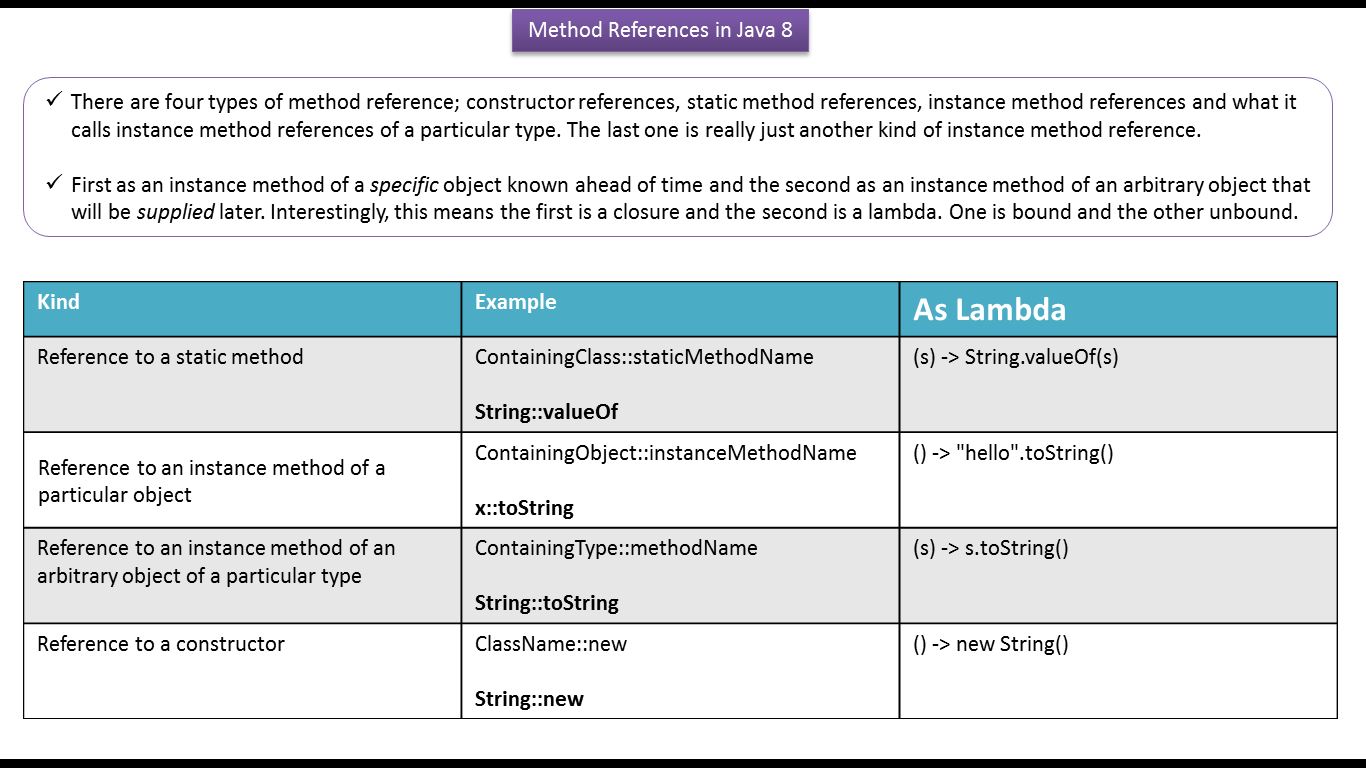
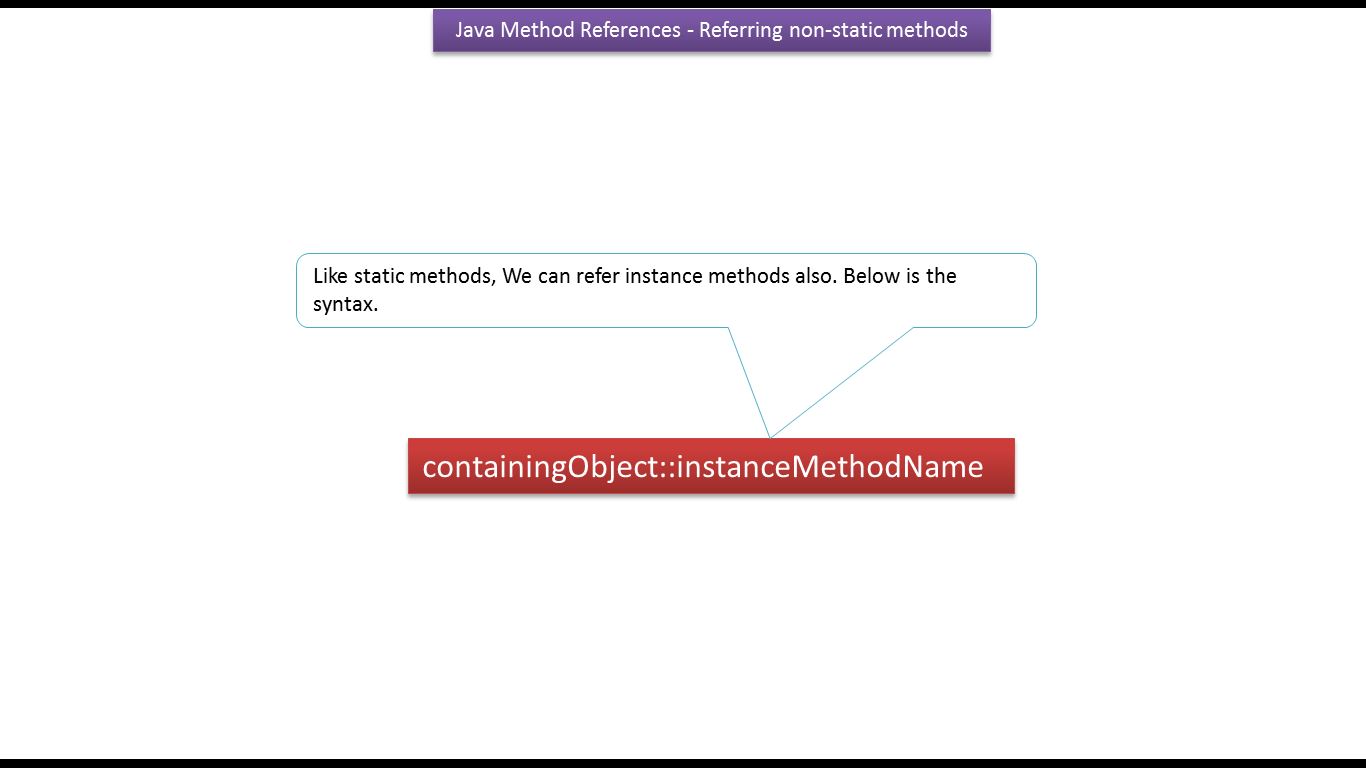
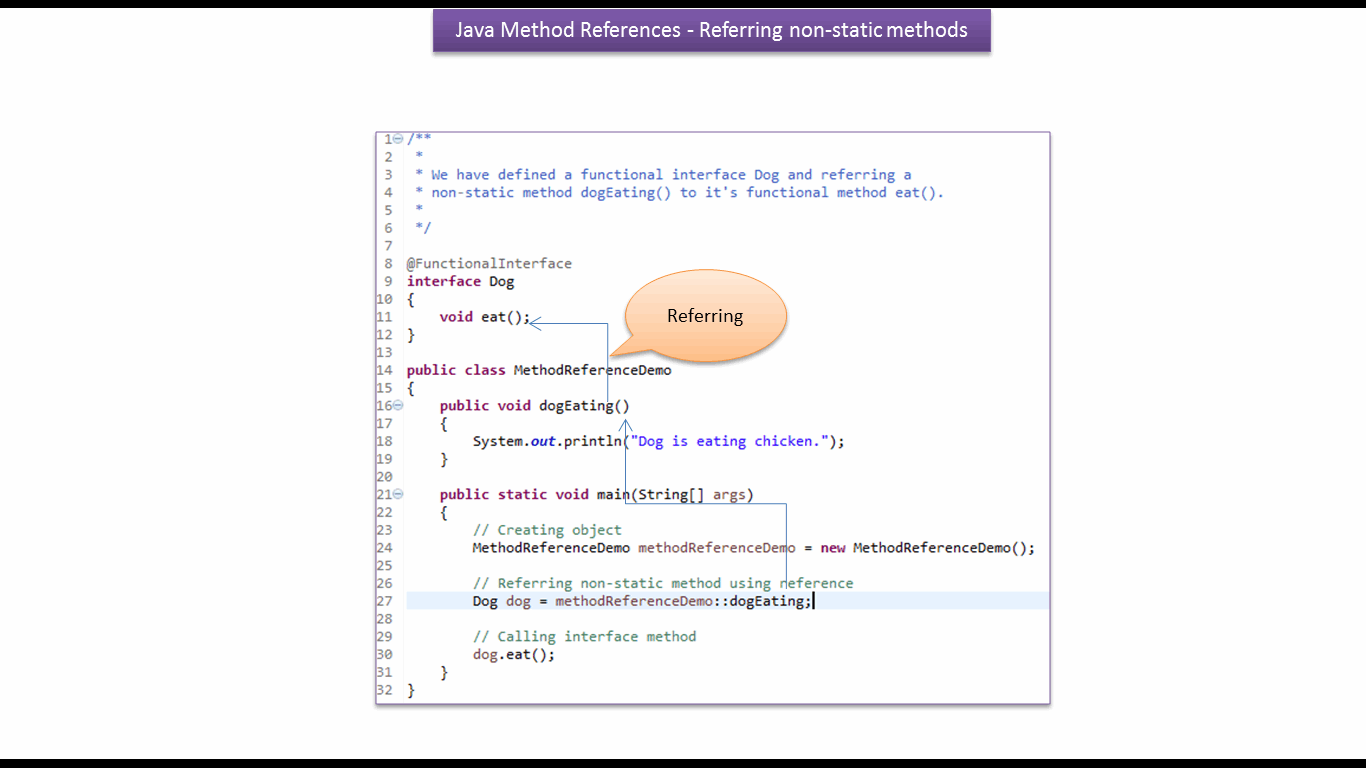
****

**Non-Static Method reference - Dog | Method reference in Java 8 | Java method reference**

****

**MethodReferenceDemo.java**

**/\*\***

**\***

**\* We have defined a functional interface Dog and referring a**

**\* non-static method dogEating() to it's functional method eat().**

**\***

**\*/**

**@FunctionalInterface**

**interface Dog**

**{**

**void eat();**

**}**

**public class MethodReferenceDemo**

**{**

**public void dogEating()**

**{**

**System.out.println("Dog is eating chicken.");**

**}**

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

**{**

**// Creating object**

**MethodReferenceDemo methodReferenceDemo = new MethodReferenceDemo();**

**// Referring non-static method using reference**

**Dog dog = methodReferenceDemo::dogEating;**

**// Calling interface method**

**dog.eat();**

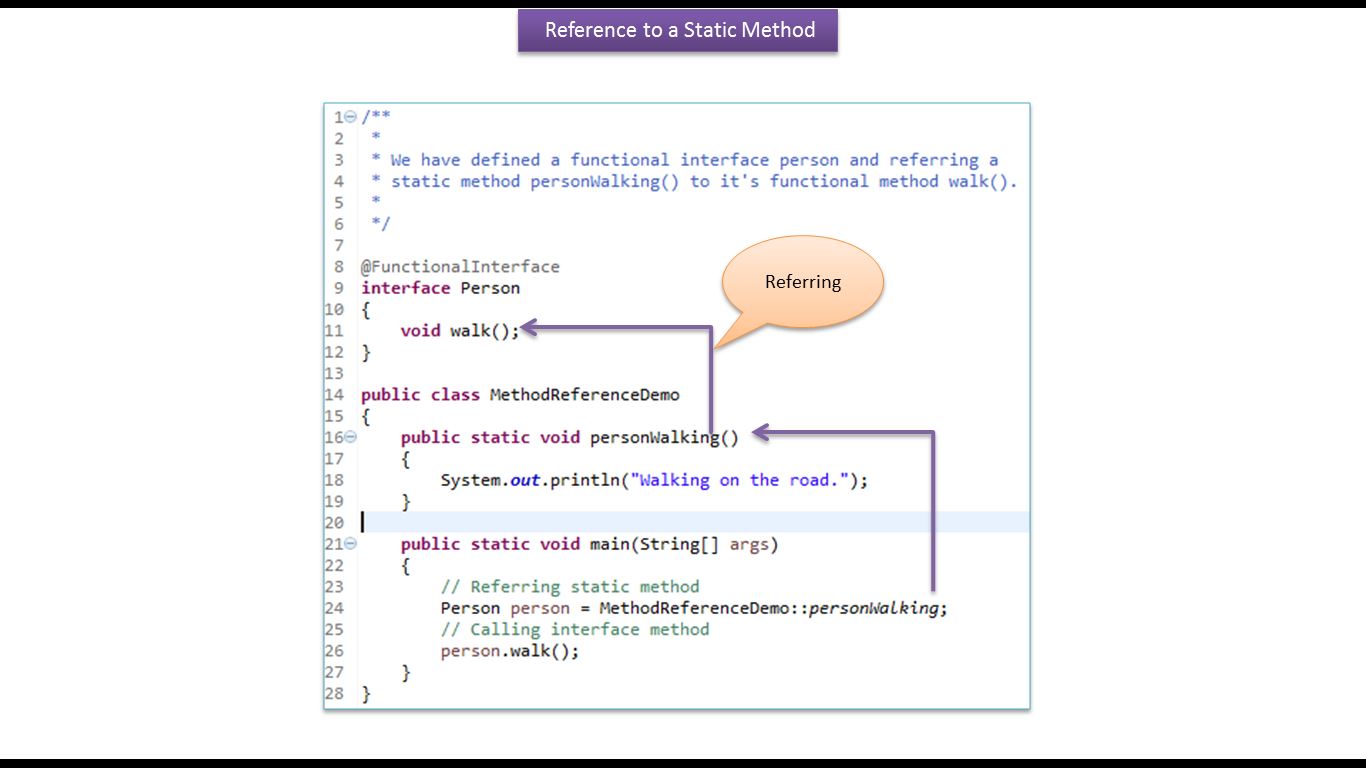
**}**

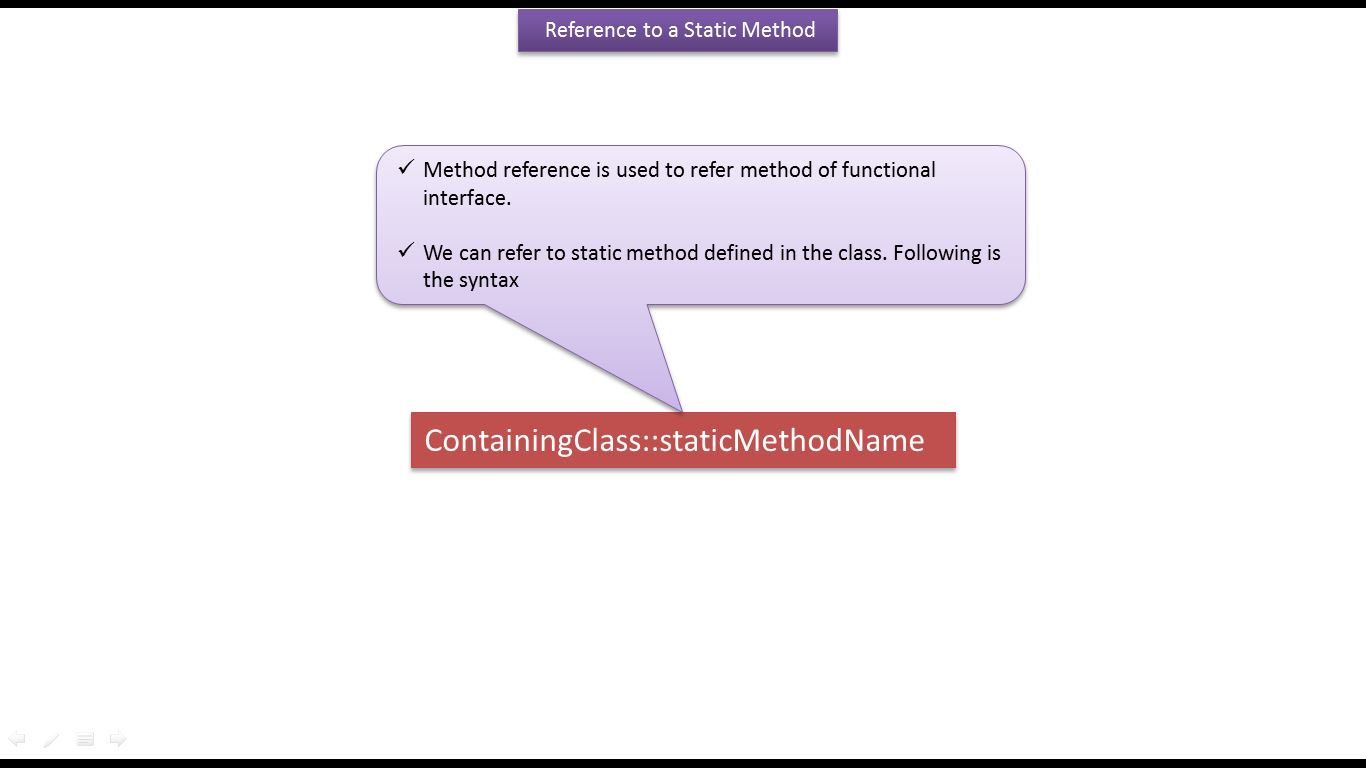
**}**

**Output**

**Dog is eating chicken.**

**Static Method reference - Person| Method reference in Java 8 | Java method reference**

****

****

**MethodReferenceDemo.java**

**/\*\***

**\***

**\* We have defined a functional interface person and referring a**

**\* static method personWalking() to it's functional method walk().**

**\***

**\*/**

**@FunctionalInterface**

**interface Person**

**{**

**void walk();**

**}**

**public class MethodReferenceDemo**

**{**

**public static void personWalking()**

**{**

**System.out.println("Walking on the road.");**

**}**

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

**{**

**// Referring static method**

**Person person = MethodReferenceDemo::personWalking;**

**// Calling interface method**

**person.walk();**

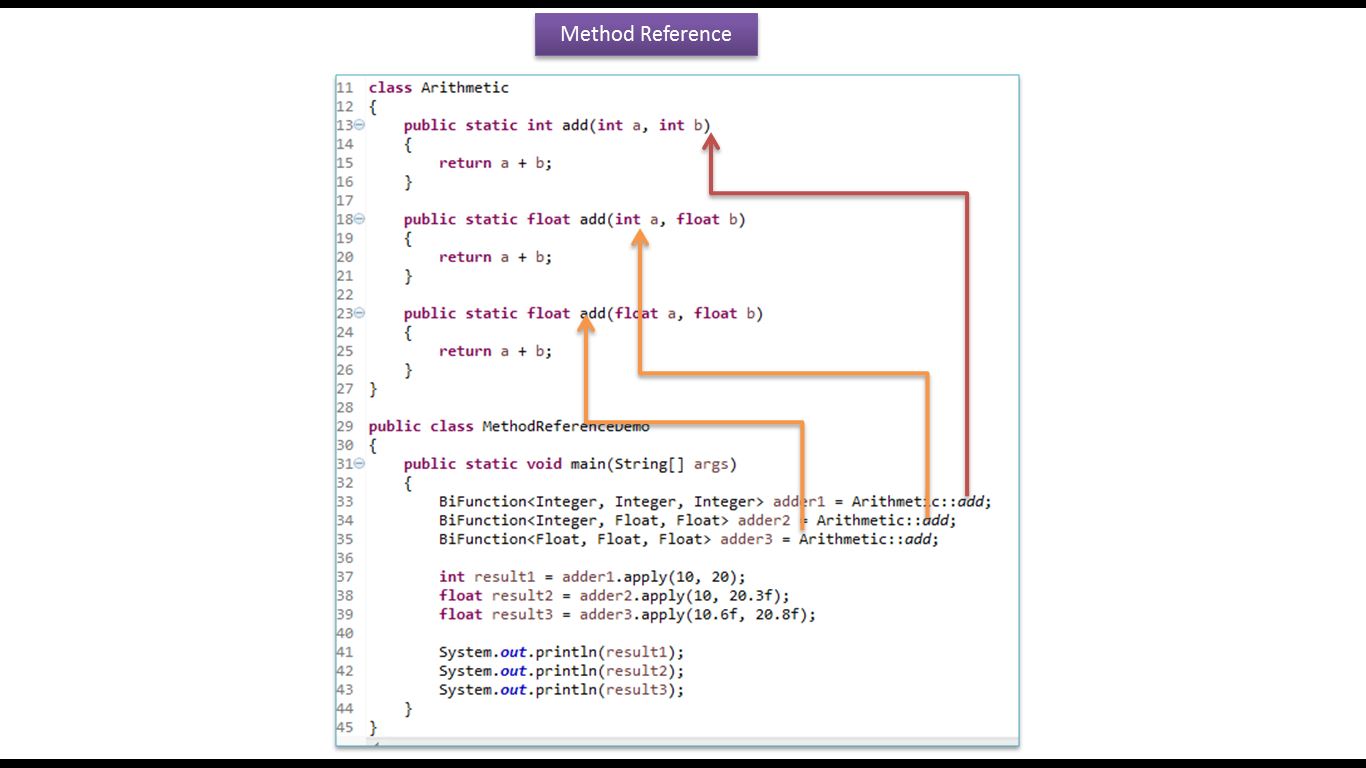
**}**

**}**

**Output**

**Walking on the road.**

**Static Method reference - BiFunction Overload methods | Method reference in Java 8**

****

**MethodReferenceDemo.java**

**import java.util.function.BiFunction;**

**/\*\***

**\***

**\* We can also override static methods by referring methods. In the**

**\* following example, we have defined and overloaded three add**

**\* methods.**

**\***

**\*/**

**class Arithmetic**

**{**

**public static int add(int a, int b)**

**{**

**return a + b;**

**}**

**public static float add(int a, float b)**

**{**

**return a + b;**

**}**

**public static float add(float a, float b)**

**{**

**return a + b;**

**}**

**}**

**public class MethodReferenceDemo**

**{**

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

**{**

**BiFunction<Integer, Integer, Integer> adder1 = Arithmetic::add;**

**BiFunction<Integer, Float, Float> adder2 = Arithmetic::add;**

**BiFunction<Float, Float, Float> adder3 = Arithmetic::add;**

**int result1 = adder1.apply(10, 20);**

**System.out.println(result1);**

**float result2 = adder2.apply(10, 20.3f);**

**System.out.println(result2);**

**float result3 = adder3.apply(10.6f, 20.8f);**

**System.out.println(result3);**

**}**

**}**

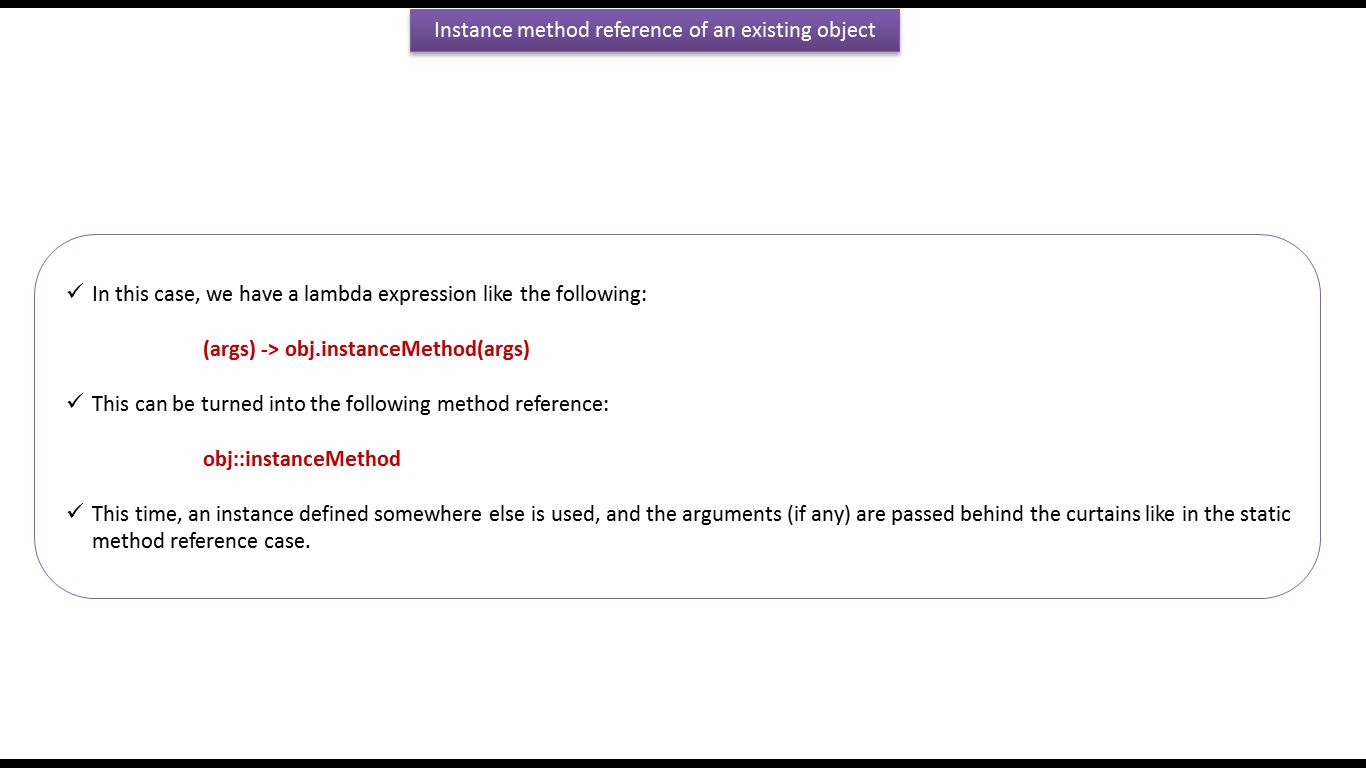
**Output**

***30***

***30.3***

***31.4***

**Instance method reference of an existing object| Method reference in Java 8 | Java method reference**

****

**Mechanic.java**

**import java.util.function.Consumer;**

**class Car**

**{**

**private String name;**

**public Car(String name)**

**{**

**super();**

**this.name = name;**

**}**

**public String getName()**

**{**

**return name;**

**}**

**public void setName(String name)**

**{**

**this.name = name;**

**}**

**}**

**public class Mechanic**

**{**

**public void fix(Car c)**

**{**

**System.*out*.println("Mechanic is fixing " + c.getName());**

**}**

**public void execute(Car car, Consumer<Car> c)**

**{**

**c.accept(car);**

**}**

**}**

**LambdaDemo.java**

**import java.util.function.Consumer;**

**public class LambdaDemo**

**{**

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

**{**

**final Mechanic mechanic = new Mechanic();**

**Car car = new Car("Honda Jazz");**

**// Using an anonymous class**

**mechanic.execute(car, new Consumer<Car>()**

**{**

**public void accept(Car c)**

**{**

**mechanic.fix(c);**

**}**

**});**

**// Using a lambda expression**

**mechanic.execute(car, c -> mechanic.fix(c));**

**// Using a method reference**

**mechanic.execute(car, mechanic::fix);**

**}**

**}**

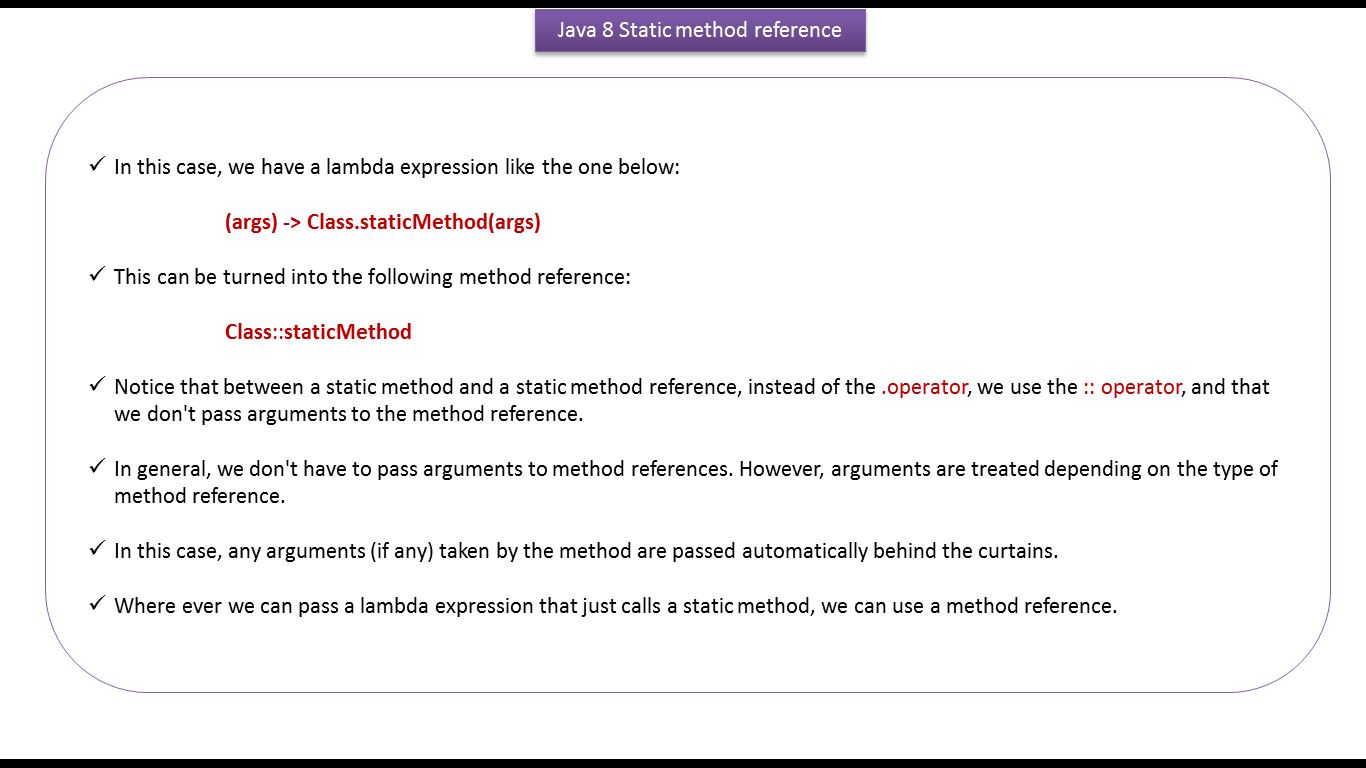
**Output**

**Mechanic is fixing Honda Jazz**

**Mechanic is fixing Honda Jazz**

**Mechanic is fixing Honda Jazz**

**Static Method reference - Number more than 50| Method reference in Java 8 | Java method reference**

****

**Numbers.java**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.function.BiPredicate;**

**class Numbers**

**{**

**public static boolean isMoreThanFifty(int n1, int n2)**

**{**

**return (n1 + n2) > 50;**

**}**

**public static List<Integer> findNumbers(List<Integer> list,**

**BiPredicate<Integer, Integer> p)**

**{**

**List<Integer> newList = new ArrayList<>();**

**for (Integer i : list)**

**{**

**if (p.test(i, i + 10))**

**{**

**newList.add(i);**

**}**

**}**

**return newList;**

**}**

**}**

**LambdaDemo.java**

**import java.util.Arrays;**

**import java.util.List;**

**import java.util.function.BiPredicate;**

**public class LambdaDemo**

**{**

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

**{**

**List<Integer> list = Arrays.*asList*(12, 5, 45, 18, 33, 24, 40);**

**// Using an anonymous class**

**List<Integer> listOfNumbers = Numbers.findNumbers(list, new BiPredicate<Integer, Integer>()**

**{**

**public boolean test(Integer i1, Integer i2)**

**{**

**return Numbers.isMoreThanFifty(i1, i2);**

**}**

**});**

**System.*out*.println("listOfNumbers using a anonymous class = "+listOfNumbers);**

**// Using a lambda expression**

**listOfNumbers =Numbers.findNumbers(list, (i1, i2) -> Numbers.isMoreThanFifty(i1, i2));**

**System.*out*.println("listOfNumbers using a lambda expression = "+listOfNumbers);**

**// Using a method reference**

**listOfNumbers = Numbers.findNumbers(list, Numbers::isMoreThanFifty);**

**System.*out*.println("listOfNumbers using a method reference = "+listOfNumbers);**

**}**

**}**

**Output**

**listOfNumbers using a anonymous class = [*45*, *33*, *24*, *40*]**

**listOfNumbers using a lambda expression = [*45*, *33*, *24*, *40*]**

**listOfNumbers using a method reference = [*45*, *33*, *24*, *40*]**

**Static Method reference - Number less than 50| Method reference in Java 8 | Java method reference**

**Numbers.java**

**import java.util.ArrayList;**

**import java.util.List;**

**import java.util.function.Predicate;**

**class Numbers**

**{**

**public static boolean isLessThanFifty(int number)**

**{**

**return number < 50;**

**}**

**public static List<Integer> findNumbers(List<Integer> list,**

**Predicate<Integer> p)**

**{**

**List<Integer> newList = new ArrayList<>();**

**for (Integer i : list)**

**{**

**/\***

**\* test the number is less than 50**

**\* then add to the newList.**

**\*/**

**if (p.test(i))**

**{**

**newList.add(i);**

**}**

**}**

**return newList;**

**}**

**}**

**LambdaDemo.java**

**import java.util.Arrays;**

**import java.util.List;**

**import java.util.function.Predicate;**

**public class LambdaDemo**

**{**

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

**{**

**List<Integer> list = Arrays.*asList*(12, 55, 75, 88, 24, 40);**

**// Using an anonymous class**

**List<Integer> listOfLessthanfiftyNumbers = Numbers.findNumbers(list, new Predicate<Integer>()**

**{**

**public boolean test(Integer i)**

**{**

**return Numbers.isLessThanFifty(i);**

**}**

**});**

**System.*out*.println("listOfLessthanfiftyNumbers - anonymous class = "+listOfLessthanfiftyNumbers);**

**// Using a lambda expression**

**listOfLessthanfiftyNumbers = Numbers.findNumbers(list, (i) -> Numbers.isLessThanFifty(i));**

**System.*out*.println("listOfLessthanfiftyNumbers - lambda expression = "+listOfLessthanfiftyNumbers);**

**// Using a method reference**

**listOfLessthanfiftyNumbers = Numbers.findNumbers(list, Numbers::isLessThanFifty);**

**System.*out*.println("listOfLessthanfiftyNumbers - method reference = "+listOfLessthanfiftyNumbers);**

**}**

**}**

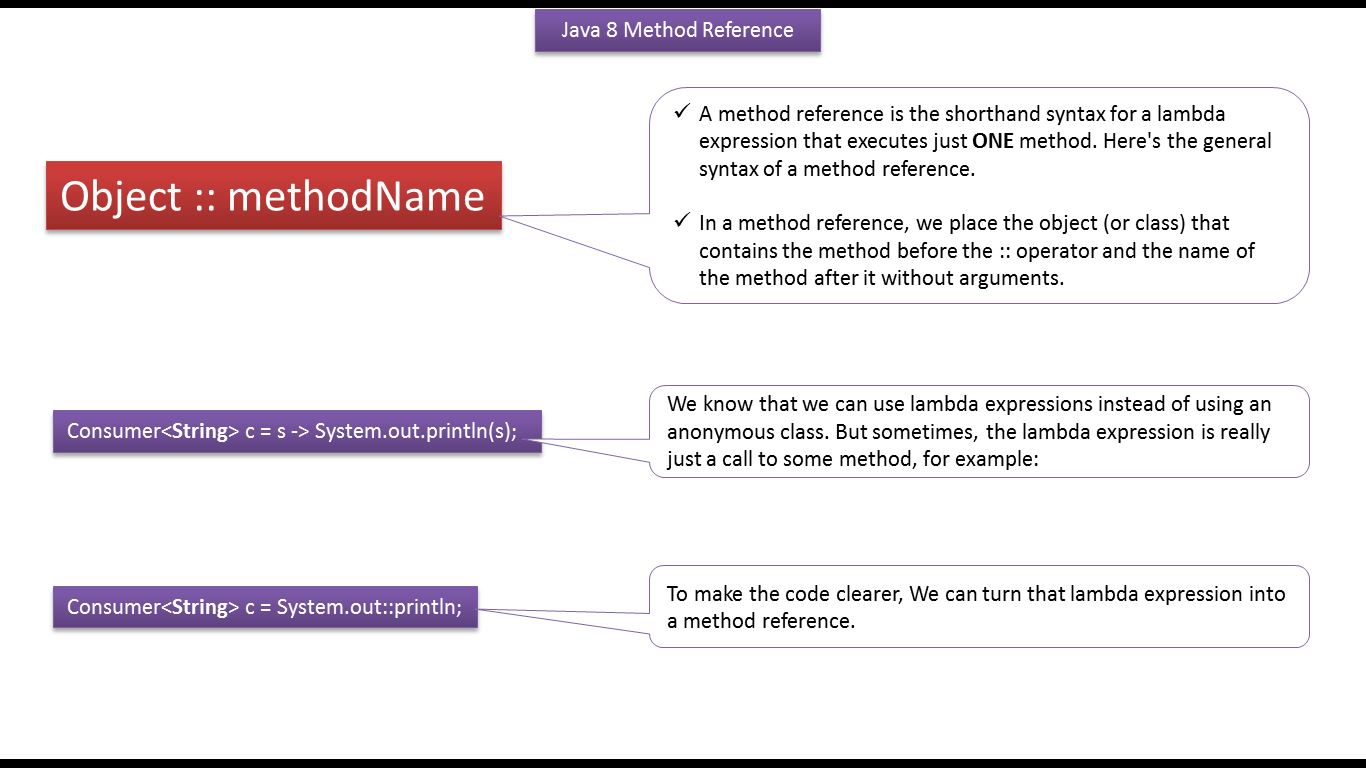
**Output**

**listOfLessthanfiftyNumbers - anonymous class = [12, 24, 40]**

**listOfLessthanfiftyNumbers - lambda expression = [12, 24, 40]**

**listOfLessthanfiftyNumbers - method reference = [12, 24, 40]**

**How to use Method reference in forEach method of List | Method reference in Java 8**

****

**MethodReferenceDemo.java**

**import java.util.ArrayList;**

**import java.util.List;**

**public class MethodReferenceDemo**

**{**

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

**{**

**List<String> nameList = new ArrayList<>();**

**nameList.add("Peter");**

**nameList.add("John");**

**nameList.add("Juli");**

**nameList.add("Stephan");**

**nameList.forEach(name -> System.*out*.println(name));**

**System.*out*.println("------------------------");**

**// method reference**

**nameList.forEach(System.*out*::println);**

**}**

**}**

**Output**

**Peter**

**John**

**Juli**

**Stephan**

**------------------------**

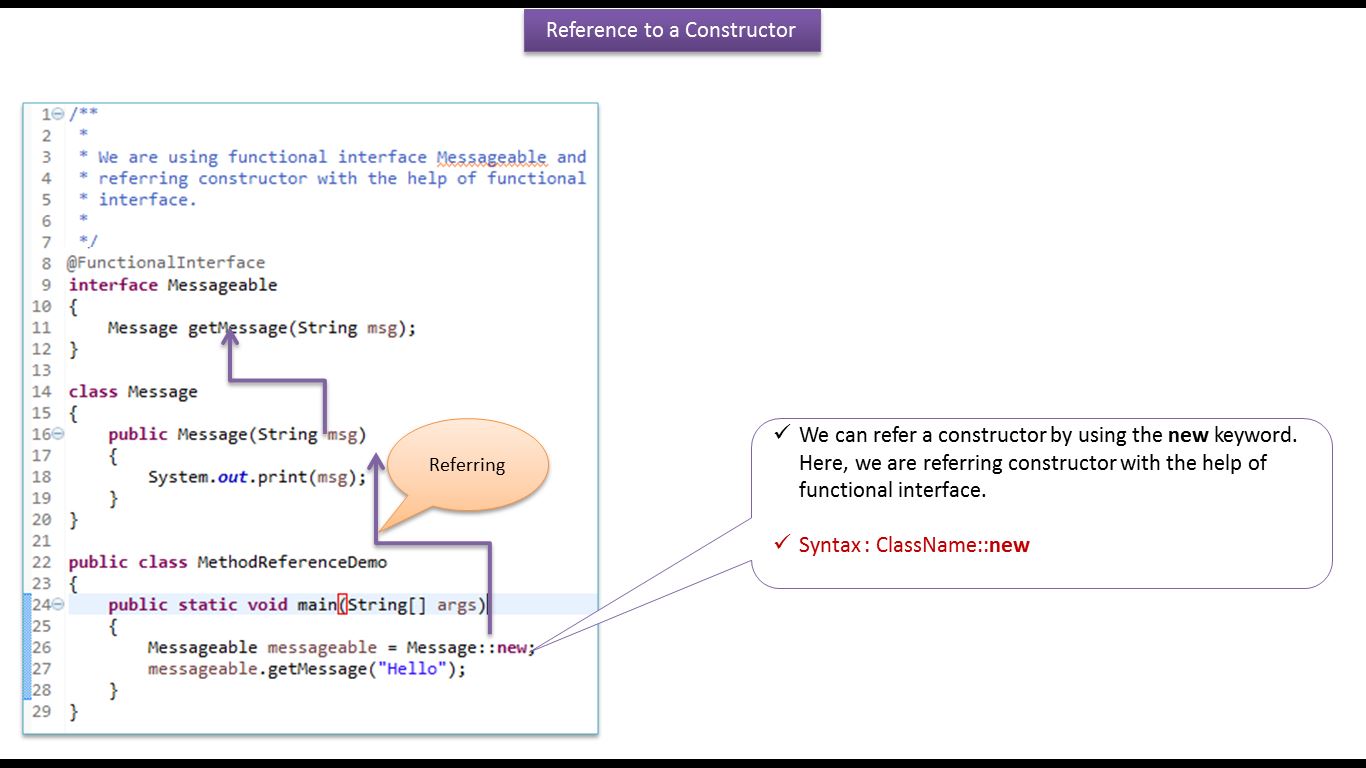
**Peter**

**John**

**Juli**

**Stephan**

**How to refer a constructor using method reference | Method reference in Java 8**

****

**MethodReferenceDemo.java**

**/\*\***

**\***

**\* We are using functional interface Messageable and referring constructor with**

**\* the help of functional interface.**

**\***

**\*/**

**@FunctionalInterface**

**interface Messageable**

**{**

**Message getMessage(String msg);**

**}**

**class Message**

**{**

**public Message(String msg)**

**{**

**System.*out*.print(msg);**

**}**

**}**

**public class MethodReferenceDemo**

**{**

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

**{**

**Messageable messageable = Message::new;**

**messageable.getMessage("Hello");**

**}**

**}**

**Output**

**Hello**

**Non-Static Method reference - BiFunction | Method reference in Java 8 | Java method reference**

**MethodReferenceDemo.java**

**import java.util.function.BiFunction;**

**/\*\***

**\***

**\* We are using predefined functional interface BiFunction and**

**\* referring a non-static method add(int a, int b) to it's functional**

**\* method apply(Integer t,Integer u).**

**\***

**\*/**

**class Arithmetic**

**{**

**public int add(int a, int b)**

**{**

**return a + b;**

**}**

**}**

**public class MethodReferenceDemo**

**{**

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

**{**

**BiFunction<Integer, Integer, Integer> adder = new Arithmetic()::add;**

**int result = adder.apply(200, 200);**

**System.*out*.println(result);**

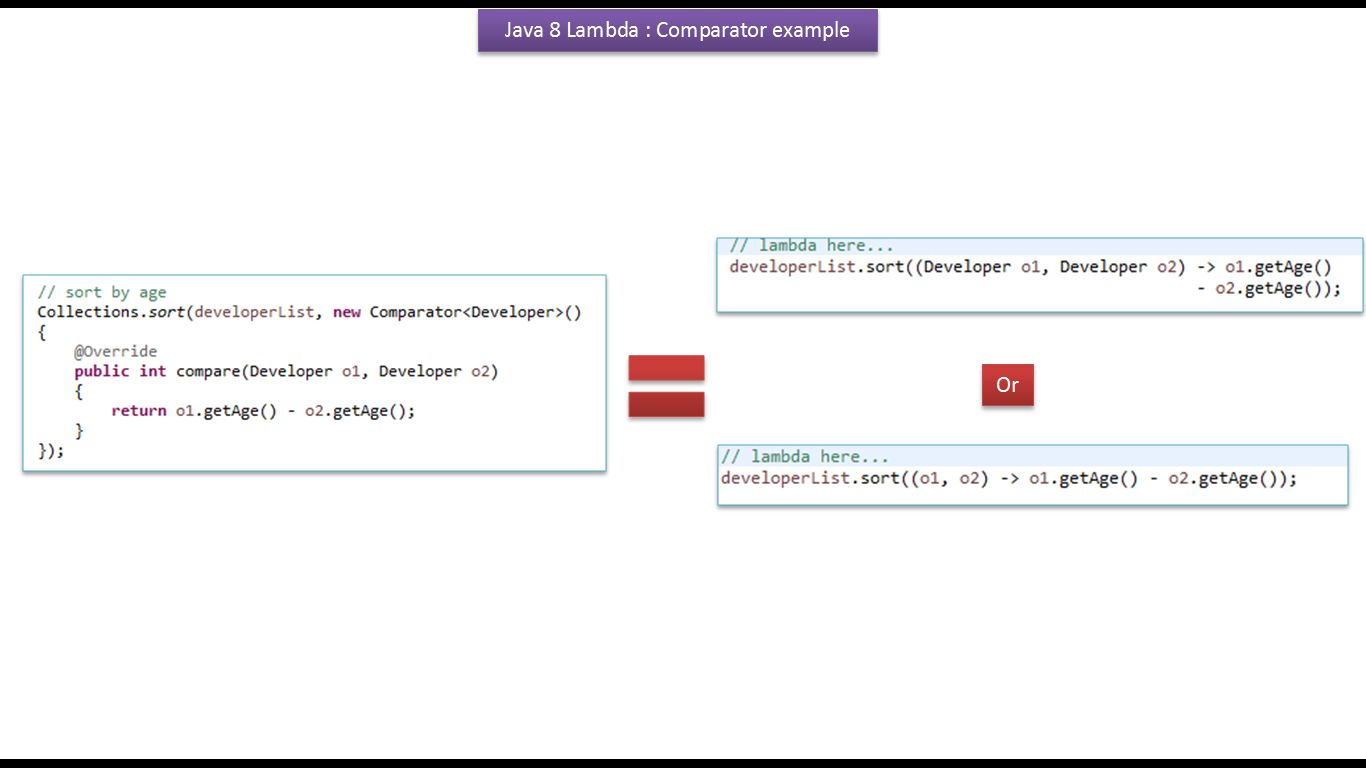
**}**

**}**

**Output**

***400***

**Lambda expression in Java[How to sort the list of developers using lambda expression and comparator]**

****

**Developer.java**

**public class Developer**

**{**

**private String name;**

**private int age;**

**public Developer(String name, int age)**

**{**

**super();**

**this.name = name;**

**this.age = age;**

**}**

**public String getName()**

**{**

**return name;**

**}**

**public void setName(String name)**

**{**

**this.name = name;**

**}**

**public int getAge()**

**{**

**return age;**

**}**

**public void setAge(int age)**

**{**

**this.age = age;**

**}**

**@Override**

**public String toString()**

**{**

**return "Developer [name=" + name + ", age=" + age + "]";**

**}**

**}**

**NonLambdaDemo.java**

**import java.util.ArrayList;**

**import java.util.Collections;**

**import java.util.Comparator;**

**import java.util.List;**

**/\*\***

**\* Sort without Lambda,Before JDK 8**

**\*/**

**public class NonLambdaDemo**

**{**

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

**{**

**List<Developer> developerList = *getDevelopers*();**

**System.*out*.println("-----------Before Sort-----------");**

***printDeveloperInfo*(developerList);**

**// sort by age**

**Collections.*sort*(developerList, new Comparator<Developer>()**

**{**

**@Override**

**public int compare(Developer o1, Developer o2)**

**{**

**return o1.getAge() - o2.getAge();**

**}**

**});**

**System.*out*.println("\n-----------After Sort by Age------------");**

***printDeveloperInfo*(developerList);**

**// sort by name**

**Collections.*sort*(developerList, new Comparator<Developer>()**

**{**

**@Override**

**public int compare(Developer o1, Developer o2)**

**{**

**return o1.getName().compareTo(o2.getName());**

**}**

**});**

**System.*out*.println("\n-----------After Sort by Name------------");**

***printDeveloperInfo*(developerList);**

**}**

**private static void printDeveloperInfo(List<Developer> developerList)**

**{**

**for (Developer developer : developerList)**

**{**

**System.*out*.println(developer);**

**}**

**}**

**private static List<Developer> getDevelopers()**

**{**

**List<Developer> developerList = new ArrayList<Developer>();**

**developerList.add(new Developer("Peter", 33));**

**developerList.add(new Developer("John", 22));**

**developerList.add(new Developer("Ram", 15));**

**developerList.add(new Developer("Steve", 55));**

**return developerList;**

**}**

**}**

Output

**-----------**Before Sort**-----------**

Developer [name**=**Peter, age**=***33*]

Developer [name**=**John, age**=***22*]

Developer [name**=**Ram, age**=***15*]

Developer [name**=**Steve, age**=***55*]

**-----------**After Sort by Age**------------**

Developer [name**=**Ram, age**=***15*]

Developer [name**=**John, age**=***22*]

Developer [name**=**Peter, age**=***33*]

Developer [name**=**Steve, age**=***55*]

**-----------**After Sort by Name**------------**

Developer [name**=**John, age**=***22*]

Developer [name**=**Peter, age**=***33*]

Developer [name**=**Ram, age**=***15*]

Developer [name**=**Steve, age**=***55*]

**LambdaDemo.java**

**import** java.util.ArrayList;

**import** java.util.List;

/\*\*

\* Sort with Lambda,With JDK 8

\*/

**public** **class** LambdaDemo

{

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

{

List<Developer> developerList = *getDevelopers*();

System.***out***.println("-----------Before Sort-----------");

**for** (Developer developer : developerList)

{

System.***out***.println(developer);

}

System.***out***.println("\n-----------After Sort by Age------------");

// lambda here...

developerList.sort((Developer o1, Developer o2) -> o1.getAge() - o2.getAge());

// java 8 only, lambda also, to print the List

developerList.forEach((developer) -> System.out.println(developer));

System.***out***.println("\n-----------After Sort by Name------------");

// lambda

developerList.sort((Developer o1, Developer o2) -> o1.getName().compareTo(o2.getName()));

// java 8 only, lambda also, to print the List

developerList.forEach((developer) -> System.out.println(developer));

}

**private** **static** List<Developer> getDevelopers()

{

List<Developer> developerList = **new** ArrayList<Developer>();

developerList.add(**new** Developer("Peter", 33));

developerList.add(**new** Developer("John", 22));

developerList.add(**new** Developer("Ram", 15));

developerList.add(**new** Developer("Steve", 55));

**return** developerList;

}

}

Output

**-----------**Before Sort**-----------**

Developer [name**=**Peter, age**=***33*]

Developer [name**=**John, age**=***22*]

Developer [name**=**Ram, age**=***15*]

Developer [name**=**Steve, age**=***55*]

**-----------**After Sort by Age**------------**

Developer [name**=**Ram, age**=***15*]

Developer [name**=**John, age**=***22*]

Developer [name**=**Peter, age**=***33*]

Developer [name**=**Steve, age**=***55*]

**-----------**After Sort by Name**------------**

Developer [name**=**John, age**=***22*]

Developer [name**=**Peter, age**=***33*]

Developer [name**=**Ram, age**=***15*]

Developer [name**=**Steve, age**=***55*]