Lambda Expressions , Functional Interface, Method Reference

AMOL R PATIL - 9822291613

Static and Default Methods in Interface

- Before Java 8, interfaces could have only public abstract methods. It was not possible to add new functionality to the existing interface without forcing all implementing classes to create an implementation of the new methods, nor was it possible to create interface methods with an implementation.

- In Java 8, the interfaces can have static and default methods.

Use of static Method in an Interface

- Utility Methods: Static methods in interfaces can be used to provide utility methods related to the interface's purpose. These methods are typically independent of any instance and are associated with the interface itself.

- Code Organization: Static methods in interfaces can be used to organize related utility methods in a single interface. This can improve code organization and make it more modular.

- Enhancing Existing Interfaces: Static methods allow the addition of new methods to interfaces without breaking existing implementations. This is particularly useful when you want to introduce new functionality in interfaces without forcing all implementing classes to provide a default implementation.

- Factory Methods: Static methods in interfaces can be used as factory methods to create instances of the interface or its implementing classes.

Use of default Method in an Interface

- Backward Compatibility: When new methods need to be added to an existing interface, introducing default methods allows the new methods to have a default implementation. This ensures that existing classes implementing the interface are not required to provide an implementation for the new methods, maintaining backward compatibility.

- Enhancing Interfaces: Default methods provide a way to enhance interfaces by adding new functionality without forcing implementing classes to update their code. This is especially useful when the interface is widely used, and it's desirable to avoid breaking existing code.

- Providing Optional Functionality: Default methods can be used to provide optional functionality in interfaces. Implementing classes can choose to use the default implementation or provide their own.

It's worth noting that the introduction of default methods was a significant feature in Java 8 and played a crucial role in enabling the addition of lambda expressions and the Stream API to the language. Default methods help strike a balance between adding new features to interfaces and maintaining compatibility with existing code.

Functional Interface

- An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method.
- Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces
- Runnable, ActionListener, and Comparable are some of the examples of functional interfaces.
- Functional interfaces are used and executed by representing the interface with an annotation called @FunctionalInterface.

- The main use of functional interfaces is to enable the implementation of lambda expressions, which provide a concise way to express instances of single-method interfaces.

- Lambda Expressions: Functional interfaces are often used in conjunction with lambda expressions to provide a more concise syntax for writing anonymous classes. Lambda expressions allow you to express instances of functional interfaces more compactly, improving code readability and reducing boilerplate code.

- Default Methods: Functional interfaces can have default methods (methods with an implementation) as well. This allows you to add new methods to interfaces without breaking the existing implementations.

```
@FunctionalInterface
interface MyFunctionalInterface {
    void myAbstractMethod();

    // Default method
    default void myDefaultMethod() {
        System.out.println("Default implementation");
    }
}
```

Functional Programming: Functional interfaces support functional programming paradigms by allowing the use of lambda expressions, which are a concise way to express behavior. This can lead to more modular and expressive code, especially when dealing with functional-style operations on collections using streams.

Functional Interface Annotation: The @FunctionalInterface annotation is optional but can be used to ensure that an interface is intended to be a functional interface. The compiler will generate an error if an interface annotated with @FunctionalInterface has more than one abstract method.

```
@FunctionalInterface
interface MyFunctionalInterface {
    void myAbstractMethod();

    // Error: @FunctionalInterface should have only one abstract method
    void anotherAbstractMethod();
}
```

Functional Programming

- Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It emphasizes the use of functions that have no side effects and are treated as first-class citizens. Here are some key principles and characteristics of functional programming:

- Pure Functions: Functions in functional programming are treated as mathematical functions, meaning that their output is solely determined by their input parameters, and they have no side effects. Pure functions make code more predictable, testable, and easier to reason about.

- Immutability: Data in functional programming is immutable, meaning once a value is assigned, it cannot be changed. Instead of modifying existing data, functional programs create new data structures with the desired values. This helps avoid unintended side effects and makes code more robust.

- First-Class and Higher-Order Functions: Functions in functional programming are first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, and returned as values from other functions. Higher-order functions take one or more functions as arguments or return a function as a result.

- No Mutable State: Functional programming discourages the use of mutable state and encourages the use of immutable data structures. This reduces the risk of bugs related to state changes and makes it easier to reason about program behavior.

- Referential Transparency: An expression is referentially transparent if its result is the same for the same inputs in all contexts. This property enables optimizations and makes it easier to understand and reason about the behavior of code.

- Lazy Evaluation: Lazy evaluation delays the evaluation of an expression until its value is actually needed. This can lead to more efficient use of resources and allows for the creation of infinite data structures.

- Pattern Matching: Functional programming languages often include pattern matching, a mechanism for checking a value against a pattern. This can simplify code by allowing concise and expressive handling of different cases.

- Recursion: Recursion is a common technique in functional programming for solving problems by breaking them down into smaller sub-problems. It replaces loops in imperative programming.

- Declarative Style: Functional programming tends to be more declarative than imperative. Instead of specifying step-by-step instructions for the computer to follow, developers focus on defining what the program should accomplish.

- Functional programming languages include Haskell, Lisp, Scala, Clojure, and others. While many modern programming languages, including Java, JavaScript, and Python, support functional programming features, they are often used in conjunction with other paradigms. The adoption of functional programming concepts has been growing as developers recognize the benefits of writing more maintainable, scalable, and bug-resistant code.

Lambda Expression

- Lambda Expression is an anonymous (nameless) function. In other words, the function which doesn't have the name, return type and access modifiers. Lambda Expression is also known as anonymous functions or closures.

- The primary Objective of introducing Lambda Expression is to promote the benefits of functional programming in Java

```
(parameters) -> expression

(parameters) -> { statements; }

() -> expression
```

Lambda Expression Advantages

- We can reduce length of the code so that readability of the code will be improved.
- We can resolve complexity of anonymous inner classes.
- We can provide Lambda expression in the place of object.
- We can pass lambda expression as argument to methods

Lambda Expression Variables

- Whatever the variables declare inside lambda expression are simply acts as local variables
- Within lambda expression ‘this” keyword represents current outer class object reference (that is current enclosing class reference in which we declare lambda expression
- 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

Lambda Function Ex’s

- Implement Sorting using Comparable and Comparator with different Collections

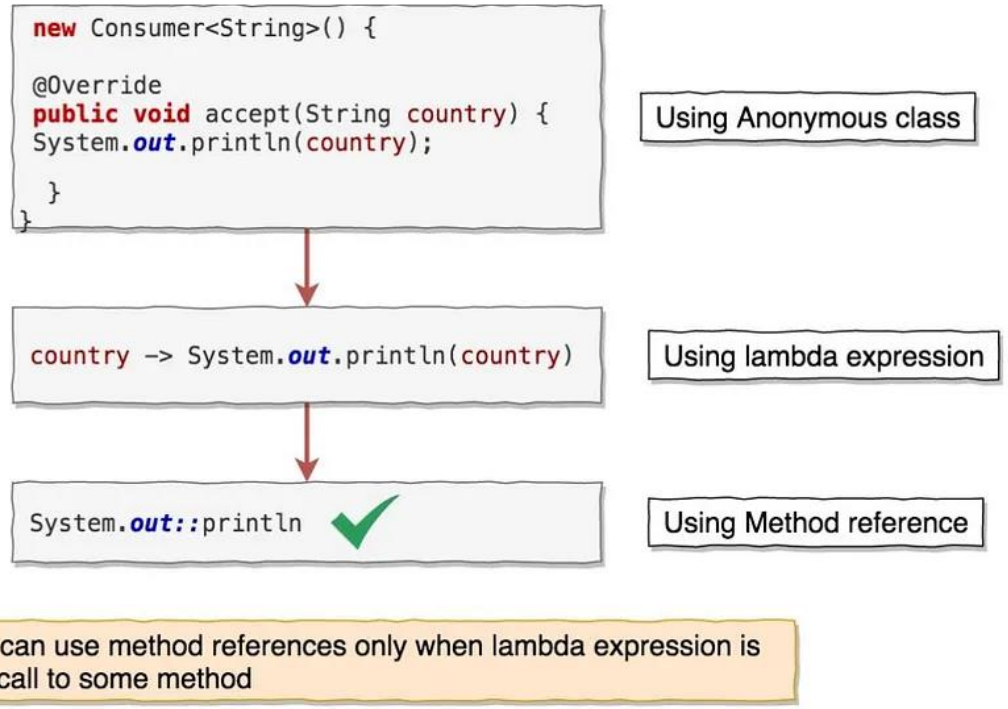
  Using List Comparable / Comparator [using lambda expression]

  Using Set

  Using Map

Method Reference in Java 8

- Method references provide a shorthand notation for lambda expressions that only call a single method.
- Method references can be used to simplify the code and make it more readable



- Types of method references
- Reference to static method
- Reference to instance method of object type
- Reference to instance method of existing object
- Reference constructor

Built in Functional Interfaces

- Java SE 8 provides a rich set of 43 functional interfaces
- All these interfaces are included under package java.util.function
- This set of interfaces can be utilized to implement lambda expressions
- In Java 8, there are 4 main functional interfaces are introduced which could be used in different scenarios. These are given below.
    - Consumer
    - Predicate
    - Function
    - Supplier

Consumer Interface

A Consumer is an in-build functional interface in the java.util.function package. we use consumers when we need to consume objects, the consumer takes an input value and returns nothing. The consumer interface has two methods.

```
void accept(T value);
default Consumer<T> andThen(Consumer<? super T> after);
```

andThen() Method

This method is used to chain multiple Consumer interface implementations one after another. The method returns a composed consumer of different implementations defined in the order. If the evaluation of any Consumer implementation along the chain throws an exception, it is relayed to the caller of the composed function.

Supplier Interface

- Supplier won't take any input and it will always supply objects. Supplier
- Functional Interface contains only one method get().

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

Supplier Functional interface does not contain any default and static methods

Predicate Interface

- Predicate functional interface of Java is a type of function that accepts a single value or argument and does some sort of processing on it and returns a boolean (True/ False) answer.

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

- Predicate interface present in java.util.function package.
- It’s a functional interface and it contains only one method i.e., test()
- As predicate is a functional interface and hence it can refer lambda expression

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

- 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

Function Interface

A Function is another in-build functional interface in java.util.function package, the function takes an input value and returns a value. The function interface has four methods, mostly function used in map feature of stream APIs.

```
R apply(T var1);
default <V> Function<V, R> compose(Function<V, T> before);
default <V> Function<T, V> andThen(Function<R, V> after);
static <T> Function<T, T> identity();
```

Function Chaining

We can combine multiple functions together 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

Predicate Ex’s

- WA Program to display names starts with 'K' by using Predicate:
- Predicate Example to remove null values and empty strings from the given list:
- Program for User Authentication by using Predicate:
- Given an ArrayList containing 10 words, write a program to filter the words which are palindrome, with the help of Predicate.

Consumer Interface

- The Consumer interface accepts one argument, but there is no return value.
- The name of the function inside this interface is accept.

```
Consumer<String> consumer = (s) -> System.out.println(s.toUpperCase());
consumer.accept("Hello World");
```

- There are also functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer. These variants accept primitive values as arguments.
- 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.

Consumer I/F Ex's.

- Display movie information by using Consumer
- Given an ArrayList containing 10 words, write a program to reverse each word and update the same ArrayList , with the help of Consumer.
- Given an ArrayList containing 10 numbers, write a program using Consumer methods to display each number and whether is it odd or even.
  Example: For number 2, it should print "2 even" For number 5, it should print "5 odd"

Supplier I/F Ex's.

- Generate random name using Supplier interface
- Generate 6 digit OTP using Supplier interface
- Write a program using Supplier, which generates and returns an ArrayList containing first 10 prime numbers.

Function Ex's

- Write a function to find length of String

- Program to remove spaces present in the given String by using Function:

- Program to perform Salary Increment for Employees by using Predicate & Function

- Create class Student
  Check if marks > 60 Using Predicate
  Generate Grade Using Function
  and Print Student Data Using Consumer

Given an ArrayList with 5 Employee(id,name,location,salary) objects, write a program to extract the location details of each Employee and store it in an ArrayList, with the help of Function..

Lambda Function Ex’s

- Create an ArrayList al and add 25 random numbers.
  Write a code to print all the prime numbers that are present in it, using lambda expression.

- Create an ArrayList al and add 10 different words.
  Write a code to print all the Strings in reverse order, using lambda expression.

- Create an ArrayList and add 10 different words.
  Write a code to print all the Strings whose length is odd, using lambda expression.

- Create an interface Wordcount with a single abstract method int count(String str), to count and return the no of words in the given String. Implement count method using Lambda expression in another class MyClassWithLambda. Invoke it to display the result on the console.

Function Ex’s

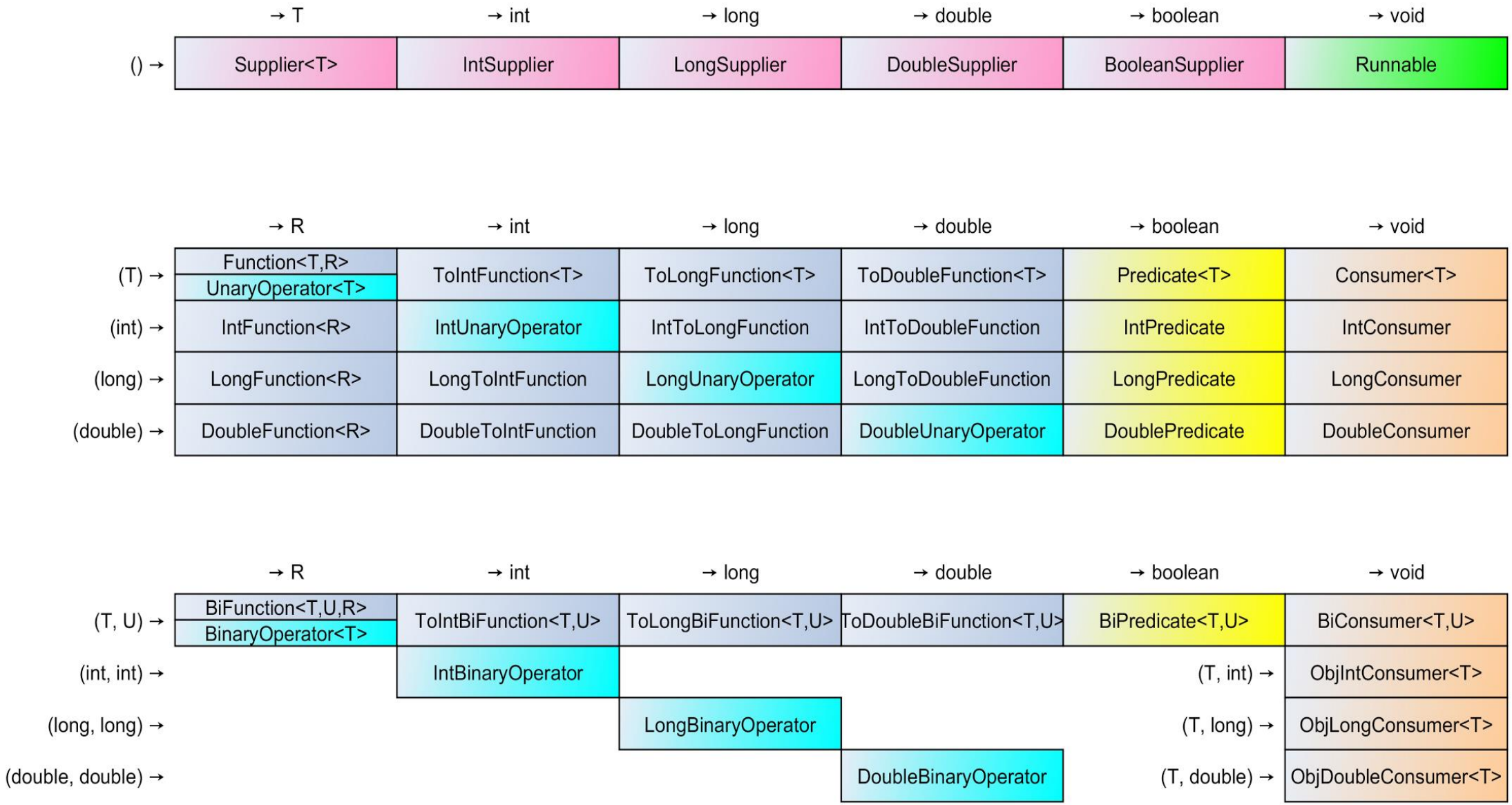
- Given an ArrayList containing 10 numbers, write a program to filter the perfect square numbers.

Example for perfect square numbers: 0, 1, 4, 9, 16, 25, 36, 49, 64 etc..

- Given an ArrayList containing 10 numbers, write a program to calculate the sum of all the elements, with the help of Function

Comparison Table of Predicate, Function, Consumer and Supplier

Property	Predicate	Function	Consumer	Supplier
1) Purpose	To take some Input and perform some conditional checks	To take some Input and perform required Operation and return the result	To consume some Input and perform required Operation. It won't return anything.	To supply some Value base on our Requirement.
2) Interface Declaration	interface Predicate <T> { :::::::::::: }	interface Function <T, R> { :::::::::::: }	interface Consumer <T> { :::::::::::: }	interface Supplier <R> { :::::::::::: }
3) Signle Abstract Method (SAM)	public boolean test (T t);	public R apply (T t);	public void accept (T t);	public R get();
4) Default Methods	and(), or(), negate()	andThen(), compose()	andThen()	-
5) Static Method	isEqual()	identify()	-	-



Java 8 Functional Interface Naming Guide
by Esko Luontola, www.orfjackal.net

Functional Interface	Primitive Versions
Predicate<T>	IntPredicate LongPredicate DoublePredicate
Consumer<T>	IntConsumer LongConsumer DoubleConsumer
Function<T, R>	IntFunction<R> IntToDoubleFunction IntToLongFunction LongFunction<R> LongToDoubleFunction LongToIntFunction DoubleFunction<R> DoubleToIntFunction DoubleToLongFunction ToIntFunction<T> ToDoubleFunction<T> ToLongFunction<T>
Supplier<T>	BooleanSupplier IntSupplier LongSupplier DoubleSupplier
UnaryOperator<T>	IntUnaryOperator LongUnaryOperator DoubleUnaryOperator

Classroom Assignement

- Employee Management

  Employee name designation salary city

  Find all managers
  Find employees from Pune city
  Find employees whose salary is < 20000
  Find all managers from Banglore city
  Find all employees with salary < 20000 or from Banglore city
  Find all employees information who are not managers

Method Ref

Java provides a new feature called method reference in Java 8.

Method reference is used to refer method of the functional interface.

It is a compact and easy form of a lambda expression.

Each time when you are using a lambda expression to just referring a method, you can replace your lambda expression with a method reference.

```
list.forEach(s -> System.out.println(s));

list.forEach(System.out::println);
```

Method Ref

Define your own class with an instance method "int factorial(int n)" which should return the factorial of the given input "n".
Define your own functional interface to refer this instance method and invoke it to get the factorial result.

Define your own class with a static method "int digitCount(int n)" which should return the number of digits in a given input "n".
Define your own functional interface to refer this static method and invoke it to get the number of digits.

Define your own class and a parameterized constructor with one integer argument. It should check the argument and display "Prime" or "Not Prime".
Define your own functional interface to refer this constructor and invoke it to check whether the given number is Prime or Not.

http://www.java2s.com/Tutorials/Java/java.util.function/BinaryOperator/1020__BinaryOperator.minBy.htm