with in their respective file.
- Take the previous Interf.java and Sample.java files.

ObjectCreationUsingConstructorReference.java

```
package com.sahu.methodreference;

public class ObjectCreationUsingConstructorReference {

```

```
        public static void main(String[] args) {
                Interf interf = Sample::new;
                interf.get();
        }

}
```

Streams

- To process objects of the collection, in 1.8 version Streams concept introduced.

Q. What is the differences between java.util.stream and java.io streams?

Ans.

- If we want to write some text data or binary data to the file and we wan to read some text data or binary data from file. So if we want to represent data it may be character or binary data with respective to the file operations then we should go for java.io streams concepts.
- But java.util.stream or Java 1.8 Streams no way related to files, these Streams is applicable for collection objects.
- If we want to perform certain operation related to the collection or processing objects from the collection then we should go for the Steams.

Q. What is the difference between Collection and Stream?

Ans.

- If we want to represent a group of individual objects as a single entity then we should go for collection.
- If we want to process a group of objects from the collection then we should go for streams.
- We can create a stream object to the collection by using stream() method of Collection interface.
- stream() method is a default method added to the Collection in 1.8 version.

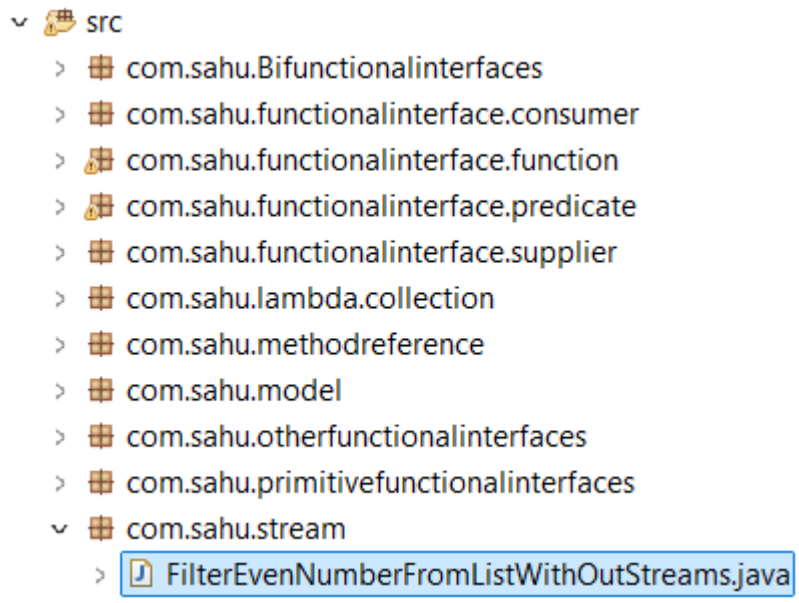
```
default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
}
```

Ex: Stream s = c.stream();
c is any collection type object.

Q. What are the benefits of Streams concept?

Ans.

Ex: Filter out the even numbers from the list without Streams.



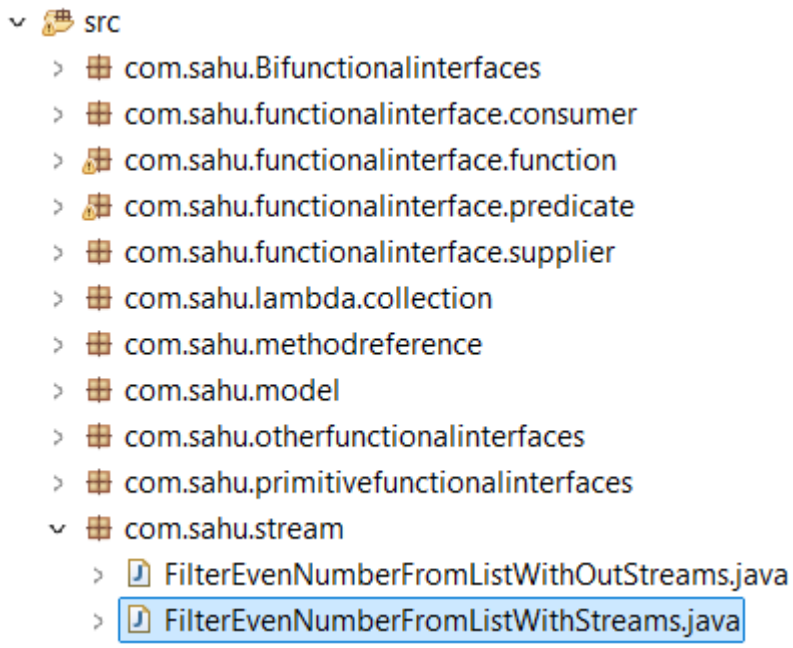
- Add the selected java files, then places the following code with in their respective file.

FilterEvenNumberFromListWithOutStreams.java

```
package com.sahu.stream;

import java.util.ArrayList;
import java.util.List;

public class FilterEvenNumberFromListWithOutStreams {

	public static void main(String[] args) {
		List<Integer> numList = new ArrayList<>();
		numList.add(0);
		numList.add(10);
		numList.add(20);
		numList.add(5);
		numList.add(15);
		numList.add(25);
		System.out.println(numList);

		List<Integer> evenList = new ArrayList<>();
```

```
                for (Integer num : numList) {
                        if (num % 2 == 0) {
                                evenList.add(num);
                        }
                }
                System.out.println(evenList);
        }

}
```

Ex: Filter out the even numbers from the list with Streams.



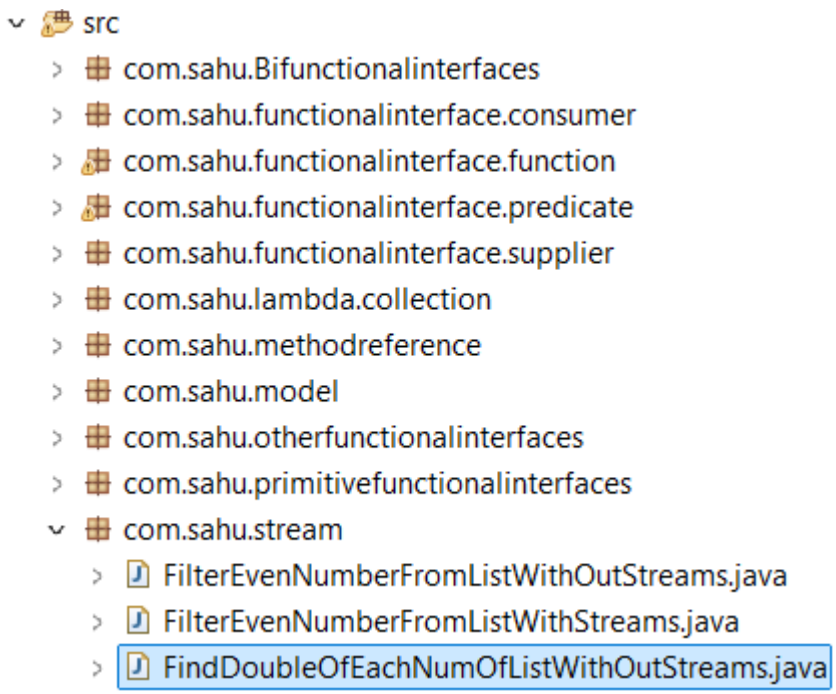
- Add the selected java files, then places the following code with in their respective file.

FilterEvenNumberFromListWithStreams.java

```
package com.sahu.stream;

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

public class FilterEvenNumberFromListWithStreams {

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

```
                List<Integer> numList = new ArrayList<>();
                numList.add(0);
                numList.add(10);
                numList.add(20);
                numList.add(5);
                numList.add(15);
                numList.add(25);
                System.out.println(numList);

                List<Integer> evenList = numList.stream().filter(num -> num % 2
== 0).collect(Collectors.toList());
                System.out.println(evenList);
        }

}
```

Ex: Find the double of each number from the list without Streams.



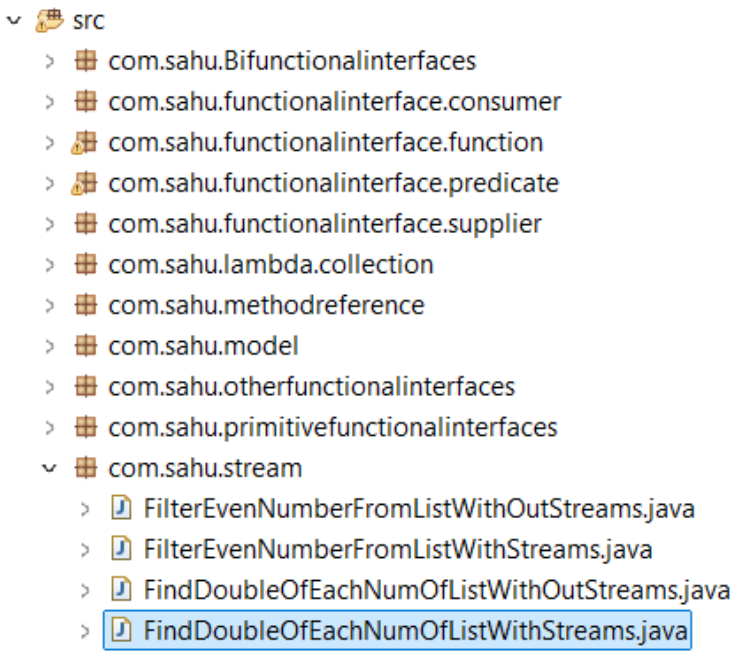
- Add the selected java files, then places the following code with in their respective file.

FindDoubleOfEachNumOfListWithOutStreams.java

```
package com.sahu.stream;
```

```java
import java.util.ArrayList;
import java.util.List;

public class FindDoubleOfEachNumOfListWithOutStreams {

	public static void main(String[] args) {
		List<Integer> numList = new ArrayList<>();
		numList.add(0);
		numList.add(10);
		numList.add(20);
		numList.add(5);
		numList.add(15);
		numList.add(25);
		System.out.println(numList);

		List<Integer> evenList = new ArrayList<>();
		for (Integer num : numList) {
			evenList.add(num * 2);
		}
		System.out.println(evenList);
	}

}
```

Ex: Find the double of each number from the list with Streams.



- Add the selected java files, then places the following code with in their respective file.

FindDoubleOfEachNumOfListWithStreams.java

```
package com.sahu.stream;

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

public class FindDoubleOfEachNumOfListWithStreams {

	public static void main(String[] args) {
		List<Integer> numList = new ArrayList<>();
		numList.add(0);
		numList.add(10);
		numList.add(20);
		numList.add(5);
		numList.add(15);
		numList.add(25);
		System.out.println(numList);

		List<Integer> evenList = numList.stream().map(num -> num *
2).collect(Collectors.toList());
		System.out.println(evenList);
	}

}
```

- Stream is an interface present in java.util.stream package. Once we got the stream object, by using that we can process objects of that collection.
- We can process the objects in the following 2 phases
  - Configuration
  - Processing

Configuration:

- We can configure either by using filter mechanism or by using map mechanism.

Filtering:

- If we want to filter elements from the collection based on some boolean condition then we should go for filtering.
- We can configure filter by filter() method of Stream interface.

```
Stream<T> filter(Predicate<? super T> predicate);
```

it can be a boolean valued function or lambda expression.

Ex:

```
Stream s = c.stream();
Stream s1 = s.filter(i -> i%2 == 0);
```

Mapping:

- If we want to create a separate new object, for every object present in the collection based on son function then we should go for mapping.
- We can implement mapping by using map() method of Stream interface.

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

Ex:

```
Stream s = c.stream();
Stream s1 = s.map(i-> i * 2);
```

Processing:

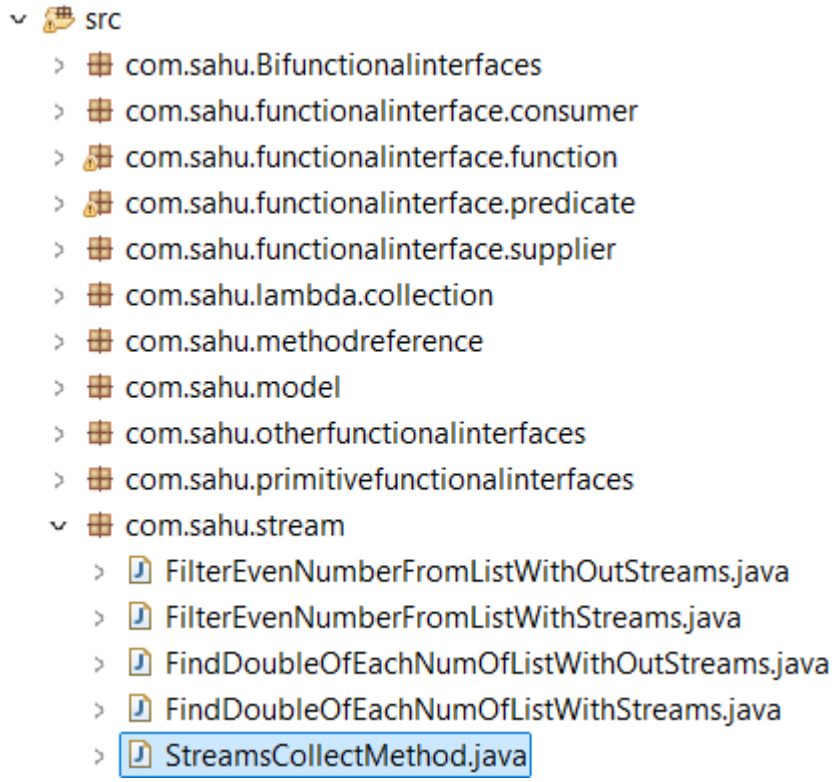
- Once we performed configuration, we can process objects by using several methods.
  - Processing by using collect() method
  - Processing by using count()method
  - Processing by using sorted()method
  - Processing by using min() and max() methods
  - Processing by using forEach() method
  - Processing by using toArray() method
  - Processing by using Stream.of() method

Processing by using collect() method:

- This method collects the elements from the stream and adding to the specified collection.

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

Ex:



- Add the selected java files, then places the following code with in their respective file.

StreamsCollectMethod.java

```
package com.sahu.stream;

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

public class StreamsCollectMethod {

	public static void main(String[] args) {
		List<String> nameList = new ArrayList<>();
		nameList.add("Pavan");
		nameList.add("RaviTeja");
		nameList.add("Chiranjeevi");
		nameList.add("Venkatesh");
		nameList.add("Nagarjuna");
		System.out.println(nameList);
```

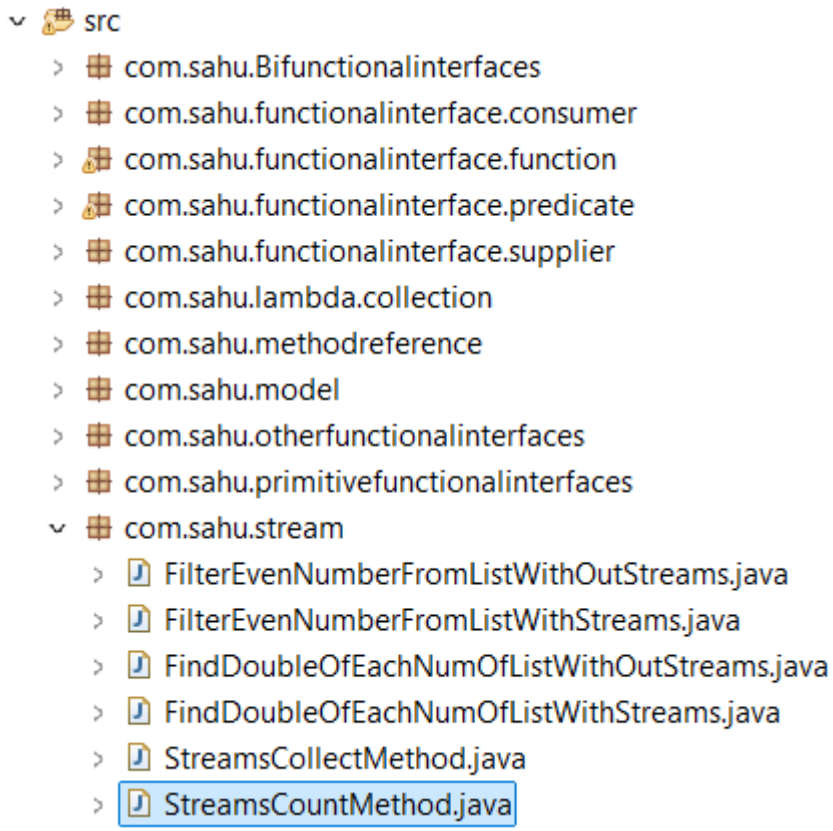
```
                List<String> filteredNameList = nameList.stream().filter(name ->
name.length() >= 9)
                                .collect(Collectors.toList());
                System.out.println(filteredNameList);

                List<String> upperCaseNameList =
nameList.stream().map(name ->
name.toUpperCase()).collect(Collectors.toList());
                System.out.println(upperCaseNameList);
        }

}
```

Processing by using count() method:

- This method returns number of elements present in the stream.

```
long count();
```

Ex:



- Add the selected java files, then places the following code with in their respective file.

StreamsCountMethod.java

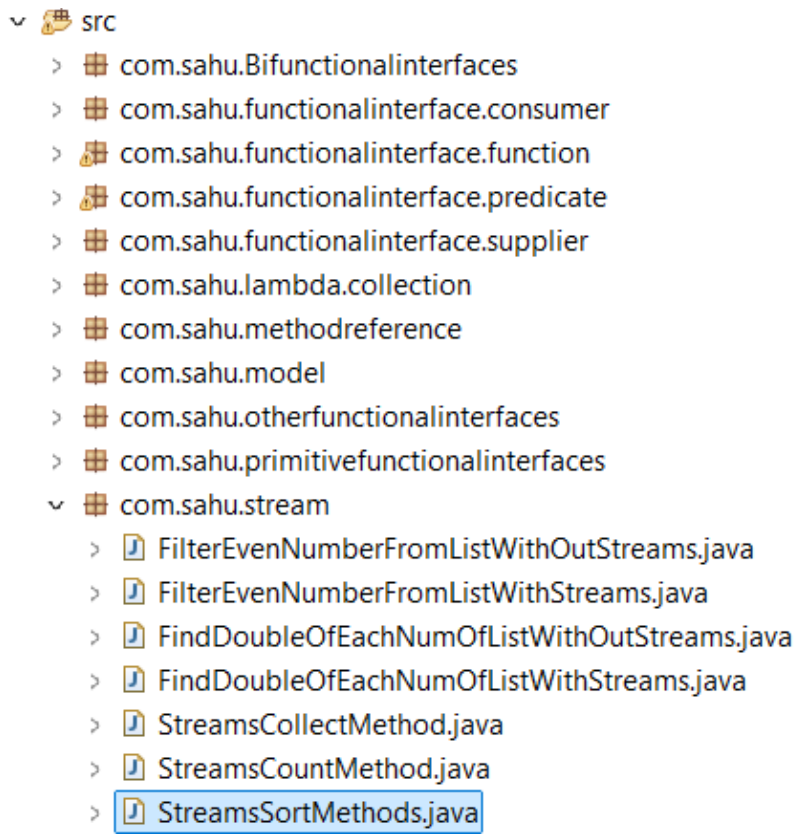
```
package com.sahu.stream;

import java.util.ArrayList;
import java.util.List;

public class StreamsCountMethod {

	public static void main(String[] args) {
		List<String> nameList = new ArrayList<>();
		nameList.add("Pavan");
		nameList.add("RaviTeja");
		nameList.add("Chiranjeevi");
		nameList.add("Venkatesh");
		nameList.add("Nagarjuna");
		System.out.println(nameList);
		Long count = nameList.stream().filter(name -> name.length() >= 9).count();
		System.out.println("The number of Strings whose length >= 9: " + count);
	}

}
```

Processing by using sorted()method:

- We can use sorted() method of Stream interface to sort element inside Stream.
- We can sort either based on default natural sorting order or based on our own customized sorting order specified by Comparator Object.
- sorted() for default natural sorting order
- sorted(Comparator c) for customized sorting order.

```
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);
```

Ex:



- Add the selected java files, then places the following code with in their respective file.

StreamsSortMethods.java

```
package com.sahu.stream;

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

public class StreamsSortMethods {

	public static void main(String[] args) {
		List<Integer> numList = new ArrayList<>();
		numList.add(0);
		numList.add(10);
		numList.add(20);
		numList.add(5);
		numList.add(15);
		numList.add(25);
		System.out.println(numList);
```

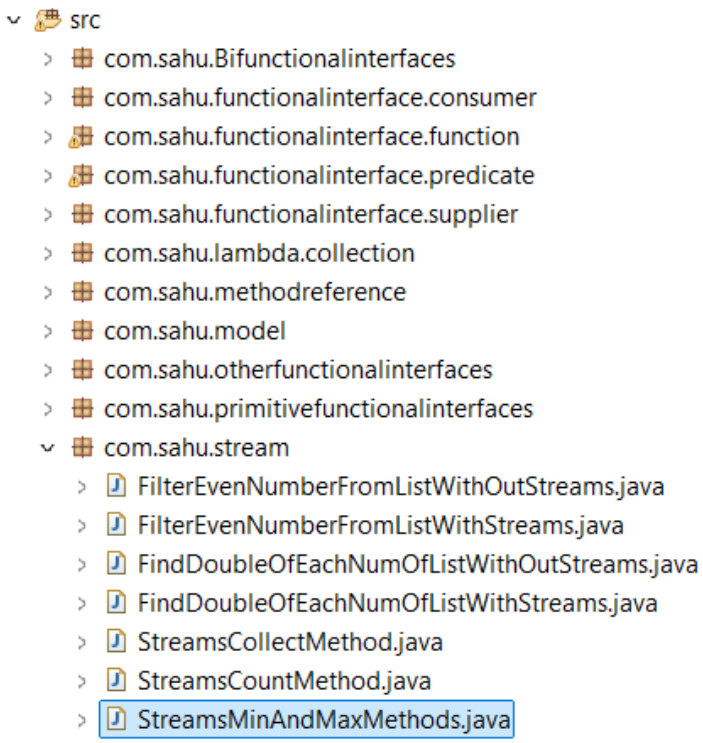
```
                List<Integer> defaultNaturalSortingList =
numList.stream().sorted().collect(Collectors.toList());
                System.out.println(defaultNaturalSortingList);

                List<Integer> customizedSortingList =
numList.stream().sorted((num1, num2) -> num2.compareTo(num1))
                                .collect(Collectors.toList());
                System.out.println(customizedSortingList);
        }

}
```

Processing by using min() and max() methods:

- min(Comparator c) - returns minimum value according to specified comparator.
- max(Comparator c) - returns maximum value according to specified comparator

```
Optional<T> min(Comparator<? super T> comparator);
Optional<T> max(Comparator<? super T> comparator);
```

Ex:



- Add the selected java files, then places the following code with in their respective file.

StreamsSortMethods.java

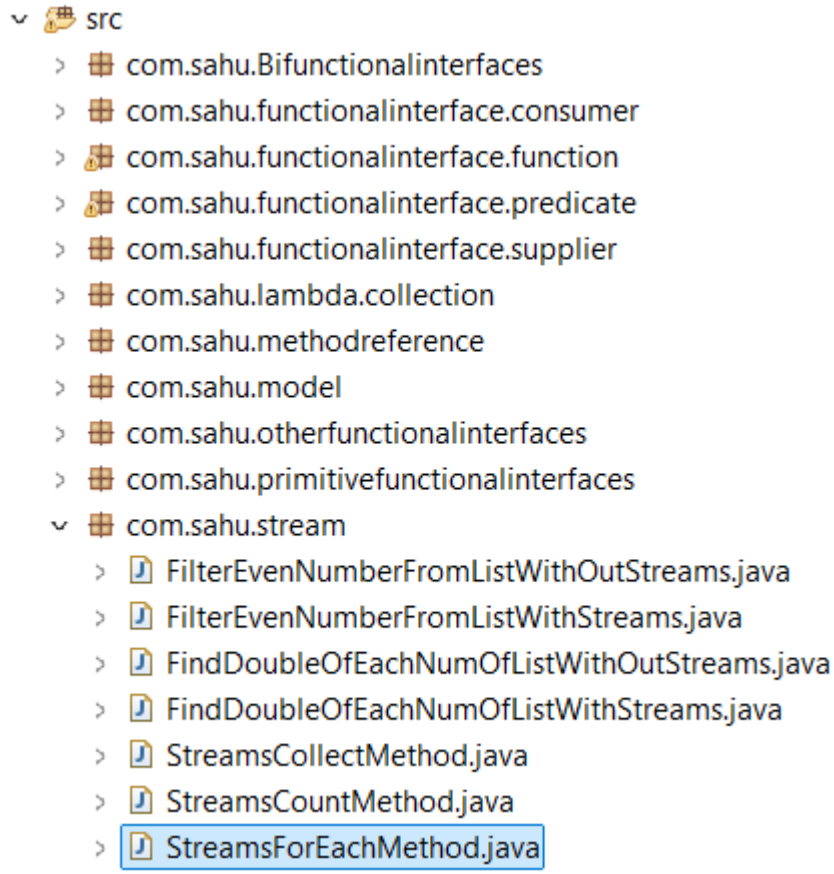
```
package com.sahu.stream;

import java.util.ArrayList;
import java.util.List;

public class StreamsMinAndMaxMethods {

	public static void main(String[] args) {
		List<Integer> numList = new ArrayList<>();
		numList.add(0);
		numList.add(10);
		numList.add(20);
		numList.add(5);
		numList.add(15);
		numList.add(25);
		System.out.println(numList);

		Integer minimumValue = numList.stream().min((num1, num2) -
> num1.compareTo(num2)).get();
		System.out.println("Minimum value is - "+minimumValue);

		Integer maximumValue = numList.stream().min((num1, num2) -
> num2.compareTo(num1)).get();
		System.out.println("Maximum value is - "+maximumValue);
	}

}
```

Processing by using forEach() method:

- This method won't return anything.
- This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.
- This method also presents in Stream interface.

```
void forEach(Consumer<? super T> action);
```

Ex:



- Add the selected java files, then places the following code with in their respective file.

StreamsForEachMethod.java

```
package com.sahu.stream;

import java.util.ArrayList;
import java.util.List;

public class StreamsForEachMethod {

	public static void main(String[] args) {
		List<String> wordList = new ArrayList<>();
		wordList.add("A");
		wordList.add("BB");
		wordList.add("CCC");

		wordList.stream().forEach(str -> System.out.println(str));
```

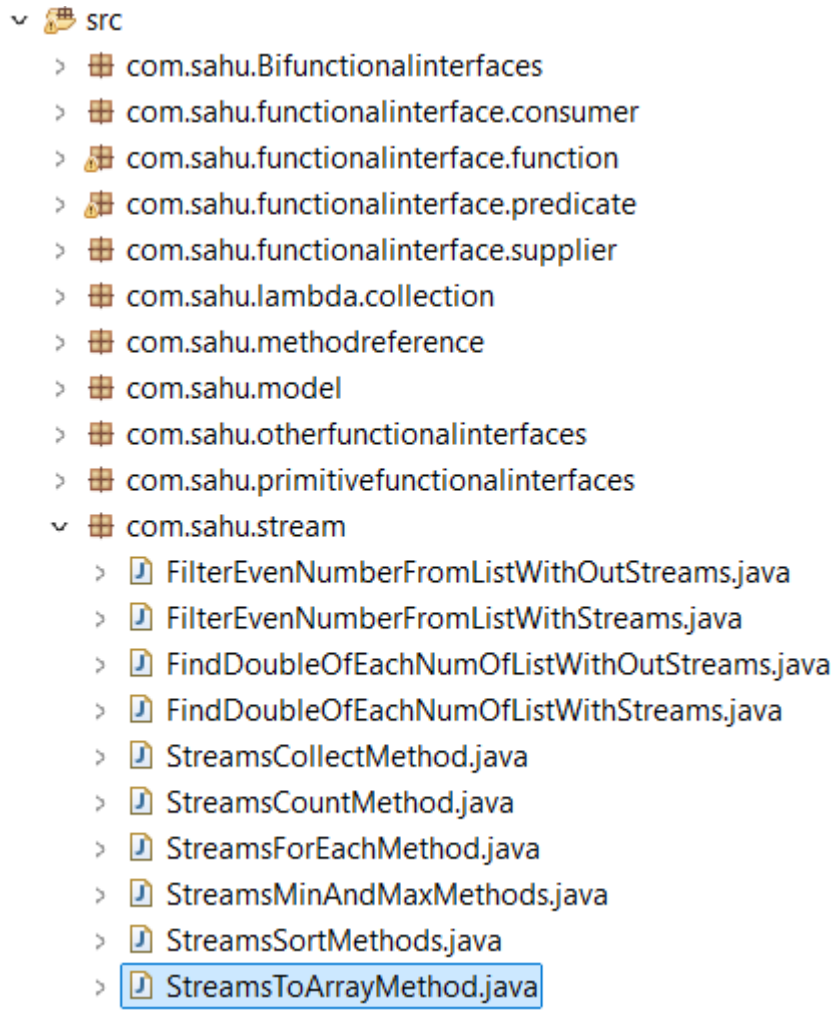
```
                wordList.stream().forEach(System.out::println);
        }

}
```

Processing by using toArray() method:

- We can use toArray() method to copy elements present in the stream into specified array.
- This method also presents in Stream interface.

```
Object[] toArray();
<A> A[] toArray(IntFunction<A[]> generator);
```

Ex:



- Add the selected java files, then places the following code with in their respective file.

StreamsToArrayMethod.java

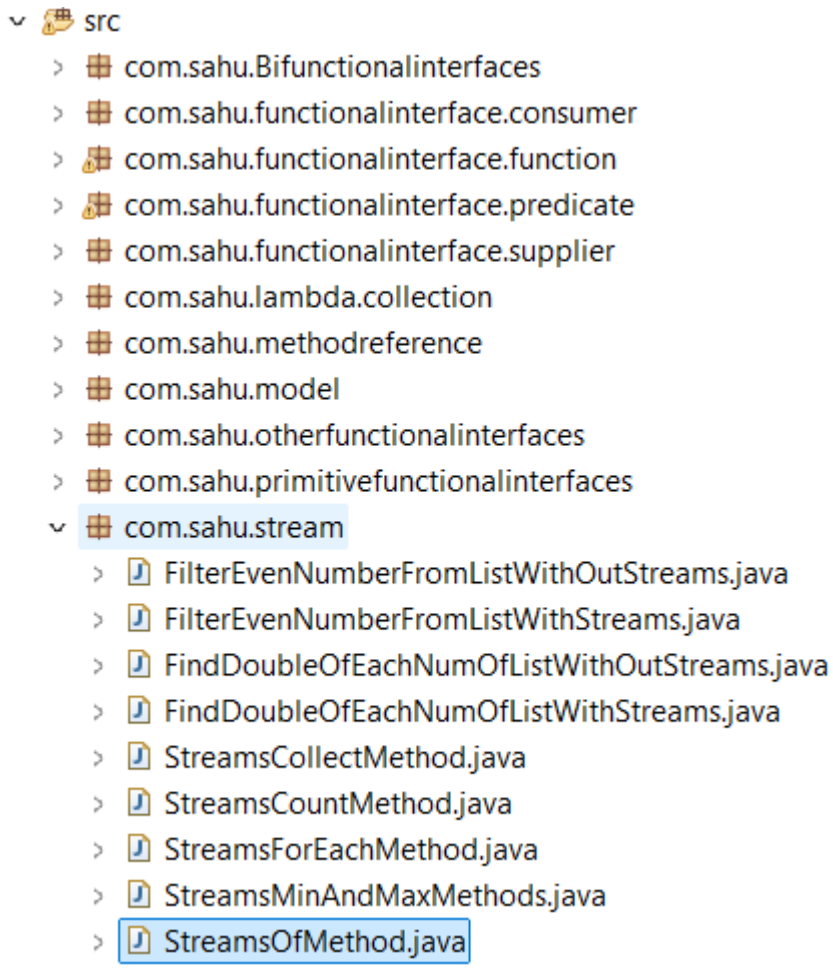
```
package com.sahu.stream;

import java.util.ArrayList;
import java.util.List;

public class StreamsToArrayMethod {

	public static void main(String[] args) {
		List<Integer> numList = new ArrayList<>();
		numList.add(0);
		numList.add(10);
		numList.add(20);
		numList.add(5);
		numList.add(15);
		numList.add(25);
		System.out.println(numList);

		Integer[] nums = numList.stream().toArray(Integer[]::new);
		for (Integer num : nums) {
			System.out.println(num);
		}
	}

}
```

Processing by using Stream.of() method:

- We can also apply a stream for group of values and for arrays.
- This method is a static method that also presents in Streams interface.

```
public static<T> Stream<T> of(T t) {
	return StreamSupport.stream(new
Streams.StreamBuilderImpl<>(t),false);
}

@SafeVarargs
@SuppressWarnings("varargs")
public static<T> Stream<T> of(T... values) {
	return Arrays.stream(values);
}
```

Ex:



- Add the selected java files, then places the following code with in their respective file.

StreamsOfMethod.java

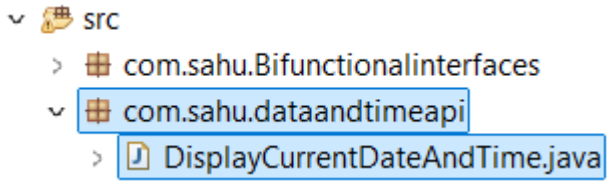
```
package com.sahu.stream;

import java.util.stream.Stream;

public class StreamsOfMethod {

	public static void main(String[] args) {
		Stream<Integer> numStream = Stream.of(9, 99, 999, 9999,
99999);
		numStream.forEach(System.out::println);

		Double[] doubleArray = {10.0, 10.1, 10.2, 10.3, 10.4};
```

```
                Stream<Double> doubleStream = Stream.of(doubleArray);
                doubleStream.forEach(System.out::println);
        }

}
```

Date and Time API (Joda-Time API)

- Until Java 1.7 version the classes present in java.util package to handle Date and Time (like Date, Calendar, TimeZone, etc.) are not up to the mark with respect to convenience and performance.
- To overcome this problem in Java 1.8 version oracle people introduced Joda-Time API. This API developed by joda.org and available in Java in the form of java.time package.

Ex: Program for to display System Date and time.

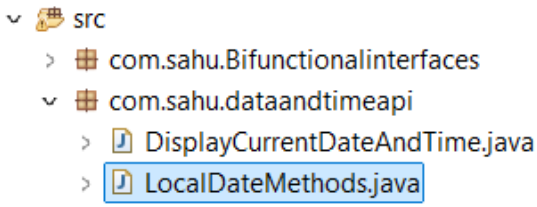


- Add the selected java files, then places the following code with in their respective file.

DisplayCurrentDateAndTime.java

```
package com.sahu.dataandtimeapi;

import java.time.LocalDate;
import java.time.LocalTime;

public class DisplayCurrentDateAndTime {

        public static void main(String[] args) {
                LocalDate date = LocalDate.now();
                System.out.println(date);
                LocalTime time = LocalTime.now();
                System.out.println(time);
        }

}
```

Ex: Once we get LocalDate object we can call the following methods on that object to retrieve Day, month and year values separately.

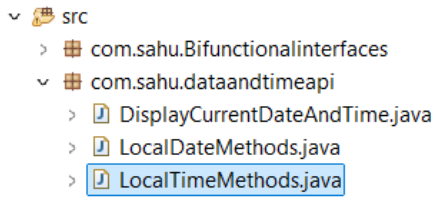


- Add the selected java files, then places the following code with in their respective file.

LocalDateMethods.java

```
package com.sahu.dataandtimeapi;

import java.time.LocalDate;

public class LocalDateMethods {

	public static void main(String[] args) {
		LocalDate date = LocalDate.now();
		System.out.println(date);

		int dd = date.getDayOfMonth();
		int mm = date.getMonthValue();
		int yyyy = date.getYear();

		System.out.println(dd+"-"+mm+"-"+yyyy);
		System.out.printf("%d-%d-%d", dd, mm, yyyy);
	}

}
```

Ex: Once we get LocalTime object we can call the following methods on that object.

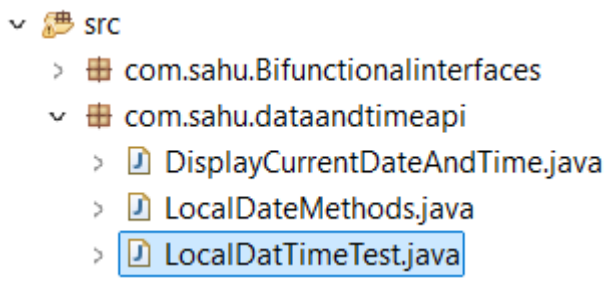


- Add the selected java files, then places the following code with in their respective file.

LocalTimeMethods.java

```
package com.sahu.dataandtimeapi;

import java.time.LocalTime;

public class LocalTimeMethods {

	public static void main(String[] args) {
		LocalTime time = LocalTime.now();
		System.out.println(time);

		int hour = time.getHour();
		int minute = time.getMinute();
		int second = time.getSecond();
		int nano = time.getNano();
		System.out.printf("%d:%d:%d:%d", hour, minute, second,
nano);
	}

}
```

Ex: If we want to represent both Date and Time then we should go for LocalDateTime object.



- Add the selected java files, then places the following code with in their respective file.

LocalDateTimeTest.java

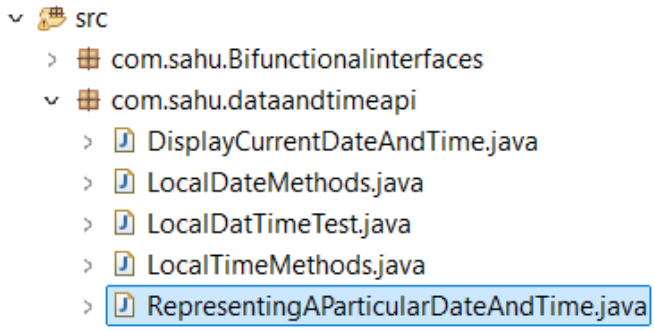
```
package com.sahu.dataandtimeapi;
```

```
import java.time.LocalDateTime;

public class LocalDatTimeTest {

	public static void main(String[] args) {
		LocalDateTime dateTime = LocalDateTime.now();
		System.out.println(dateTime);

		int day = dateTime.getDayOfMonth();
		int month = dateTime.getMonthValue();
		int year = dateTime.getYear();
		System.out.printf("%d-%d-%d", day, month, year);

		int hour = dateTime.getHour();
		int minute = dateTime.getMinute();
		int second = dateTime.getSecond();
		int nano = dateTime.getNano();
		System.out.printf("%d:%d:%d:%d", hour, minute, second,
nano);
	}

}
```

Ex: We can represent a particular Date and Time by using LocalDateTime object as follows.

LocalDateTime dateTime = LocalDateTime.of(yy, mm, dd, h, m, s, n);

Ex:



- Add the selected java files, then places the following code with in their respective file.

LocalDateTimeTest.java

```
package com.sahu.dataandtimeapi;

import java.time.LocalDateTime;
import java.time.Month;

public class RepresentingAParticularDateAndTime {

	public static void main(String[] args) {
		LocalDateTime dateTime =
LocalDateTime.of(1995,Month.APRIL,28,12,45);
		System.out.println(dateTime);

		System.out.println("After six months -
"+dateTime.plusMonths(6));
		System.out.println("Before six months -
"+dateTime.minusMonths(6));
	}

}
```

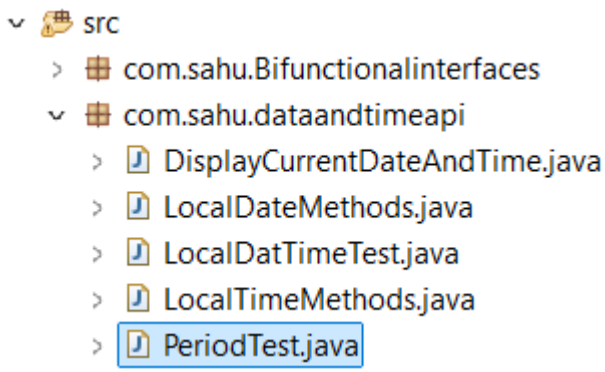
Period:

- Period object can be used to represent quantity of time

Syntax:

Period p = Period.between(starting day, today);

Ex: Using of Period class

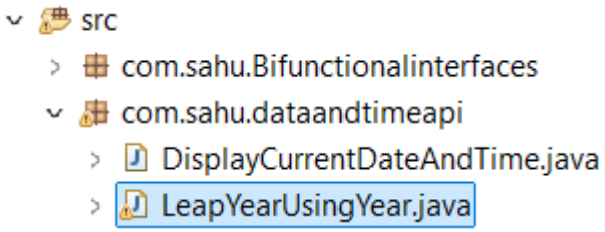


- Add the selected java files, then places the following code with in their respective file.

LeapYearUsingYear.java

```
package com.sahu.dataandtimeapi;

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

public class PeriodTest {

	public static void main(String[] args) {
		LocalDate today = LocalDate.now();
		LocalDate birthDay = LocalDate.of(1989, 8, 28);
		Period period = Period.between(birthDay, today);
		System.out.printf("Age is %d years %d monthos %d days",
period.getYears(), period.getMonths(),
				period.getDays());

		LocalDate deathDay = LocalDate.of(1989 + 60, 6, 15);
		Period deathPeriod = Period.between(today, deathDay);
		int daysOnEarth = deathPeriod.getYears() * 365 +
deathPeriod.getMonths() * 30 + deathPeriod.getDays();
		System.out.printf("\nYou will be in the earth only %d days..
Hurry up to do more important things", daysOnEarth);
	}

}
```

Ex: Check year is leap year or not by using Year class.



- Add the selected java files, then places the following code with in their respective file.

LeapYearUsingYear.java

```
package com.sahu.dataandtimeapi;
```

```java
import java.time.Year;
import java.util.Scanner;

public class LeapYearUsingYear {

	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);
		System.out.print("Enter year: ");
		int numYear = scanner.nextInt();
		Year year = Year.of(numYear);
		if (year.isLeap()) {
			System.out.printf("%d year is Leap year", numYear);
		}
		else {
			System.out.printf("%d year is not Leap year", numYear);
		}
	}

}
```

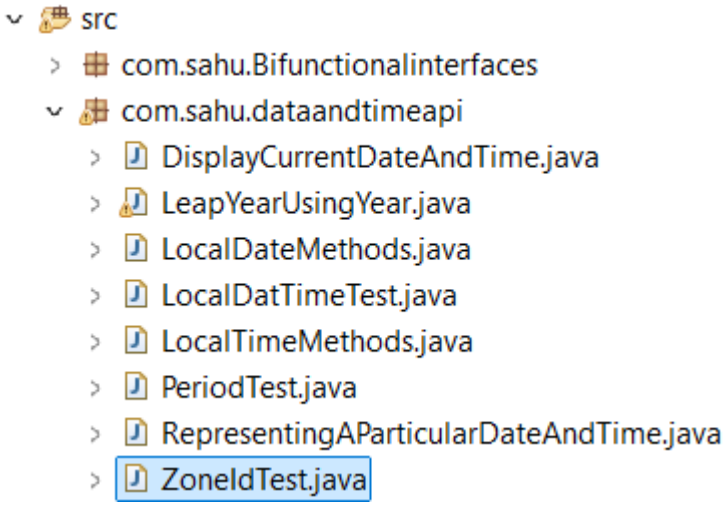
To Represent Zone:

- ZoneId object can be used to represent Zone.

Syntax:

```
ZoneId zone = ZoneId.systemDefault();
```

Ex: Use of ZoneId class

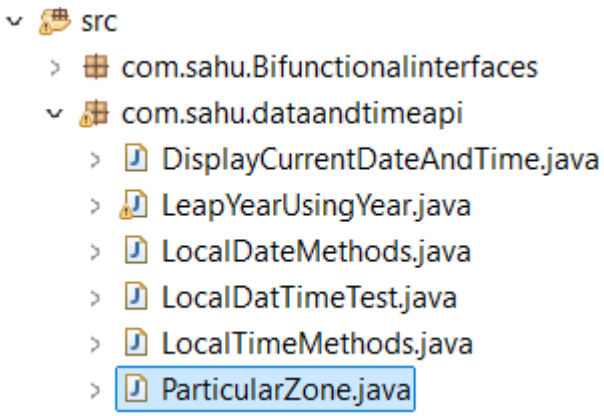


- Add the selected java files, then places the following code with in their respective file.

ZoneIdTest.java

```
package com.sahu.dataandtimeapi;

import java.time.ZoneId;

public class ZoneIdTest {

	public static void main(String[] args) {
		ZoneId systemDefaultZone = ZoneId.systemDefault();
		System.out.println(systemDefaultZone);
	}

}
```

Ex: We can create ZoneId for a particular zone as follows



- Add the selected java files, then places the following code with in their respective file.

ParticularZone.java

```
package com.sahu.dataandtimeapi;

import java.time.ZoneId;
import java.time.ZonedDateTime;

public class ParticularZone {
```

```
        public static void main(String[] args) {
                ZoneId americaLAZone = ZoneId.of("America/Los_Angeles");
                ZonedDateTime americaLAZoneDateTime =
ZonedDateTime.now(americaLAZone);
                System.out.println(americaLAZoneDateTime);
        }

}
```

Links to Practice

https://medium.com/@basecs101/list/all-java-8-features-and-best-practices-that-you-must-know-2ce2c14f1257

https://blog.devgenius.io/java-8-coding-and-programming-interview-questions-and-answers-62512c44f062

https://github.com/rohitchavan-git/Java-8-Interview-Sample-Coding-Questions

https://github.com/RameshMF/java-8-tutorial

https://javaconceptoftheday.com/java-8-interview-sample-coding-questions/

https://stackify.com/streams-guide-java-8/

https://www.javaguides.net/p/java-8.html

https://mkyong.com/tutorials/java-8-tutorials/

---------- The END ----------