Q) Why JAVA 8 ? Main agenda behind Java 8

- Significant reason for introducing JAVA 8 was to introduce Conciseness in the code.
- JAVA brings in Functional programming which is enabled by Lambda expressions (a powerful too to create concise code base.)
- If you have ever observed, with Python, Scala we can do the same thing in Very less LOC. By mid 20s JAVA lost a large market due to these languages. To prevent further loss java upgraded itself from Only OOPs language to some concepts of FP to create concise code base.

Q) What are new features which got introduced in Java 8?

New Features	Lambda Expression
	Stream API
	Default methods in the interface
	Static methods
	Functional Interface
	Optional
	Method references
	Date API
	Nashorn, JavaScript Engine

Q) What are main advantages of using Java 8?

Main advantages	Compact code (Less boiler plate code.)
	More readable and reusable code
	More testable code
	Parallel operations

Q) What is lambda expression?

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

Normal programming technique

```
public void add(int a, int b) {
        System.out.println(a+b);
}
```

Equivalent Lambda Expression

```
(a, b) -> System.out.println(a + b);
```

Q) What are functional interfaces?

- Functional interfaces are those interfaces which can have only one abstract method .
- It can have any number of static method, default methods. No restriction on that
- There are many functional interfaces already present in java such as eg : Comparable, Runnable

Q) How lambda expression and functional interfaces are related?

- **Functional Interface is used to provide reference to lambda expressions. --→ This is the relation.**
- Comparator<String> c = (s1,s2) → s1.compareTo(s2);
- (s1,s2) → s1.compareTo(s2) : This is lambda Expression
- Comparator<String> c : This is Functional Interface
- Thus you can see , To call lambda expressions we need Functional Interfaces.
-

Q) Can you create your own functional interface?

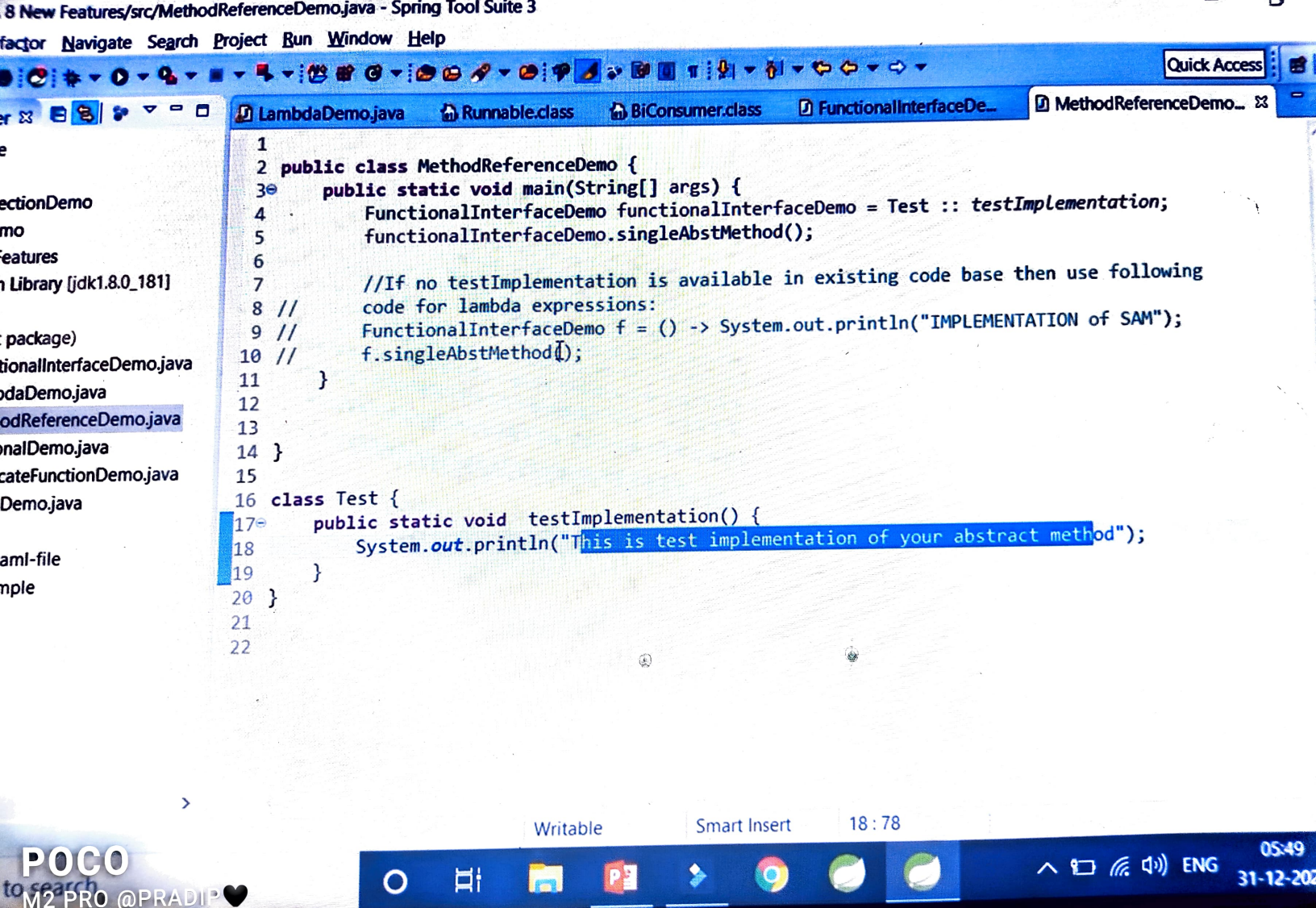
- As we know Functional interface is an interface with Exactly One Single Abstract method and can have multiple Static or default methods.

- To create our own Functional interface, You can do following steps:
  - Create An interface
  - Annotate that with @FunctionalInterface.
  - Define exactly one Abstract method.
  - There is no restriction on number of static and default methods defined in such and interface.

- Java can implicitly identify functional interface but still you can also annotate it with @FunctionalInterface . It just give you the security that in case if u by mistake add 2 abstract methods then Compiler will throw **compile time error.**

Q) What is method reference in java 8?

- EG : MethodReferenceDemo and FunctionalInterfaceDemo
- Method reference is replacement of lambda expressions. It is used to refer method of Functional interface to an existing method. Mainly it is used for code reusability.
- Functional Interface's Abstract method can be mapped to specific existing method using double colon operator (::) . This is Method reference.
- Hence Method reference is an alternative to Lambda expressions.
- Whenever we have existing Implementation of Abstract method of our Functional interface then we can go for method reference. If no such method like testImplementation() is available then go for lambda expressions.


8 New Features/src/MethodReferenceDemo.java - Spring Tool Suite 3
factor Navigate Search Project Run Window Help
Quick Access
LambdaDemo.java
Runnable.class
BiConsumer.class
FunctionalInterfaceDe...
MethodReferenceDemo...
ectionDemo
mo
Features
n Library [jdk1.8.0_181]
package)
tionalInterfaceDemo.java
odaDemo.java
odReferenceDemo.java
onalDemo.java
cateFunctionDemo.java
Demo.java
aml-file
mple
1
2 public class MethodReferenceDemo {
3 public static void main(String[] args) {
4 FunctionalInterfaceDemo functionalInterfaceDemo = Test :: testImplementation;
5 functionalInterfaceDemo.singleAbstMethod();
6
7 //If no testImplementation is available in existing code base then use following
8 // code for lambda expressions:
9 // FunctionalInterfaceDemo f = () -> System.out.println("IMPLEMENTATION of SAM");
10 // f.singleAbstMethod();
11 }
12
13
14 }
15
16 class Test {
17 public static void testImplementation() {
18 System.out.println("This is test implementation of your abstract method");
19 }
20 }
21
22
Writable
Smart Insert
18 : 78
POCO
M2 PRO @PRADIP
ENG
05:49
31-12-202

Q) What are defaults methods?

- Default method is a way for adding new methods to the interface without affecting the implementing classes. Hence with this new feature Java people defended many compile time errors that may arise due to unimplemented methods of interface.

- Eg: interface Animal{
  void legs();
  // default void eyes(){
  Syso ("two eyes")}
}

- The use of these default methods is "Backward Compatibility" which means if JDK modifies any Interface (without default method) then the classes which implement this Interface will break.

Q) Is it necessary to override default methods?

- Default methods have dummy implementations.
- Implementing classes if ok with dummy implementation then use dummy impl of default method
- If not satisfied then they can override and provide their own implementation.

Q) Is default keyword one of the access modifier?

Default is not the access modifier like public or protected or private.
For default access modifier we dot use any keyword.

Hence default keyword was only used in classes till 1.8 version for switch case only but never in interface.
Now its used for default methods in interface to provide a default implementation for all implementing classes to use.

Q) How to override default methods?

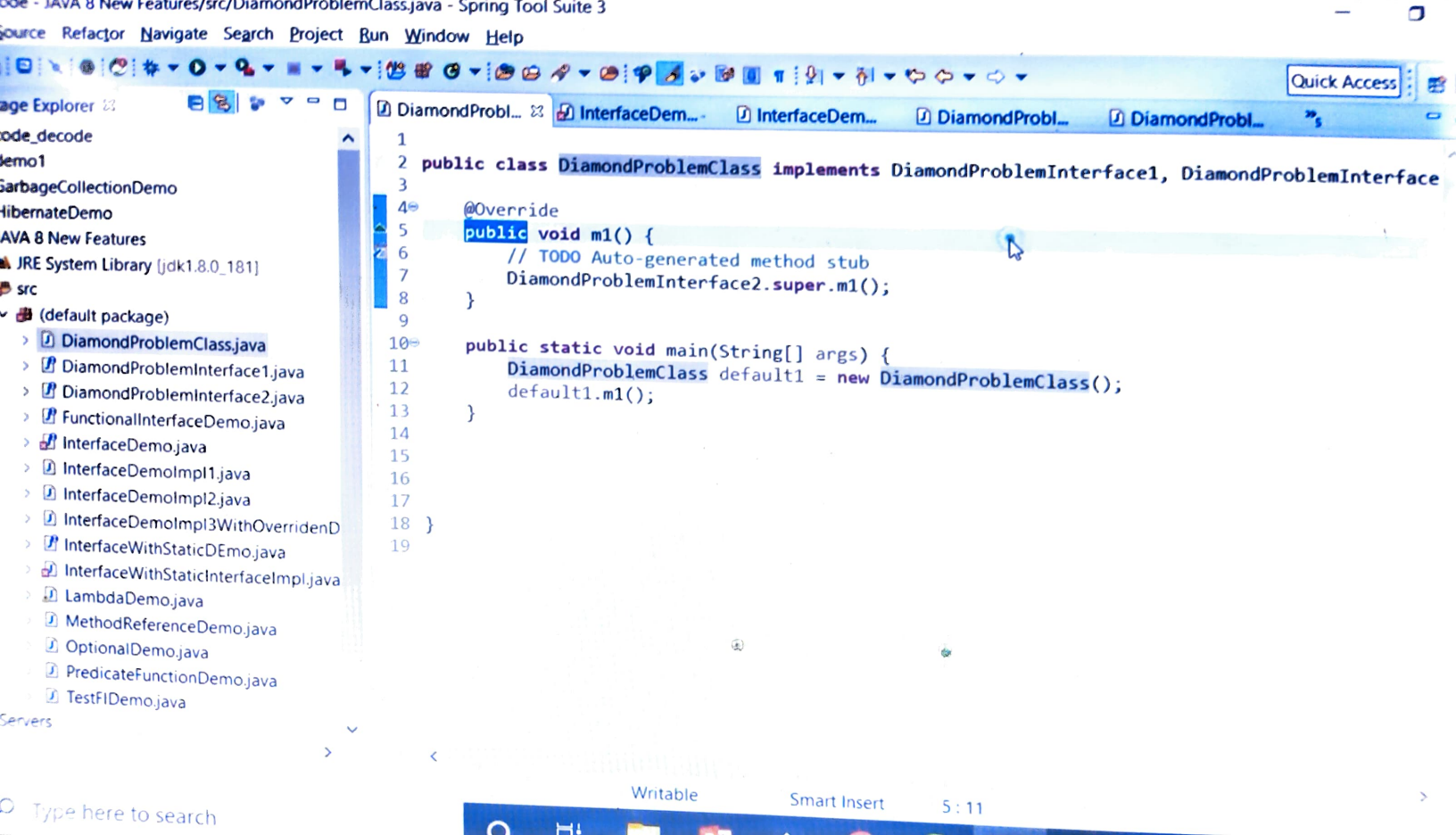
- You can override default method by keeping same method signature (name + arguments)
- Removing default keywords because in class default keyword is use in switch case to denote default case if no previous cases matched . **So you cant use Default keyword in Class**
- Adding Public as access modifier bcz in java 8, by default all methods are public so in child u cant reduce visibility of overridden default method.
- Giving our own implementation.

Q) Can you use hashcode() default implementation in Interface.

- You cannot give your default implementation of hash code() in interface for all implementing classes to use.
- We are not allowed to override Object class's method as default methods in interface else will get compile time error "".
- All implementing classes by default has access to all methods of object class.

Q) How Default methods in interface cope up with Diamond problem.

- Diamond problem of default methods of interface:
  - If 2 implemented interfaces contains same default methods then that's the diamond problem.
- In java, in such situation, The code will not compile.
- Solution to diamond problem :
  - Use InterfaceName.super.methodName();


- JAVA 8 New Features/src/DiamondProblemClass.java - Spring Tool Suite 3
Source Refactor Navigate Search Project Run Window Help
Quick Access
age Explorer
code_decode
demo1
GarbageCollectionDemo
HibernateDemo
AVA 8 New Features
JRE System Library [jdk1.8.0_181]
src
(default package)
DiamondProblemClass.java
DiamondProblemInterface1.java
DiamondProblemInterface2.java
FunctionalInterfaceDemo.java
InterfaceDemo.java
InterfaceDemoImpl1.java
InterfaceDemoImpl2.java
InterfaceDemoImpl3WithOverridenD
InterfaceWithStaticDEmo.java
InterfaceWithStaticInterfaceImpl.java
LambdaDemo.java
MethodReferenceDemo.java
OptionalDemo.java
PredicateFunctionDemo.java
TestFIDemo.java
Servers
DiamondProbl...
InterfaceDem...
InterfaceDem...
DiamondProbl...
DiamondProbl...
1
2 public class DiamondProblemClass implements DiamondProblemInterface1, DiamondProblemInterface
3
4 @Override
5 public void m1() {
6 // TODO Auto-generated method stub
7 DiamondProblemInterface2.super.m1();
8 }
9
10 public static void main(String[] args) {
11 DiamondProblemClass default1 = new DiamondProblemClass();
12 default1.m1();
13 }
14
15
16
17
18 }
19
Writable
Smart Insert
5 : 11
Type here to search

Q) Why Static methods were introduced in JAVA8

- Only reason for introducing static methods in interface is that you can call those methods with just interface name. No Need to create class and then its object.

- Since Interface can never contain :
  - Constructors,
  - Static blocks,
  - Nothings costly in terms of memory and performance.

- Hence we don't need to create object and hence if you have everything static, then for interface rather than class

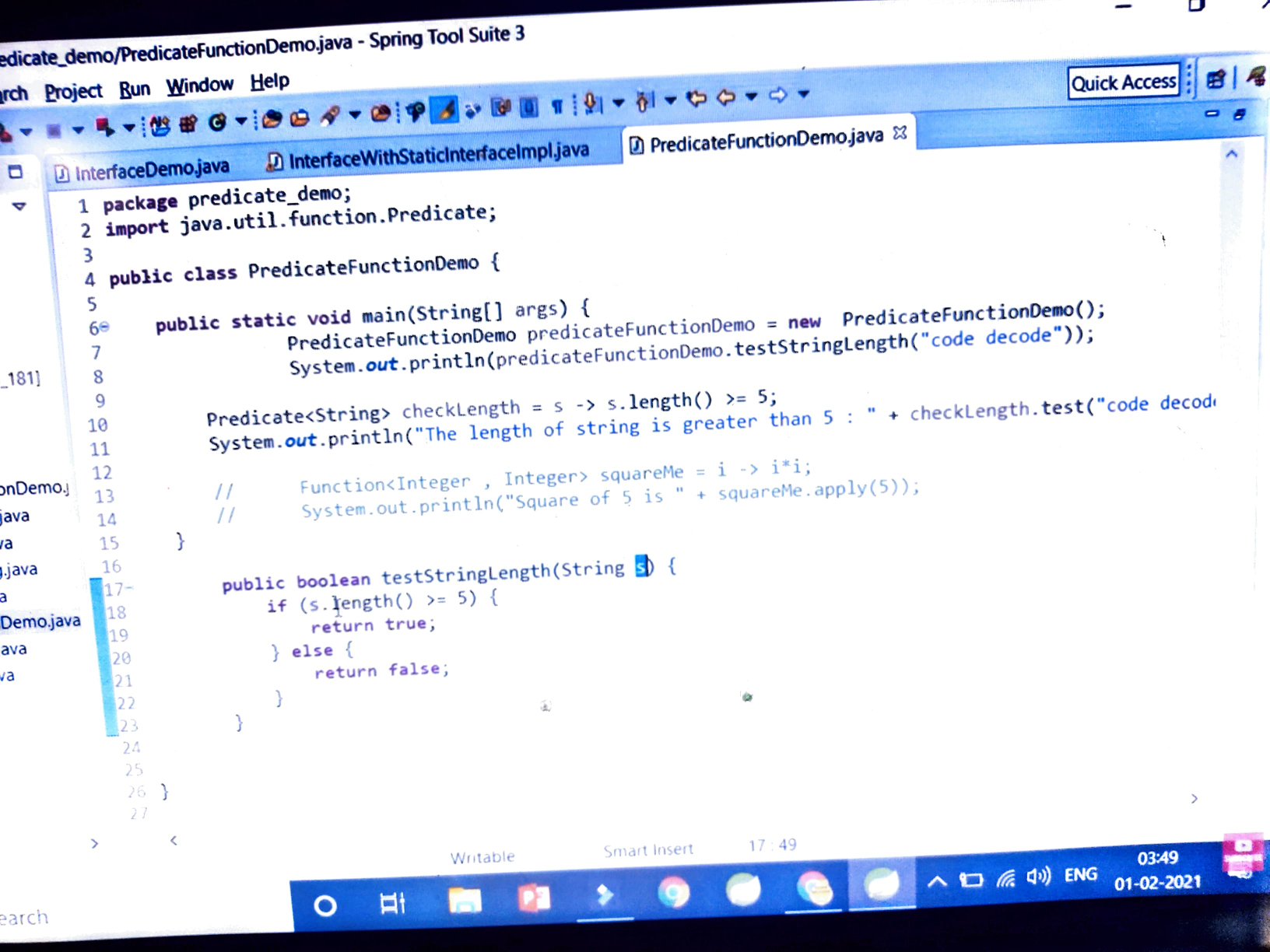
- (You have this flexibility only after java 8, before that you need to create class)

Q) Are Static Methods available to implementing classes by default?

- Static methods are not available to implementing classes .
- They are not default methods. They are static.
- Hence you can call these methods using Interface name explicitly from the implementing classes as implementing classes wont have access to these methods directly. (This is the disadvantage of static methods of interface that its not available to implementing classes.)

Q) What are Predicates

- Predicate is a predefined Functional Interface (Having only 1 abstract method).
- The only abstract method of predicate is test(T t):
  - public boolean test(T t);
- Whenever we want to check some boolean condition then you can go for Predicates.


edicate_demo/PredicateFunctionDemo.java - Spring Tool Suite 3
rch Project Run Window Help
Quick Access
InterfaceDemo.java
InterfaceWithStaticInterfaceImpl.java
PredicateFunctionDemo.java
1 package predicate_demo;
2 import java.util.function.Predicate;
3
4 public class PredicateFunctionDemo {
5
6 public static void main(String[] args) {
7 PredicateFunctionDemo predicateFunctionDemo = new PredicateFunctionDemo();
8 System.out.println(predicateFunctionDemo.testStringLength("code decode"));
9
10 Predicate<String> checkLength = s -> s.length() >= 5;
11 System.out.println("The length of string is greater than 5 : " + checkLength.test("code decod
12
13 // Function<Integer , Integer> squareMe = i -> i*i;
14 // System.out.println("Square of 5 is " + squareMe.apply(5));
15 }
16
17 public boolean testStringLength(String s) {
18 if (s.length() >= 5) {
19 return true;
20 } else {
21 return false;
22 }
23 }
24
25
26 }
27
Writable
Smart Insert
17 : 49
ENG
03:49
01-02-2021

Q) Type parameter and return types of Predicates?

- Input to predicate can be anything like
  - Predicate<String>
  - Predicate<Integer>
  - Predicate<Employee>

- Hence only 1 type argument is required which is input type in predicate.

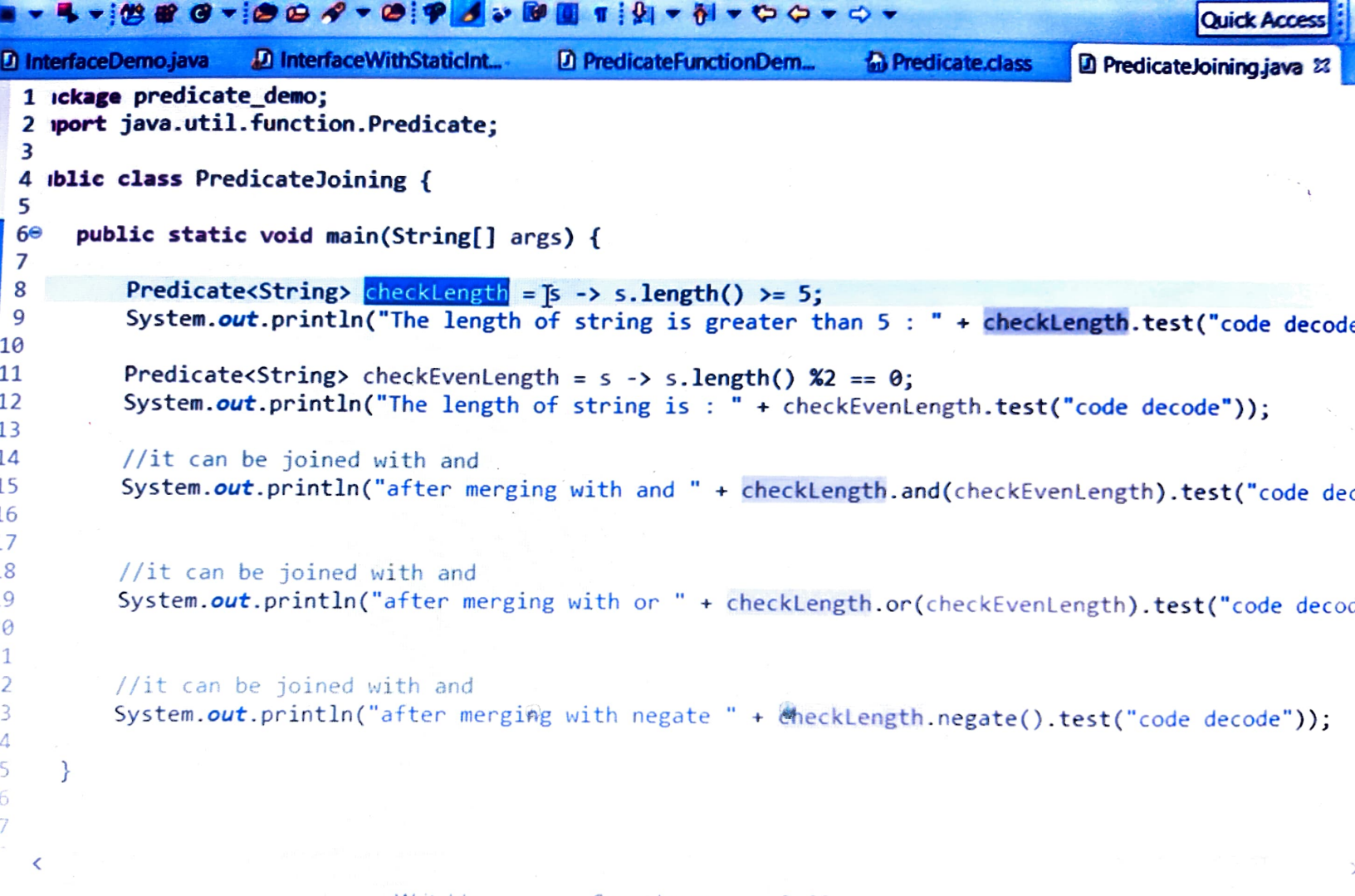
- **Return type is not required as its always Boolean only.**

Q) Advantages of Predicates?

- Code Reusability
- If you have same conditions being used 100 times in a prgram then you can write once and just use 100 times with **checkLength.test(different string to be tested)**.
- Conditional checks are holded by Functional interfaces.

Q) What is Predicate joining?

- You can combine predicates in serial predicate
- Three ways to join :
  - And
  - Or
  - Negate
- Eg if you want to test 2 conditions:
  - To check length of string > 5
  - To check if length is even.


Quick Access
InterfaceDemo.java
InterfaceWithStaticInt...
PredicateFunctionDem...
Predicate.class
PredicateJoining.java
ickage predicate_demo;
iport java.util.function.Predicate;
iblic class PredicateJoining {
public static void main(String[] args) {
Predicate<String> checkLength = s -> s.length() >= 5;
System.out.println("The length of string is greater than 5 : " + checkLength.test("code decode
Predicate<String> checkEvenLength = s -> s.length() %2 == 0;
System.out.println("The length of string is : " + checkEvenLength.test("code decode"));
//it can be joined with and
System.out.println("after merging with and " + checkLength.and(checkEvenLength).test("code dec
//it can be joined with and
System.out.println("after merging with or " + checkLength.or(checkEvenLength).test("code decod
//it can be joined with and
System.out.println("after merging with negate " + checkLength.negate().test("code decode"));
}
Writable
Smart Insert
8 : 38

Q) What are Functions

- Function is also a predefined Functional Interface (Having only 1 abstract method).
- The only abstract method of Function is apply(T t);
    - R apply(T t);
- Given some input perform some operation on input and then produce / return result (not necessary boolean value).
- This takes 1 input and returns one output.
- In predicate we used to take 1 input and return type is always boolean.
- In function return type is not fixed hence we declare both input type and return type.

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

public class FunctionDemo {

    public static void main(String[] args) {
//      FunctionDemo functionDemo = new FunctionDemo();
//      System.out.println(functionDemo.squreInt(5));

        Function<Integer , Integer> squareMe = i -> i*i;
        System.out.println("Square of 5 is " + squareMe.apply(5));
    }


//  public int squreInt(int i) {
//      int squred = i*i;
//      return squred;
//  }


}

```

Q) What is the difference between Predicate and Function?

Predicate

- It has the return type as Boolean. Its used for conditional checks
- It is written in the form of **Predicate< T>** which accepts a single argument.
- It contains test() method

Function

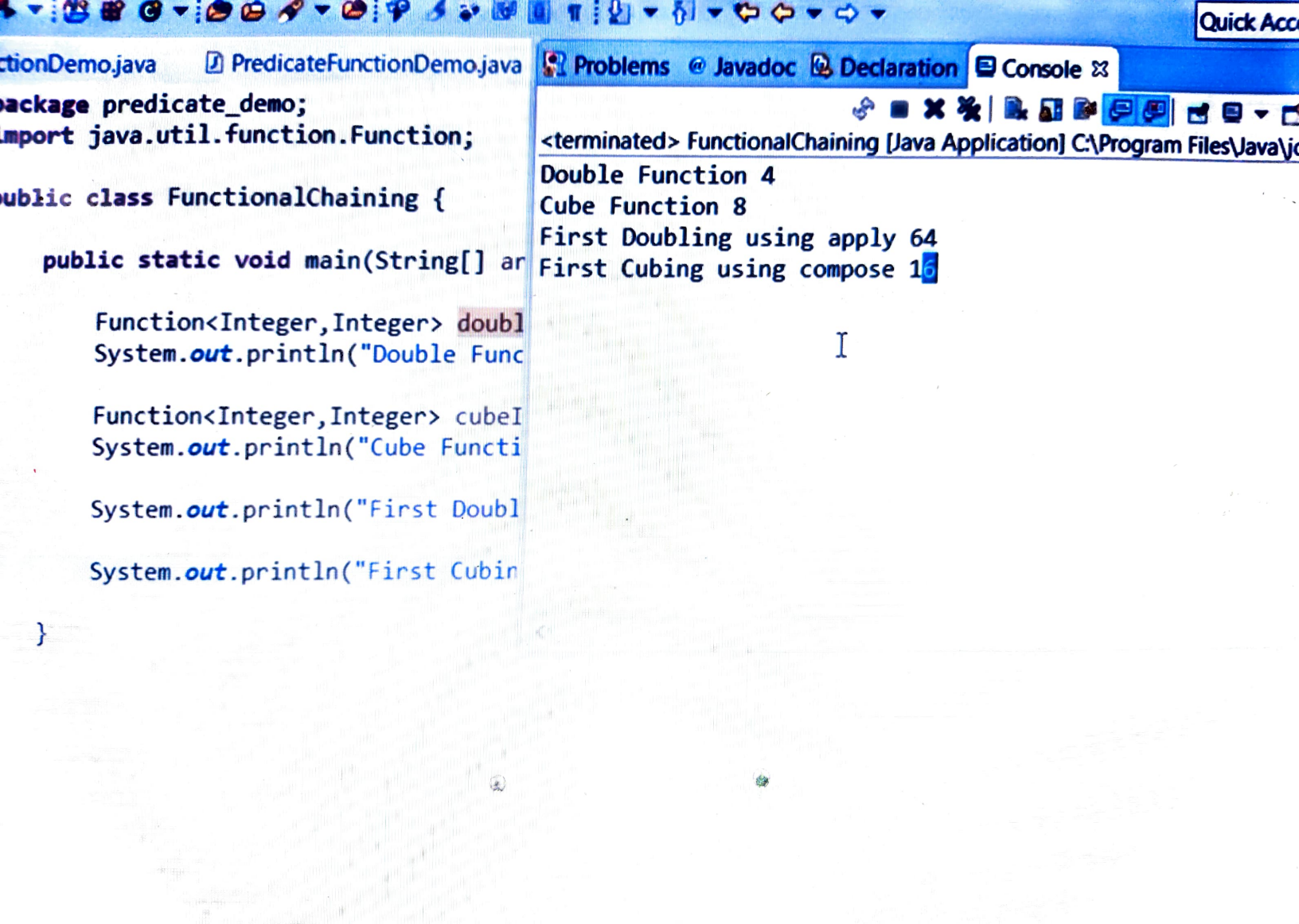
- It has the return type as Object. Its used to perform operations and return result.
- It is written in the form of **Function< T, R>** which also accepts a single argument but return Any type of object denoted by R.
- It contains apply() method

Q) What is Functional chaining

- We can combine / chain multiple functions together with **andThen** .
- There are two ways to combine functions:
  - f1.andThen(f2).apply(Input); - first f1 then f2
  - f1.compose(f2).apply(Input) - first f2 then f1

  - Multiple functions can be chained together like :
  - f1.andThen(f2).andThen(f3).andThen(f4).apply(Inputs);

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

public class FunctionalChaining {

    public static void main(String[] args) {

        Function<Integer,Integer> doubleIt = i -> 2*i;
        System.out.println("Double Function " +doubleIt.apply(2));

        Function<Integer,Integer> cubeIt = i -> i*i*i;
        System.out.println("Cube Function " + cubeIt.apply(2));

        System.out.println("First Doubling using apply " + doubleIt.andThen(cubeIt).apply(2));

        System.out.println("First Cubing using compose " + doubleIt.compose(cubeIt).apply(2));

    }


}
```


ctionDemo.java
PredicateFunctionDemo.java
package predicate_demo;
import java.util.function.Function;
public class FunctionalChaining {
public static void main(String[] ar
Function<Integer,Integer> doubl
System.out.println("Double Func
Function<Integer,Integer> cubeI
System.out.println("Cube Functi
System.out.println("First Doubl
System.out.println("First Cubin
}
Problems
Javadoc
Declaration
Console
<terminated> FunctionalChaining [Java Application] C:\Program Files\Java\jd
Double Function 4
Cube Function 8
First Doubling using apply 64
First Cubing using compose 16

Q) What is Consumer Functional interface?

- Predicate<T> takes 1 input and returns boolean.
- Function<T.R> takes 1 input and 1 return type produced after performing some operation on that input.

- **Consumer<T> →** It will consume Item. Consumers never return anything (never supply), they just consume.

- EG : take any object and save its details in Database and don't return anything.

```
Interface Consumer <T>
        public void accept(T t)
{
```

```java
package predicate_demo;
import java.util.function.Consumer;

public class ConsumerDemo {

    public static void main(String[] args) {
        ConsumerDemo consumerDemo = new  ConsumerDemo();
        consumerDemo.squreInt(5);

        Consumer<Integer> squareMe = i -> System.out.println("Taking an input and performing opera
        squareMe.accept(5);
    }

    public void squreInt(int i) {
        int squred = i*i;
        System.out.println("Squared number is "+ squred);
    }


}
```

Q) What is Supplier Functional interfac

- Supplier<R> → It will just supply required objects and will not take any input
- Its always going to supply never consume / take any input.

- EG : always supply me current date.

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

- No chaining as no input is given to this. Only it gives u output.

Q) Use of Biconsumer BiFunction, BiPredic and Why no BiSuppiler?

Till now we had :

- Predicate<T> → test() → return boolean
- Function<T, R> → apply() → returns anything
- Consumer<T> → accept() → returns nothing
- Supplier<R> → get() → returns anything

What if we need 2 arguments for operation?

Then we need Bi XYZ Functional Interfaces.

There is no input in supplier so no 1 or 2 Input arguments needed. Hence no BiSupplier.

Project Run Window Help

FunctionalC... Consumer.class ConsumerChai... SupplierDemo... Supplier.class BiPredicate

```
1  package predicate_demo;
2  import java.util.function.BiFunction;
4
5  public class BiPredicateBiFunctionDemo {
6
7      public static void main(String[] args) {
8
9          BiPredicate<Integer, Integer> checkSumOfTwo = (a,b) -> a+b >= 5;
10         System.out.println("sum of 2 and 5 is greater than 5  : " + checkSumOfTwo.test(2,5));
11         System.out.println("sum of 2 and 1 is greater than 5  : " + checkSumOfTwo.test(2,1));
12
13         BiFunction<Integer , Integer, Integer> multiplyBoth = (a,b) -> a*b;
14         System.out.println("Multiplication of 5 and 10 is  " + multiplyBoth.apply(5, 10));
15     }
16
17
18
19 }
```

Q) If we want to operate on 3 arguments th

- triPredicate?

- There are no TriPredicate or TriFunction etc.

No QuadPredicate No QuadFunction.

Java 8 has inbuilt Functional interfaces that can take only 1 or 2 arguments no more.