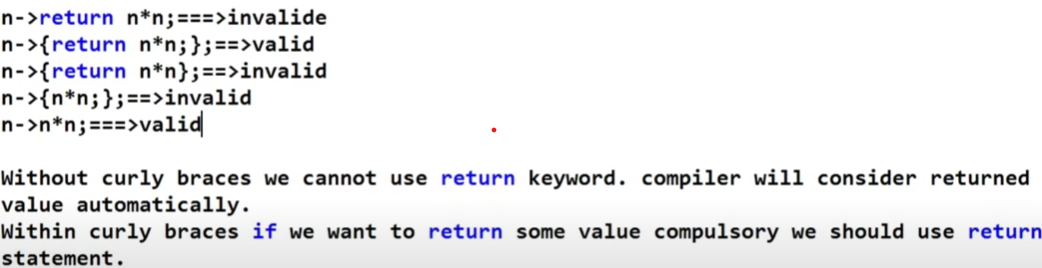
* **Ternary conditional operator:**

1. **?** This operator used to check condition.
2. **:** This operator used to if-else condition.

**Eg.** **condition ? value\_if\_true : value\_if\_false**

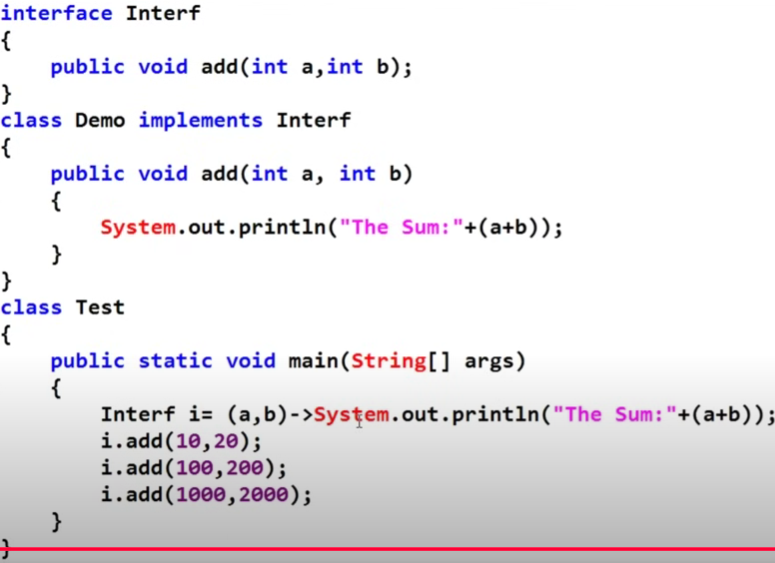
* **How to write return type using lambda expression?**

****

* **Why it is introduce java 8 features?**
* Simplifying the programing style.
* Functional programming provide.(inbuild function, expression provide).
* Concise code(reduce line of code).
* Java 8 feature came in 2014 march 18th
* Current version of java 24

1. **Lambda expression :-** It is anonymous (Nameless) function which provides implementation of single unimplemented method of functional interface .

* The main objective of lambda expression to bring benefits of functional programming into java.
* Lambda expression only applicable for functional interface.
* Lambda expression can implements an interface which contain a single abstract method.



**Q. What Is anonymous Inner Class?**

* The class without having name such type of inner class it is called as anonymous inner class.
* Anonymous inner class can extend normal class.
* Anonymous inner class can extend an abstract class
* Anonymous inner class can implement an interface which contain any number of abstract method

**IQ. Why anonymous function?**

* Not an in the method name.
* Not an in the return type.
* Not an in the modifiers.

**Q. Why anonymous inner class better than lambda function?**

* We can write in interface multiple abstract method in anonymous inner class.
* We can write only one abstract method in lambda function.
* Anonymous inner class & lambda function both are not equal.

1. **Default and static method in interface:-**
2. **Before 1.7V:-** Every method present inside interface is always public and abstract

**Eg. void m1();**

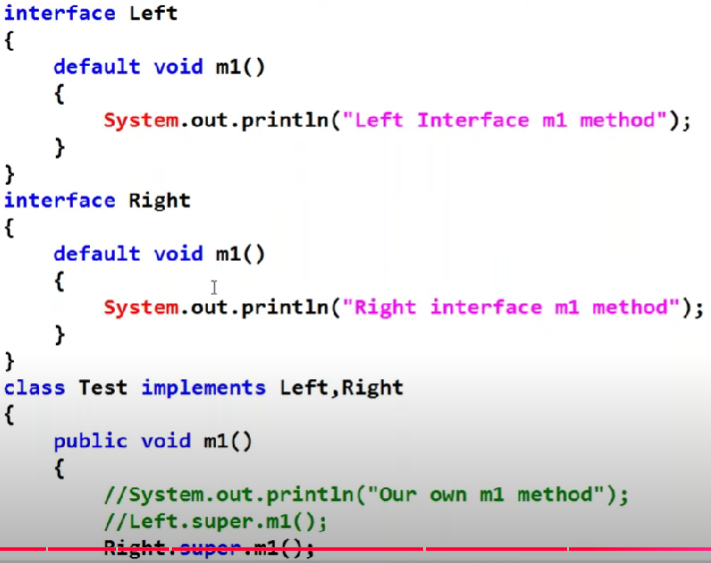
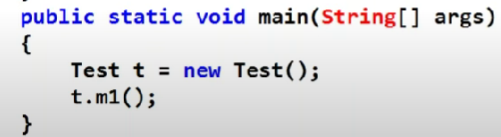
**public void m1();**

**abstract void m1();**

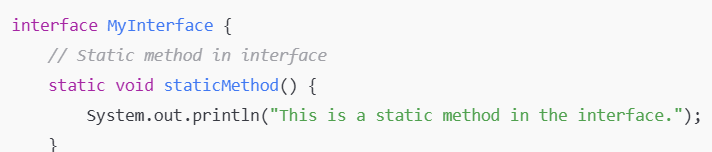
**public abstract void m1();**

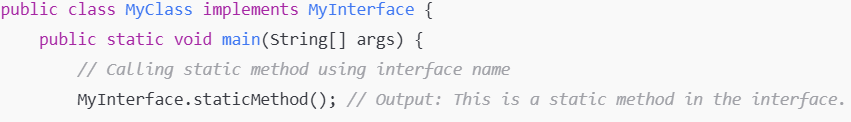
1. **After 1.8v :-** Inside interface in the 1.8v **default method and static methods** are allowed
   * Every variable inside interface are **public static final**
2. **Default Method():-**

* **Default method** has a method body.
* It is defined using the default keyword.
* It can be overridden by the implementing class, but it's not mandatory.
* Without effecting implementation classes if we want to add new method to the interface then you should go for default method.
* It is Default method mention in only interface not in class.
* "If you write a default method with the same method name in two interfaces, and a class implements both interfaces, it causes ambiguity because the class is confused about which method to call."
* "If you want to call the default method from the first or second interface without ambiguity, you can write a non-static method in the class and call the default method using the interface name in a static method."

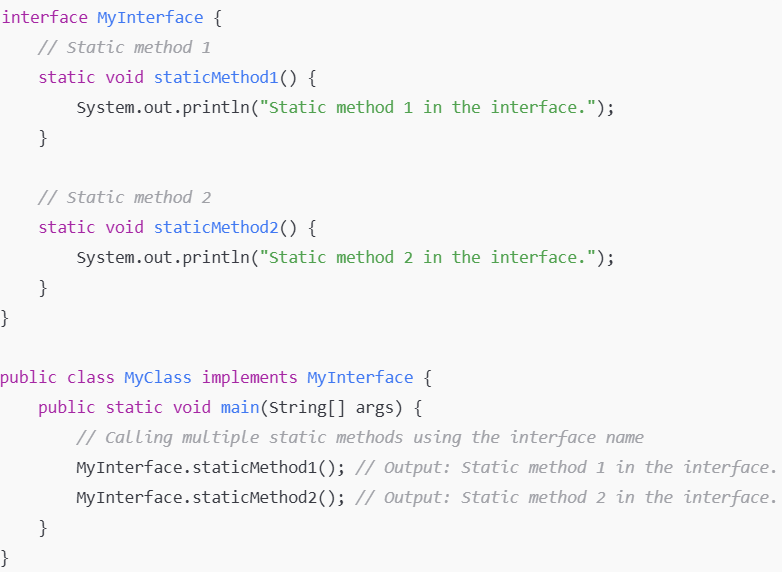
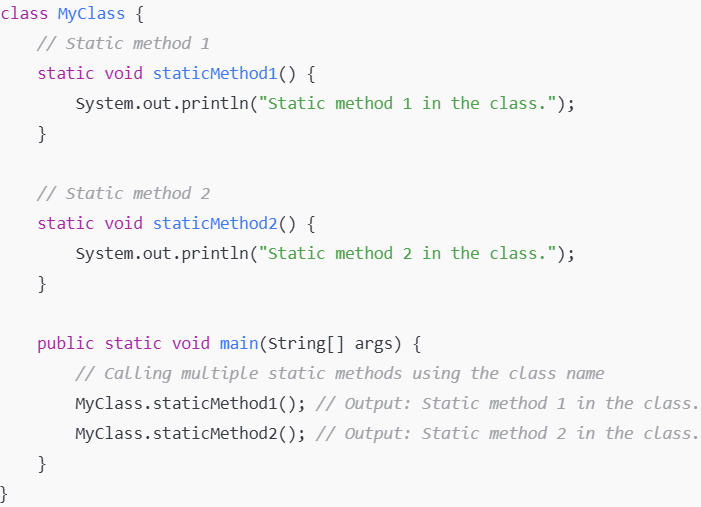
 

**2) static method():-**

* Static methods in interfaces are called using the interface name, not through object of classes that implement the interface.
* **Eg **

****

* **In an Interface (Java 8)**: You can write multiple static methods using the static keyword, but they cannot be overridden by implementing classes. They must be accessed using the interface name.
* **In a Class**: You can write multiple static methods, and they can be called using the class name. Static methods in classes can be overridden in subclasses (unless marked final).

**Q. Why static method declare in interface?**

* Static methods in interfaces are not inherited by implementing classes.
* Static methods belong to the interface and can be called using the interface name (not the class implementing it).
* They are useful for utility functions or common operations related to the interface.
* **Optional class**

1. **Method references:-** One method is referencing another method.

* Method reference is an alternative for lambda expression.

1. **Constructor references**
2. **Parallel programming**

* **Why introduce default method in interface?**
* We can use share common code.
* Flexible code
* Abstract method
* We can use only Default method in interface not in class.
* You can override default method but it is meaningless.

**Q.** If class B extends A implements I1,I2 and all of them have same method m1.Then class A method gets override. Because class have more priority than interface.

* **Lambda expression is used to provide implementation of a method of Interface.**

**Q. Lambda expression concept is taken from Anonymous inner class**

**Q. Can we provide implements for method without implementing an interface to a class?**

* Yes we can do it by creating interface(obj) of interface and inside that we can provide implementation.
* Eg. I1 i1= new I1(){ public void m1(){sop(“”)}};

**3. Functional inerface (java.util.function):-**

* Only one abstract method. It is also known as function interface
* The use of functional interface if you want to lambda expression compulsory functional interface is required.
* **@FunctionalInterface:-** It is not mandatory. But advantage of functionalinterface is if you don’t write one abstract method if you doing any mistake immediately compile time error.
* **Functional Interface Before java1.8 :-**

1. **Comparator (I)(1.2V):-**
2. **compare() :-**
3. **Comparable (I) (1.2):-**
4. **compareTo():-** It is applicable for alphabetical order if are string object
5. **Runnable (I) :-**
6. **run():-**

* **Predefine Functional Interface java1.8 (java.util.function):**-

1. **Function Interface:- Function<T, R>** interface can accept any input type T and return **Function<R,T>** any output type R.
2. **apply():-**
3. **Others Useful Method in Function<T,R> :**
4. **andThen():-**

 The **andThen** method allows you to chain multiple functions together. In simpler terms, it runs the first function and then applies the second function to the result of the first.

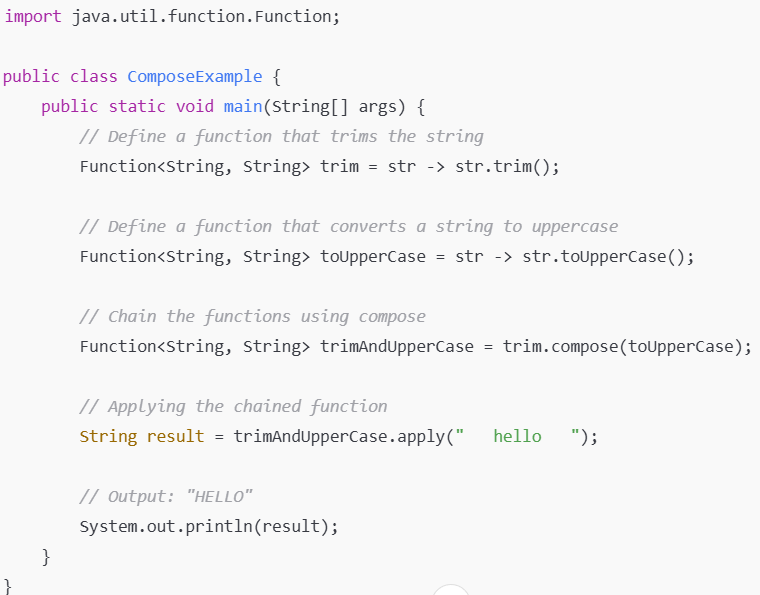
* Example: function1.andThen(function2) applies function1 first and then applies function2 to its result.

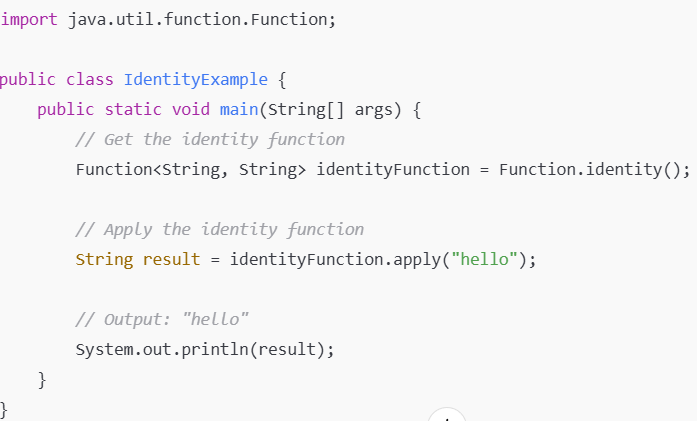
1. **Compose()**:

* This method allows you to chain a function to be applied before the current function. The argument to the current function is passed as the result of the function before it.
* Example: function1.compose(function2) applies function2 first and then applies function1 to its result.

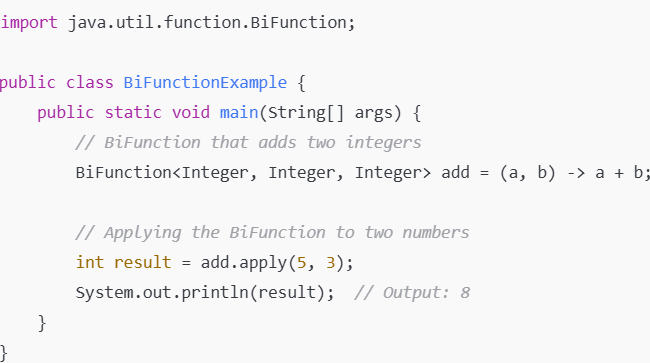
1. **identity()**:

* This static method returns a function that always returns its input argument (i.e., it acts as an identity function).
* The identityFunction simply returns whatever value is passed to it.

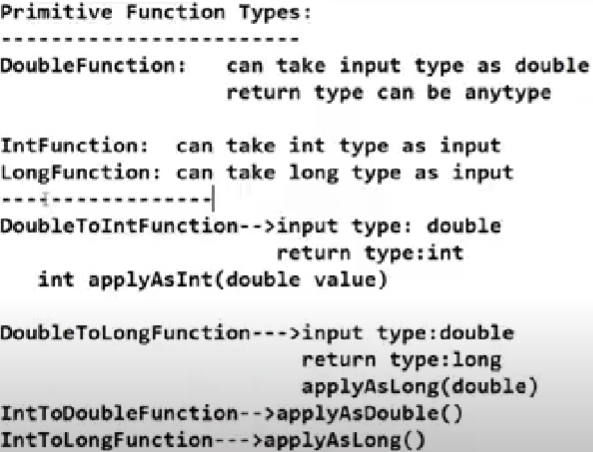
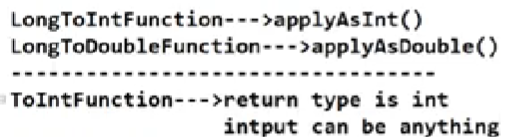




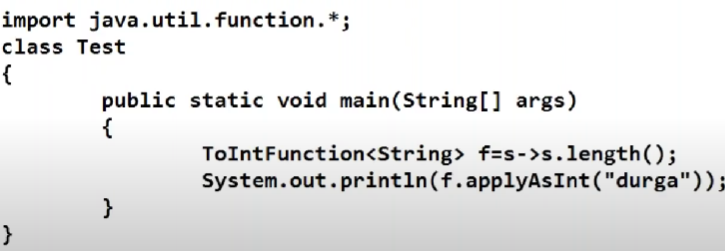
* **BiFunction Interface:-** A BiFunction takes two arguments (of types T and U).It returns a single result of type R

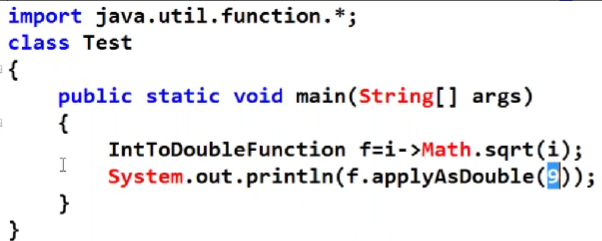
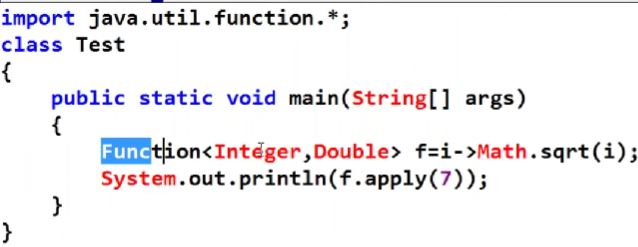


* **Premitive Function Type:**

** **

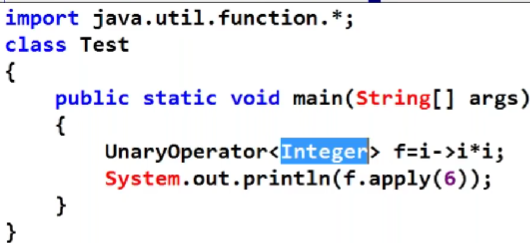
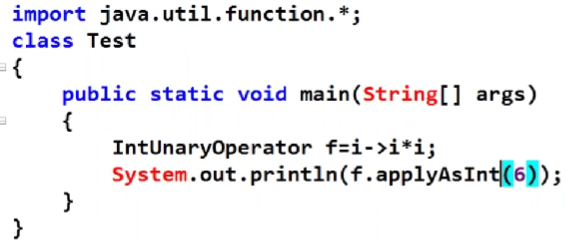
**Q.Example of IntFunction:- I have pass String value but I want to return int?**

****

**Q. Example Of IntToDoubleFunction:- I have pass int value but I want to return double**

**Q. What is UnaryOperator?**

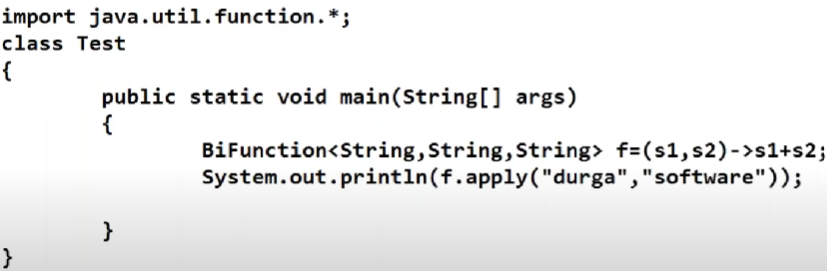
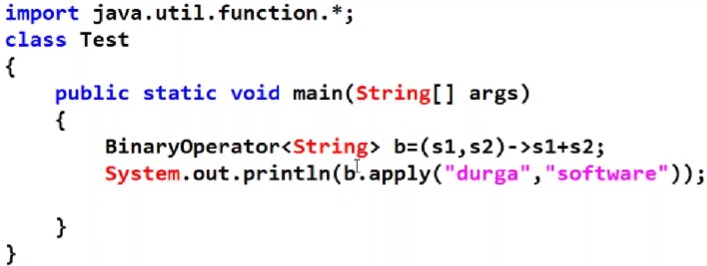
* UnaryOperator interface is child of Function interface.
* The main purpose of UnaryOperator is both the input and the output are of the same type T(means you pass the value integer type and I want to return integer in that case use of unaryoperator)
* UnaryOperator extends the Function<T, T> interface, so it inherits the apply() method, which accepts one argument and returns a value of the same type.
* **First Way Second Way**

**** ****

**Q. what is Binary Operatior:-**

* The main purpose of BinaryOperator is both the input & output types are the same.

**First way -> Second Way**

**** ****

1. **Predicate Interface:- Predicate<T>** always takes an input of type T and returns a boolean result (true or false).

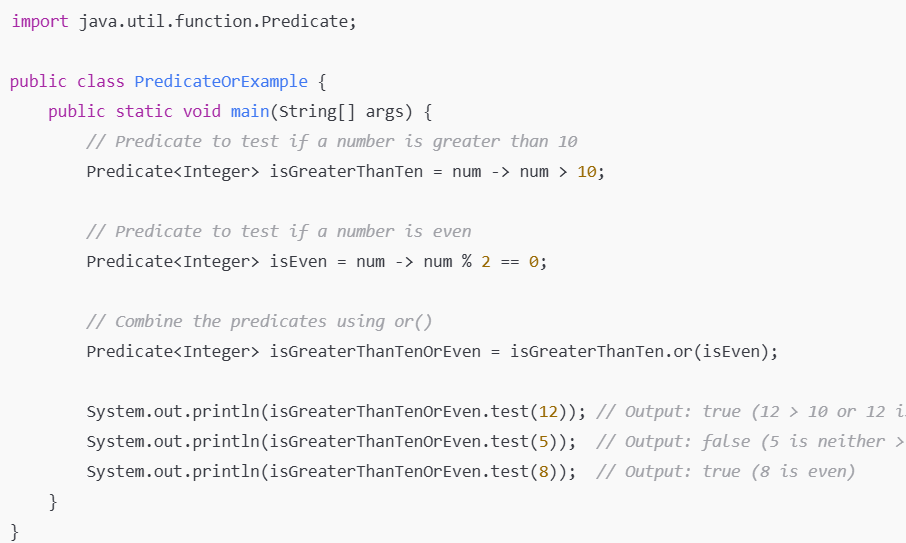
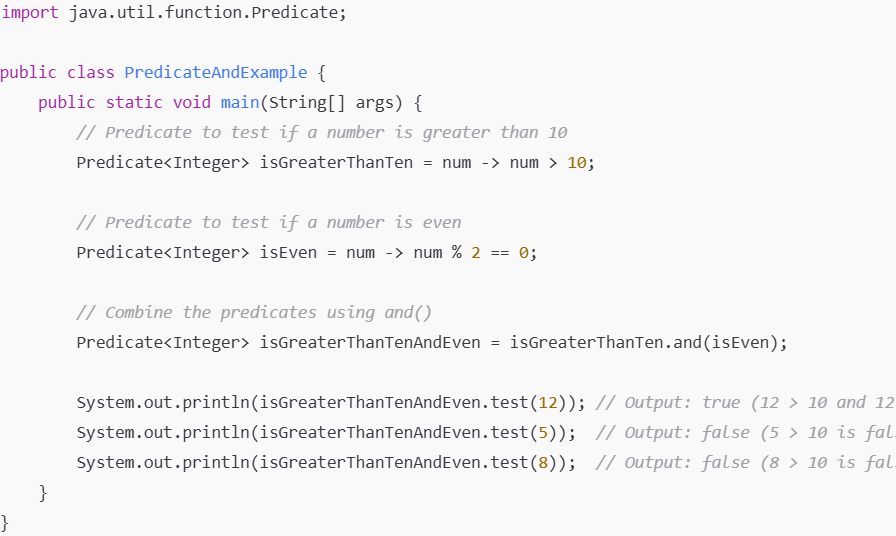
* Predicate interface contain only one abstract method.
* How to internally working predicate

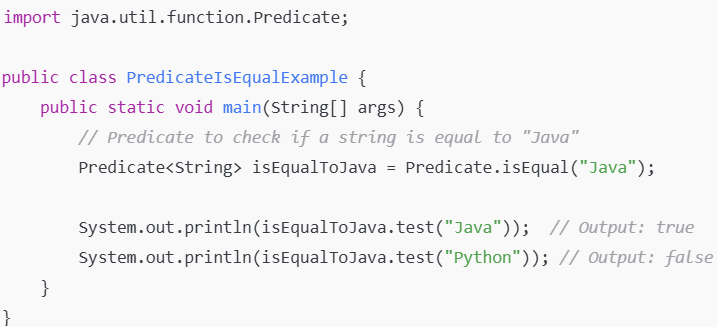
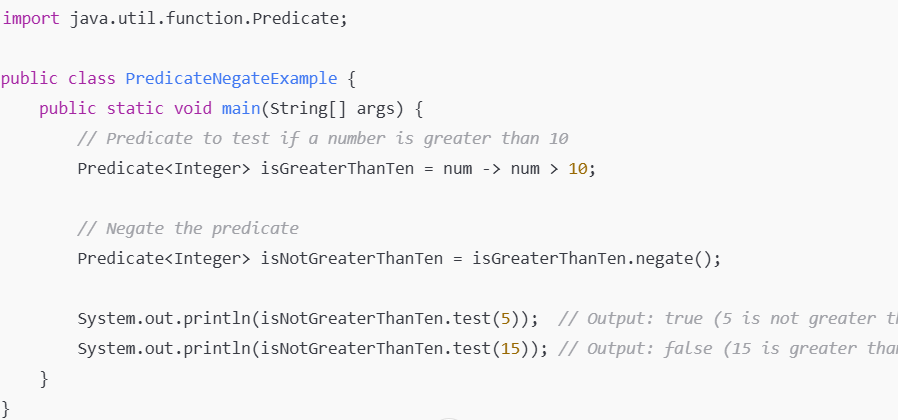
Int(primitive DT) =>convert into => Integer(non-pre-DT)=>int

That means autoboxing to autounboxing(Premitive DT convert into wrapper class and wrapper class convert into again primitive DT)

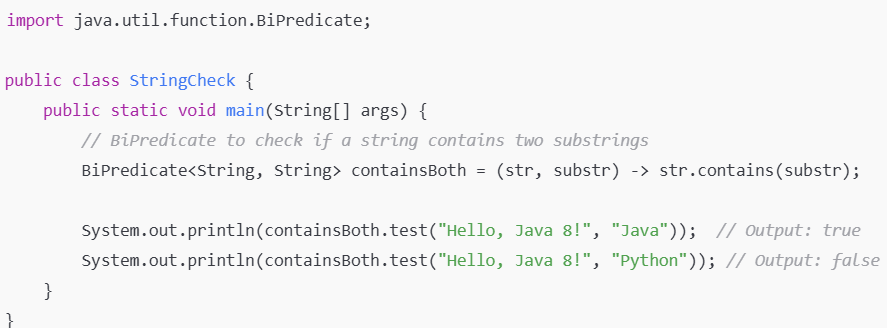
1. **test():-**
2. **Default Methods in Predicate Interface:-**
3. **and()**: -The and() method returns true only when both conditions are satisfied.
4. **or()**:-The or() method returns true if at least one of the predicates is true.
5. **negate():-** The negate() method allows you to reverse the condition of a predicate.
6. **isEqual():-**The isEqual() method is a static method in Predicate<T>.

isEqual() it is check both input matches or not.





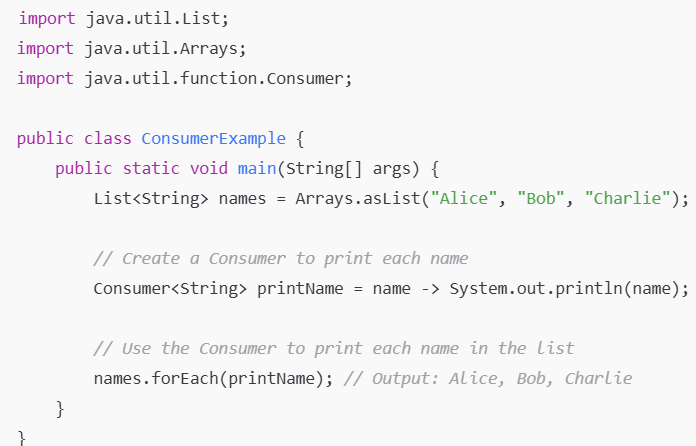
* **BiPredicate Interface:-** It represents a functional interface that takes **two arguments** and returns a boolean value. But only **Predicate interface** only take **one argument.**



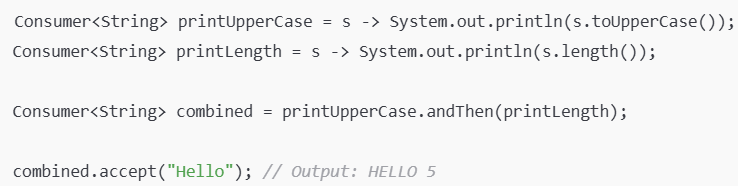
1. **Consumer Interface:- Consumer<T>** is a functional interface that takes a single input of type T and performs an operation on that input and no returning a result.

* Chaining method is allowed **andThen().**

1. **accept():-** The accept(T t) method is the key method, used to perform the action on the input.



1. **Default Method in Consumer Inteface:-**
2. **andThen():**-The Consumer interface also provides a default method called **andThen(),** which allows you to chain multiple consumer operations.



* **BiConsumer Interface:-** BiConsumer accepts two input arguments (of types T and U).

**import java.util.function.BiConsumer;**

**public class BiConsumerExample {**

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

**// Create a BiConsumer that prints the name and age**

**BiConsumer<String, Integer> printDetails = (name, age) -> System.out.println(name + " is " + age + " years old.");**

**// Apply the BiConsumer to print details**

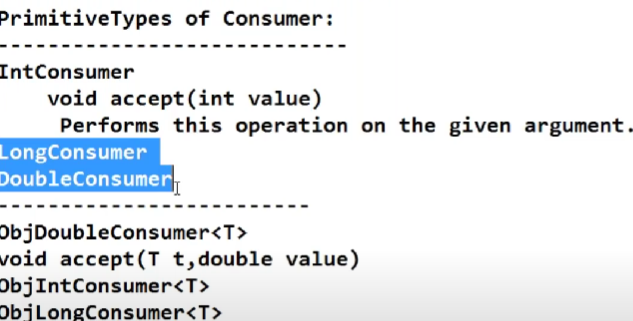
**printDetails.accept("Alice", 25); // Output: Alice is 25 years old.**

**printDetails.accept("Bob", 30); // Output: Bob is 30 years old.**

**}**

**}**

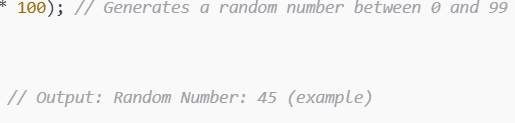
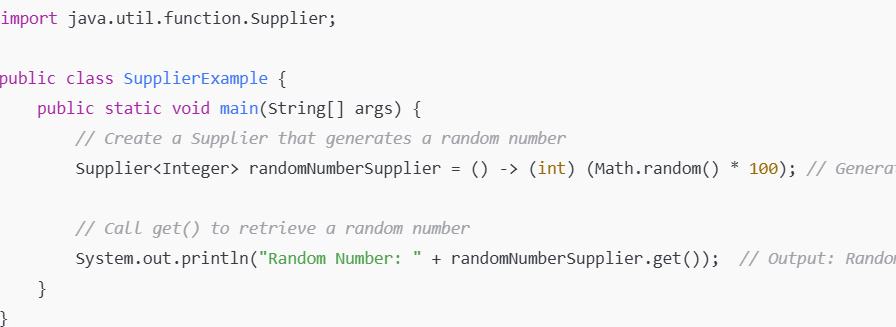
* **Primitives Types of Consumer:-**

****

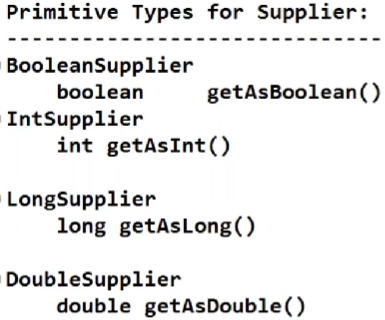
1. **Supplier Interface :-** The Supplier doesn't take any argument.

* It only provides a result, but doesn't modify any input.
* **Eg.** It can be used to generate values, such as creating new objects, numbers, or even random values.

1. **get():-** This method doesn't take any parameters but returns a value of type T.



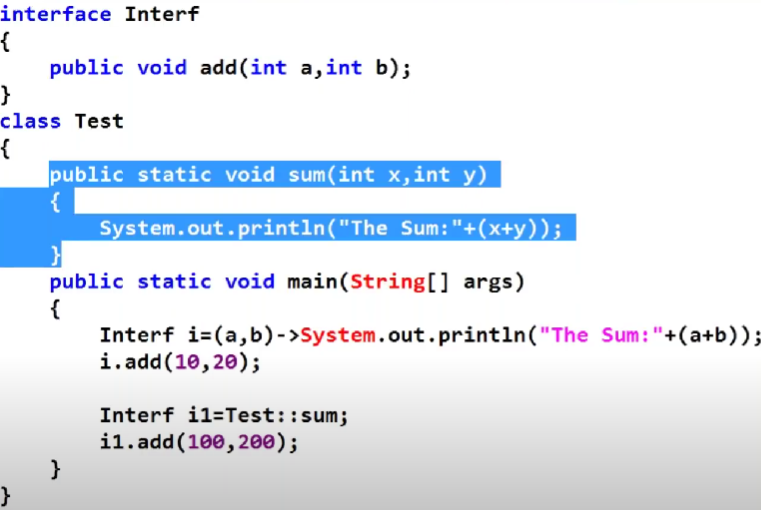
* **Primitives of Supplier:-**

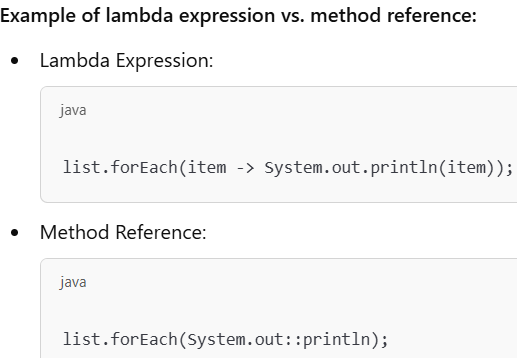


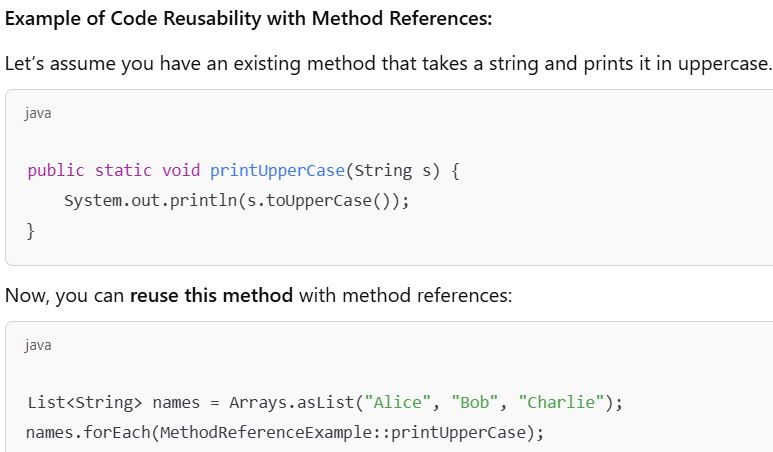
**4.Method () & Constructor ():-**

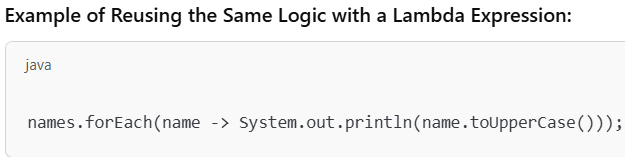
1. **Method () reference:-**

* Method references and constructor references provide a shorter, more concise way of writing code compared to lambda expressions.
* It is alternative way of lambda expression are method & constructor reference
* If implementation is already available , you can use it method reference.
* If implementation is already not available , you can use it lambda expression.



* It is Code reusability.
* 
* Method references allow you to **directly reuse** an existing method or constructor without having to re-implement its logic in the lambda.

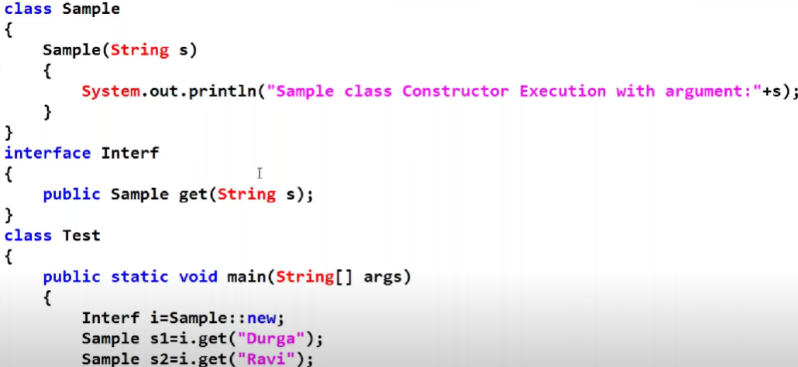


* 

1. **Constructor Reference :-**

* If the functional interface methods returns an object in that particular cases, you can use constructor reference.
* Argument is same required

"If multiple constructors are created, but I want to specify a particular constructor when calling it by passing the argument."

****

****

* **Optional Class:-**
* The optional is a class of the “java.util” package which is introduce in java8 to avoid NullPointerxception it is a public final class.
* **Stream API:- Stream is a interface .stream is a sequence of object/element**
* **API:- menas APPLICATION PROGRAMMING INTRFACE (multiple classes & interface).**
* **Working principle for stream:-**
* **Pipeline of operations:-**
* **Stream to processes different operation on collection object or array.**
* **Source=>intermediate operation =>terminal operation**

**Source means stream**

**Intermediate means applying any method and return stream**

**Terminal means print the data .**

* **Stream is lazy loder because**
* **Stream is immutable can’t be changed**

|  |  |
| --- | --- |
| **Stream()** | **ParallelStream()** |
| **Only single thread work** | **Multiple thread work (Huge amount of data work )** |
| **Slower** | **Faster** |

* **Difference between map and flatmap**