Functional Interfaces

- Functional Interfaces introduced in Java 8 allow us to use a lambda expression to initiate the interface's method and avoid using lengthy codes for the anonymous class implementation.
- Various built-in interfaces were declared with @FunctionalInterface annotation and made functional from Java 8.
- They are of 4 types, Function, Consumer, Predicate, and Supplier.

What is a functional Interface?

- In Java, everything revolves around class or Objects.
- No function is independently present on its own in java. They are part of classes or interfaces.
- to use them we require either the class or the object of the respective class to call that function.
- Functional Interface in Java enables users to implement functional programming in Java.
- In functional programming, the function is an independent entity.
- The Function can do anything a variable can. Like passing a function as a parameter, a function returned by another function, etc.
- A functional interface can contain only one abstract method and it can contain any number of static and default (non-abstract) methods.
  - **Abstract Method -** does not provide implementation, only declaration. And must be overridden by the class which implements the interfaces.
  - **Default Method -** Can provide implementation to a method in the interface and also can override the method and redefine it.
  - **Static Method -** Can provide implementation to a static method in the interface. Can call using the name of the interface preceding the method name. It can not be overridden by the class implementing an interface.
- e.g,

```
@FunctionalInterface
public interface Demo {
// abstract method
public void m1();

//n number of static methods and default method
public static void m2() {
```

```
            System.out.println("m2");
        }

        public static void m4() {
            System.out.println("m2");
        }

        public default void m3() {
            System.out.println("m3");
        }

    }
```

- **@FunctionalInterface Annotation -**
  - @FunctionalInterface Annotation is written above the interface declaration.
  - It effectively acts as a function thus, it can be passed as a parameter to a method or can be returned as a value by a method.
  - It is optional, but when mentioned java compiler ensures that the interface has only one abstract method.
- If we try to add more than one abstract method, the compiler flags an Unexpected @FunctionalInterface annotation message.
- To implement the abstract method of a functional interface in Java, we can either use lambda expression or we can implement the interface to our class and override the method
  - Implementing abstract method using Class

    e.g.,

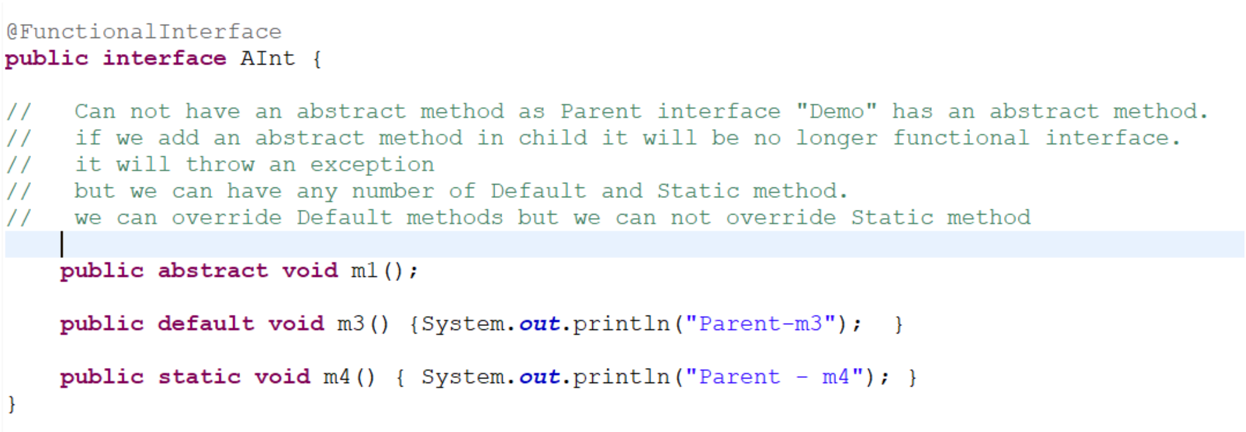
```
public class FunctionalInterfaceDemo {

    public static void main(String[] args) {
        Demo d = new Demo()
        {

        @Override
        public void m1() {
            System.out.println("implementing using class.");
        }};
        d.m1();
    }
}
```

  - Implementing using Lambda Expression

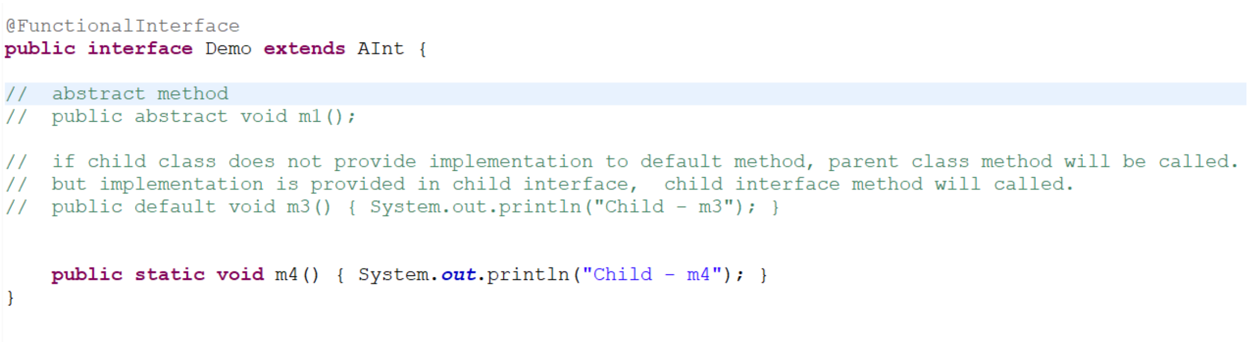
```
public class FunctionalInterfaceDemo {

    public static void main(String[] args) {
        Demo d = () -> {
            System.out.println("Using Lambda Expression");};
        d.m1();
    }
}
```

More Examples

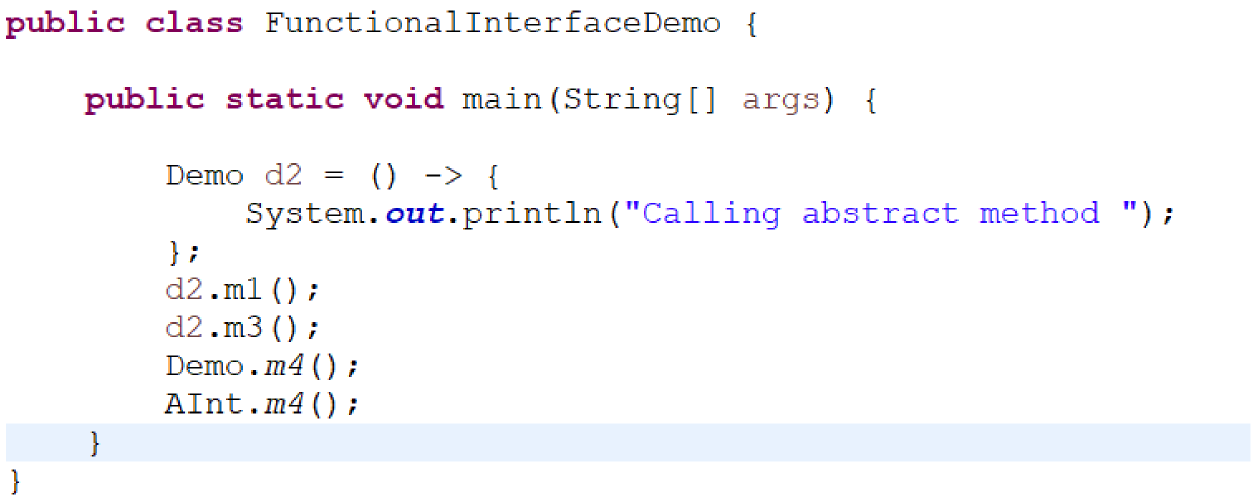
- Functional Interface Extending to a Non-Functional Interface.
  - The child interface inherits the methods of the parent interface.
  - The parent interface must not be functional as well as it should not have any abstract method.
  - The functional interface i.e. our child interface can have a single abstract method and multiple default and static methods.
  - The **child** and **parent interfaces can have** the **same abstract method**, **or the child interface can have no methods if both interfaces are Functional.**
  - e.g.,



```java
@FunctionalInterface
public interface AInt {

//   Can not have an abstract method as Parent interface "Demo" has an abstract method.
//   if we add an abstract method in child it will be no longer functional interface.
//   it will throw an exception
//   but we can have any number of Default and Static method.
//   we can override Default methods but we can not override Static method

    public abstract void m1();

    public default void m3() {System.out.println("Parent-m3");  }

    public static void m4() { System.out.println("Parent - m4"); }
}
```

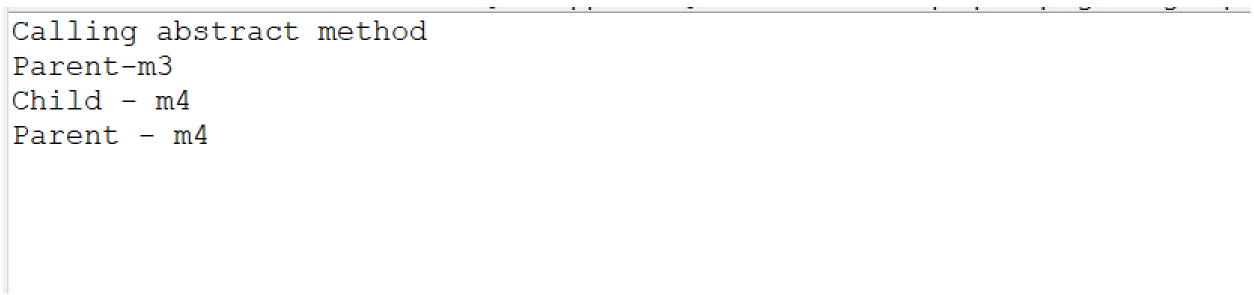


```java
@FunctionalInterface
public interface Demo extends AInt {

//  abstract method
//  public abstract void m1();

//  if child class does not provide implementation to default method, parent class method will be called.
//  but implementation is provided in child interface,  child interface method will called.
//  public default void m3() { System.out.println("Child - m3"); }


    public static void m4() { System.out.println("Child - m4"); }
}
```



```java
public class FunctionalInterfaceDemo {

    public static void main(String[] args) {

        Demo d2 = () -> {
            System.out.println("Calling abstract method ");
        };
        d2.m1();
        d2.m3();
        Demo.m4();
        AInt.m4();
    }
}
```

Output



```
Calling abstract method
Parent-m3
Child - m4
Parent - m4
```

Types of Functional interfaces

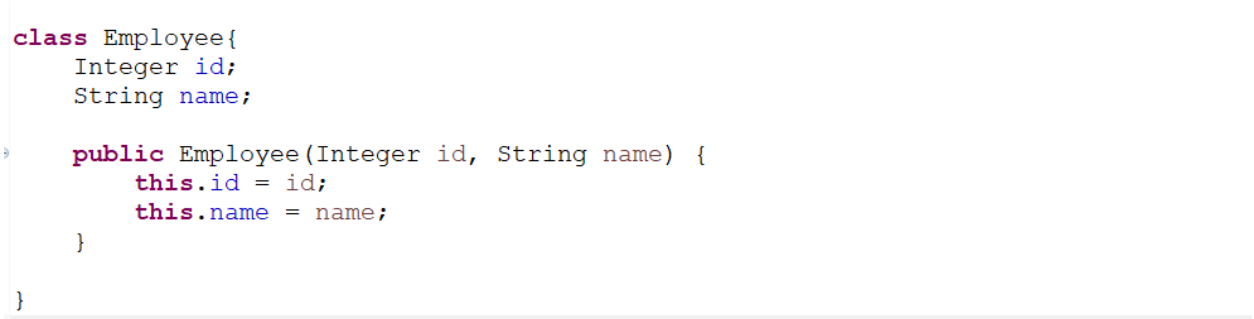
1. Function
2. Supplier
3. Consumer
4. Predicate

1. Function

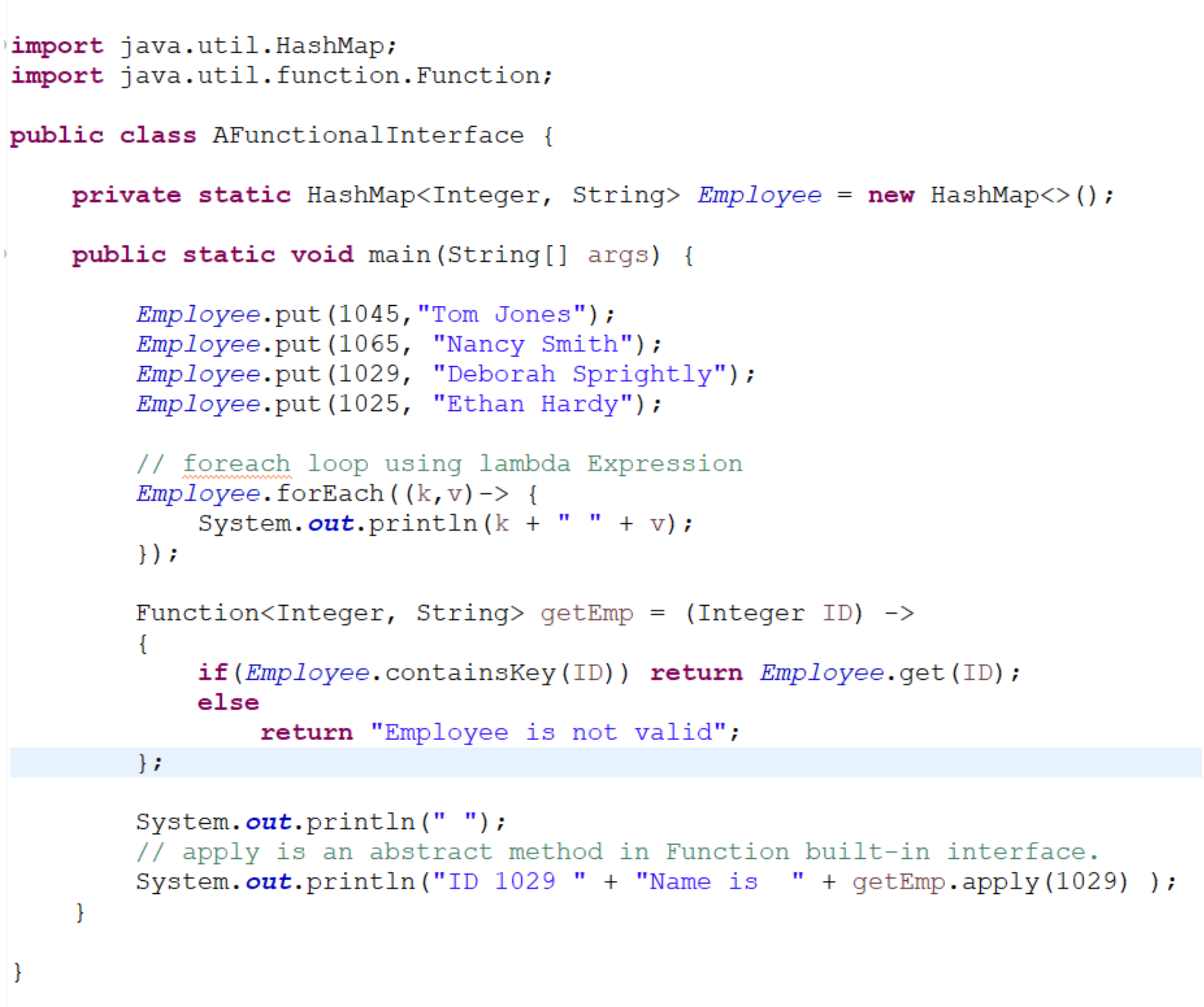
- **Receives a single argument**.
- **processes it, and returns a value.**
- e.g, Taking the key from the user as input and searching for the value in the map for the given key.
- Syntax

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

- e.g.,



```
class Employee{
    Integer id;
    String name;

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

}
```



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

public class AFunctionalInterface {

    private static HashMap<Integer, String> Employee = new HashMap<>();

    public static void main(String[] args) {

        Employee.put(1045,"Tom Jones");
        Employee.put(1065, "Nancy Smith");
        Employee.put(1029, "Deborah Sprightly");
        Employee.put(1025, "Ethan Hardy");

        // foreach loop using lambda Expression
        Employee.forEach((k,v)-> {
            System.out.println(k + " " + v);
        });

        Function<Integer, String> getEmp = (Integer ID) ->
        {
            if(Employee.containsKey(ID)) return Employee.get(ID);
            else
                return "Employee is not valid";
        };

        System.out.println(" ");
        // apply is an abstract method in Function built-in interface.
        System.out.println("ID 1029 " + "Name is  " + getEmp.apply(1029) );
    }

}
```

Bi-Function

- It is just like a Function but it takes two arguments.
- Two arguments are required in Bi-function.
- Just like a function it also returns a value.

- e.g., is UnaryOperator and BinaryOperator Interfaces

  UnaryOperator -

  - It **extends Function Interface**
  - It **takes one argument** and **returns a value. It should be the same as an argument.**

Binary Operator -

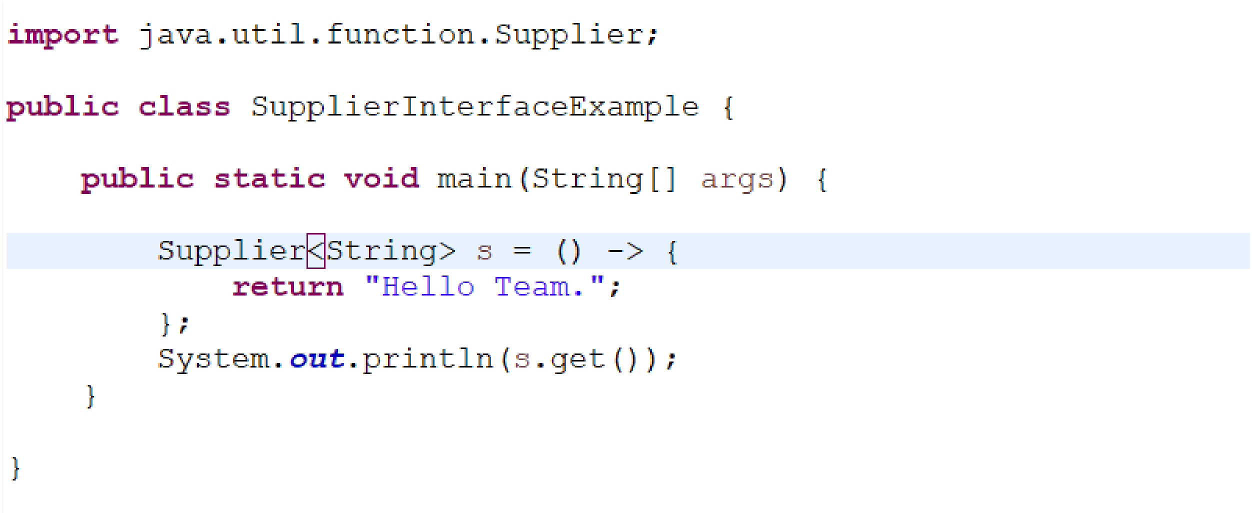
- Binary **takes two arguments** but **they must be of the same type.**
- **return value must be of the same type as the arguments.**

2. Supplier

- The supplier functional interface in Java is much like a functional interface.
- The only difference is **it doesn't take any arguments.**
- **It simply returns a value.**
- Syntax:

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

- e.g.,



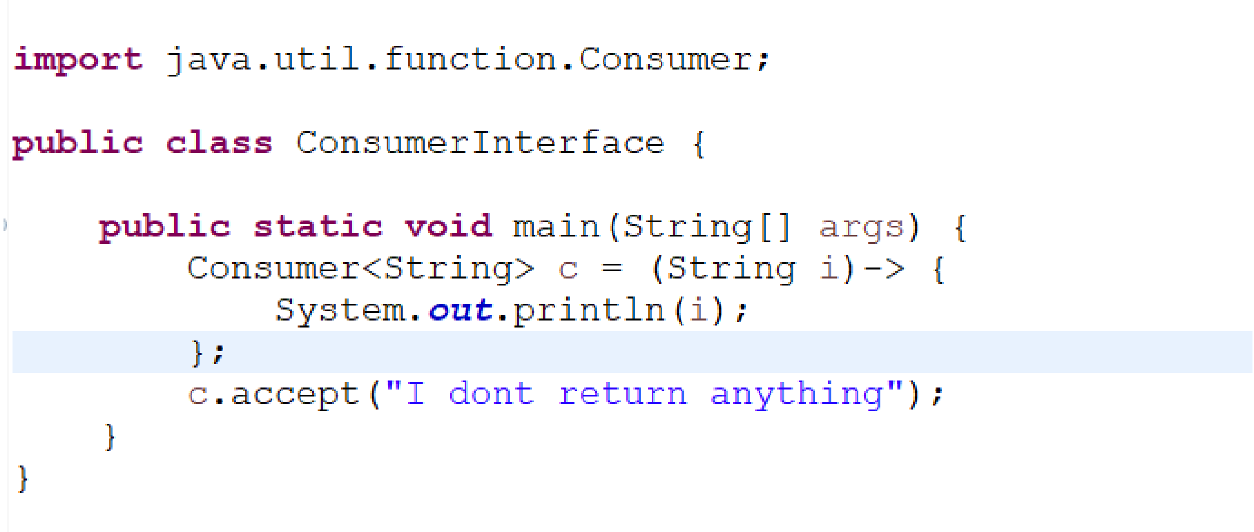
```
import java.util.function.Supplier;

public class SupplierInterfaceExample {

    public static void main(String[] args) {

        Supplier<String> s = () -> {
            return "Hello Team.";
        };
        System.out.println(s.get());
    }

}
```

3. Consumer

- The Consumer functional interface in Java **accepts a single gentrified argument**
- **But, it does not return any value.**
- Syntax

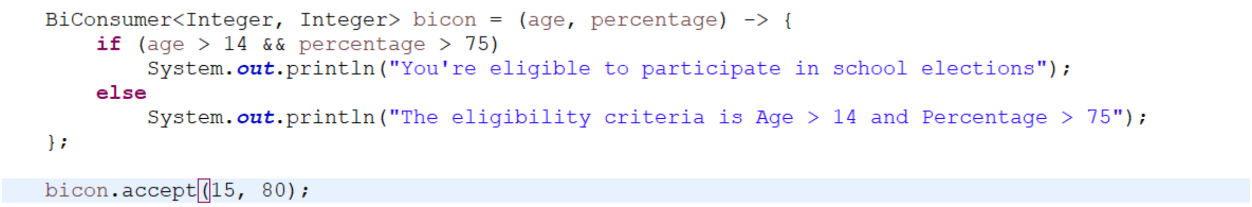
```
@FunctionalInterface
public interface Consumer<T>{
   void accept(T t);
}
```

- accept is the abstract method of the Consumer.
- e.g.,



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

public class ConsumerInterface {

    public static void main(String[] args) {
        Consumer<String> c = (String i)-> {
            System.out.println(i);
        };
        c.accept("I dont return anything");
    }
}
```

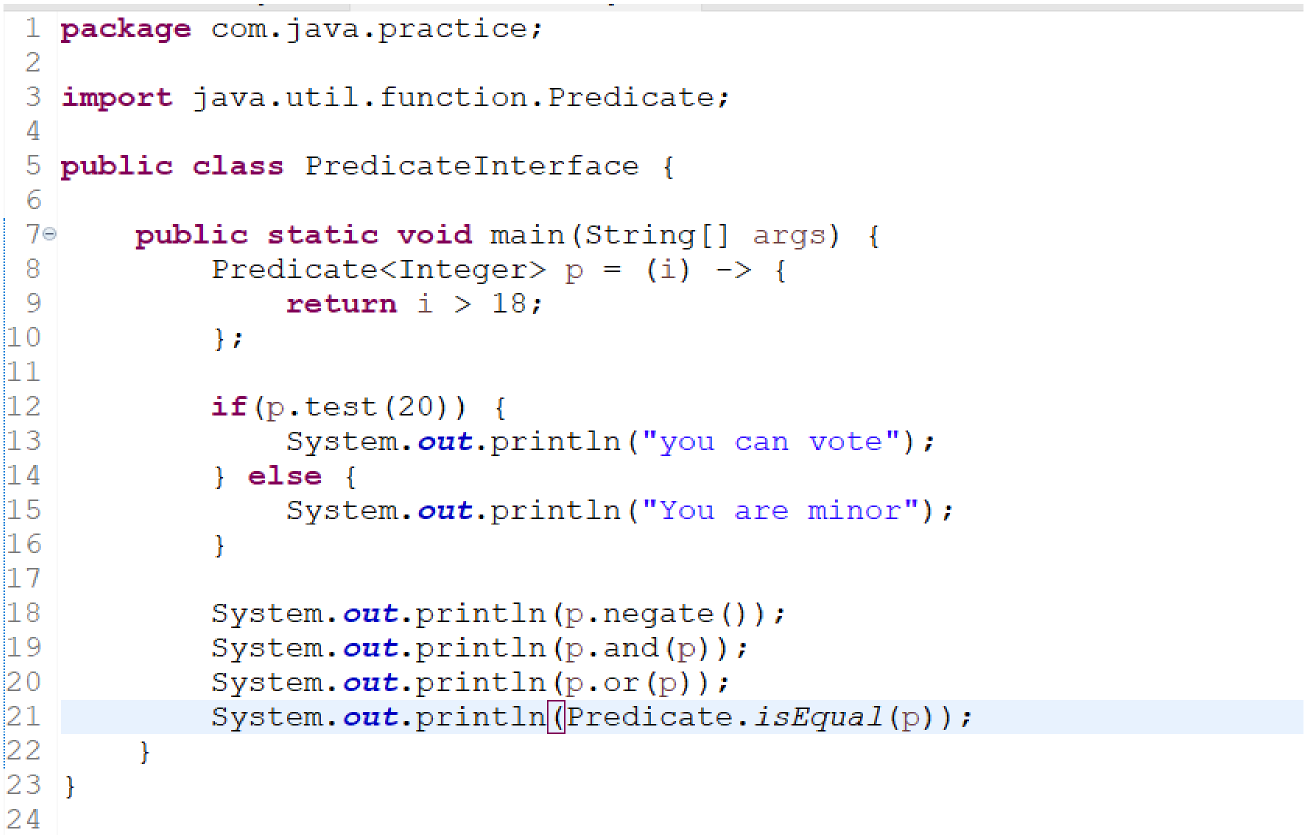
- **BiConsumer**
  - It takes two arguments, one generic, and the other of primitive type.
  - It also doesn't return a value.



```java
BiConsumer<Integer, Integer> bicon = (age, percentage) -> {
    if (age > 14 && percentage > 75)
        System.out.println("You're eligible to participate in school elections");
    else
        System.out.println("The eligibility criteria is Age > 14 and Percentage > 75");
};

bicon.accept(15, 80);
```

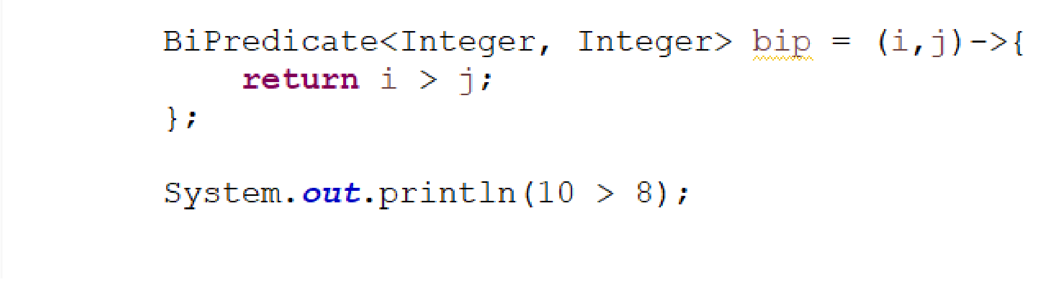
4. Predicate

- **Takes a single argument and returns a boolean value.**
- It is usually used in filtering values from the collection.
- Predicate interface has one **abstract** method **test()**
- 3 **default** methods **and()**,**negate()**, and **or()**.
- 1 **static** method **isEqual()**.



```java
package com.java.practice;

import java.util.function.Predicate;

public class PredicateInterface {

    public static void main(String[] args) {
        Predicate<Integer> p = (i) -> {
            return i > 18;
        };

        if(p.test(20)) {
            System.out.println("you can vote");
        } else {
            System.out.println("You are minor");
        }

        System.out.println(p.negate());
        System.out.println(p.and(p));
        System.out.println(p.or(p));
        System.out.println(Predicate.isEqual(p));
    }
}

```

- Bi-Predicate
    - It takes two arguments.
    - Returns boolean value
    - e.g.,



```java
BiPredicate<Integer, Integer> bip = (i,j)->{
    return i > j;
};

System.out.println(10 > 8);
```