Java -8

What is Java Interface?

An interface in Java is a blueprint of a class.is *a mechanism to achieve abstraction*.

Theoretically, a Java Interface is an abstract type which is used to specify the behavior of a class. Since it is an abstract type, it will only represent the behavior of a class, but not the complete implementation. Abstraction is the process of hiding certain details and showing only essential information to the user. Generally, abstraction can be achieved with either interfaces or abstract classes. That's why we can also say that an interface is a blueprint of a class. In other words, the interface provides us a mechanism to achieve abstraction.

Before JDK 8:

An Interface in Java is a reference type similar to a class that can contain constants as fields/variables, abstract methods and nested types. Fields are by default public, static and final whereas methods are by default public and abstract whether we declare or not.

- Constant Variables
- Abstract Methods
- Nested Types

JDK 8 Onward:

JDK 8 onward, an Interface can contain constant variables, abstract methods, default methods, static methods, nested types. However, default and static methods will have a method body, but abstract method will not. When declaring default and static methods, it is mandatory to write keywords default and static respectively. If you don't write these keywords, the method will be considered as the abstract method by default by the compiler. Even in this case, you will not be allowed to write method body as well.

The purpose of providing default methods is to allow the developers to add new functionalities to their interfaces if needed, without breaking their older version of code.

- Constant Variables
- Abstract Methods
- Nested Types
- Default Methods
- Static Methods

JDK 9 Onward:

In addition to all members of Java 8 interface, Java 9 introduced private methods and private static methods in interfaces. Static methods in java 8 can not be private but from java 9 onward they can be. Therefore, from Java 9 onward an interface can have the following components inside it.

- Constant Variables
- Abstract Methods
- Nested Types
- Default Methods
- Static Methods
- Private Methods
- Private Static Methods

What are the key rules about private methods in Interfaces?

There are some key rules about private methods inside interfaces, which we need to understand properly. They are as follows:

1) A private method must have a body. It means they can't be declared as abstract.

2) Private methods can be accessed within the interface only.

3) We can't override them in implementing classes as they are private.

4) Private static method can be used inside other static and non-static interface methods.

5) Private non-static methods can't be used inside private static methods.

6) With private methods, only static & strictfp access modifiers are allowed.

Why do we use interface?

The Java Interface offers a Java Programmer to achieve abstraction.However, JDK 8 onwards, the interface doesn't provide a complete abstraction as we can define default & static methods as well.

Since Java doesn't support multiple inheritance, and if you want a class to achieve multiple inheritances, there is only one way: using interfaces.

While designing your application's API, Interfaces allow you to specify what methods a class should implement. Therefore, Interface becomes key to API design.

A class can't implement two interfaces that have methods with the same name, but different return type.

'implements' keyword is used by classes to implement an interface.

What is the difference between Java Interface and a Java Class?

Below differences are based on the current features of Java interfaces(including Java 8 & Java 9) as of now.

1) We can create an object of from a class, whereas we can't create an object from an interface.

2) A class can extend only one class, whereas a class can implement any number of interfaces.

3) We can use private, public, protected access specifiers with the methods of a class, whereas we can use only private and public access specifiers with the methods of an interface. Remember that since Java 9, we have been allowed to declare private methods in an Interface.

4) A class can extend another class and also implement one or multiple interfaces, whereas an Interface can only extend one or more interfaces.

5) At the time of declaration, the interface variable must be initialized. Otherwise, the compiler will throw an error. It is fine with the class without initializing any variable at the time of declaration.

Difference between Interface and Abstract class

How many types of Interfaces are there in Java ?

Typically we have three types of Interfaces till now.

1) Normal Interface

Normal Interface is an interface which has either one or multiple number of abstract methods.

2) Marker Interface

It is an empty interface (no field or methods). Examples of marker interface are Serializable, Cloneable and Remote interface. All these interfaces are empty interfaces.

3) Functional Interface

Functional Interface is an interface which has only one abstract method. Further, it can have any number of static, default methods and even public methods of java.lang.Object class.

How to write a Java Functional Interface?

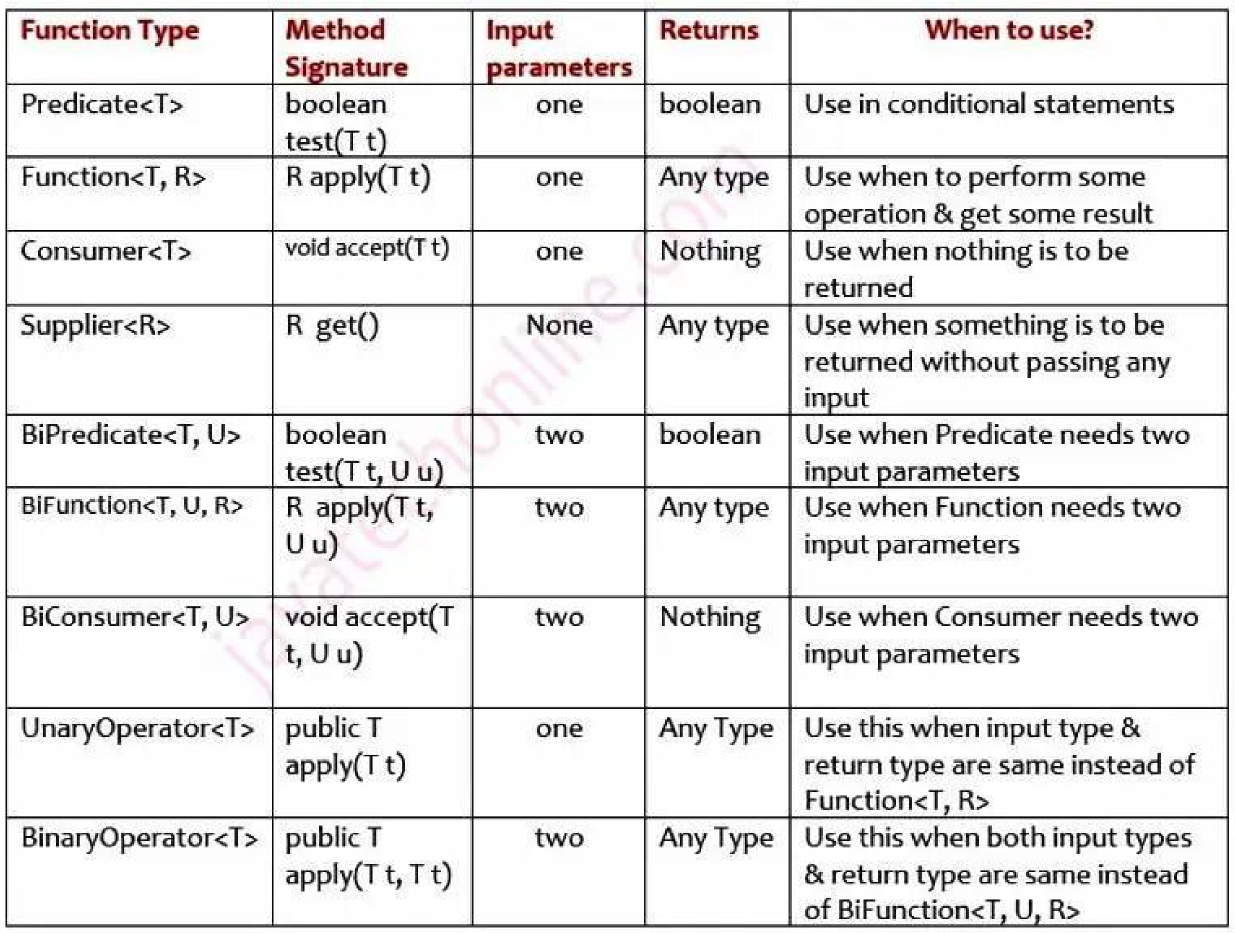
In addition to having only one abstract method, we should write *@FunctionalInterface* annotation in order to let the compiler know that the interface is a Functional. In fact, if you add annotation *@FunctionalInterface*, the compiler will not let you add any other abstract method inside it.

If an interface extends Functional Interface and child interface doesn't contain any abstract method, then again child interface is also a Functional Interface.

In the child interface we are allowed to define exactly same single abstract method as it is in the parent interface as below.

In other words, the child interface will not have any new abstract methods otherwise child interface won't remain a Functional Interface and if we try to use *@FunctionalInterface* annotation in child interface, compiler will show an error message. If we don't use *@FunctionalInterface* in child interface & declare new abstract method, the compiler will consider it as a normal interface & will not show any error.

Predefined Functional Interfaces



Function Type	Method Signature	Input parameters	Returns	When to use?
Predicate<T>	boolean test(T t)	one	boolean	Use in conditional statements
Function<T, R>	R apply(T t)	one	Any type	Use when to perform some operation & get some result
Consumer<T>	void accept(T t)	one	Nothing	Use when nothing is to be returned
Supplier<R>	R get()	None	Any type	Use when something is to be returned without passing any input
BiPredicate<T, U>	boolean test(T t, U u)	two	boolean	Use when Predicate needs two input parameters
BiFunction<T, U, R>	R apply(T t, U u)	two	Any type	Use when Function needs two input parameters
BiConsumer<T, U>	void accept(T t, U u)	two	Nothing	Use when Consumer needs two input parameters
UnaryOperator<T>	public T apply(T t)	one	Any Type	Use this when input type & return type are same instead of Function<T, R>
BinaryOperator<T>	public T apply(T t, T t)	two	Any Type	Use this when both input types & return type are same instead of BiFunction<T, U, R>

Lambda Expression In Java

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

What is a Lambda (λ) 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.

We use Lambda Expression in Java when we provide an implementation of Functional Interfaces which is also a new feature of Java 8. Lambda expression in Java can also be used in a method as an argument which is one of the important feature of functional programming as the primary focus of Java 8.

Lambda Expression : A Lambda expression has only two parts :

1) Parameter List

2) Body

Note on Lambda Expression in Java

1) If only one method parameter is available and compiler can understand the type based on the context, then we can remove the type & parenthesis both. Suppose (String s) -> {System.out.println(s)}; can be simplified to s -> {System.out.println(s)};

2) Similar to method body lambda expression body can contain multiple statements. If more than one statements present then we have to enclose it within curly braces. If one statement present then curly braces are optional. As in example at point # 1 can again be simplified as

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

3) If no parameters are available, then we can use empty parenthesis like :

```
( ) -> System.out.println(“No parameter test”);
```

4) We can call the Lambda expression just like a traditional method, but only with the help of functional interfaces. We have covered Functional Interfaces topic in detail in other section.

5) Equally important, if you are learning Lambda expressions for the first time, Keep below syntax in your mind :

```
Interface iObj  = (params) -> { method body };
```

Benefits Of Using Lambda Expressions

1) We can write more readable, maintainable & concise code using Lambda expression in Java.

2) Also, we can incorporate functional programming capabilities in java language with Lambda Expressions.

3) We can use Lambda expressions in place of Inner classes to reduce the complexity of code accordingly.

4) Even we can pass a lambda expression as an argument to a method.

5) Additionally, we can provide lambda expression in place of an object.

```
//Code without using Lambda Expressions

Thread t = new Thread(new Runnable() {

   public void run() {

     for (int i = 0; i < 10; i++) {

        System.out.println("Child thread is running");

     }

   }

});
```

⇓

```
//Code after using Lambda expression

Thread t = new Thread(() -> {

   for (int i = 0; i < 10; i++) {

      System.out.println("Child thread is running");

   }

});
```

Variable’s Behavior Inside Lambda Expression

1) We can't declare instance variable inside a Lambda expression.

2) Of course, variables declared inside Lambda expressions will be treated as a local variable.

3) "this" keyword within Lambda expressions represents current outer class Object reference i.e. the class reference in which the Lambda expression is declared.

Difference between Lambda Expression & Anonymous Inner class

Now its turn to find out the differences between them.

Lambda Expression	Anonymous Inner class
“this“ keyword inside Lambda Expression always refers to current outer class object ie. enclosing class object.	“this“ keyword inside anonymous inner class always refers to current anonymous Inner class object but not outer class object.
Lambda expression can’t extend any class like abstract and concrete classes.	In contrast, an Anonymous inner class can extend any class like abstract and concrete classes.

Lambda expression can implement an Interface containing single abstract method ie Functional Interface.	Moreover, An Anonymous inner class can implement an interface containing any number of abstract methods.
Lambda is called anonymous function ie. method without name.	However, an Anonymous Inner class is called a class without name.
However, a Lambda Expression occupies permanent memory of JVM.	Anonymous Inner class occupies memory whenever object is created on demand.
For Lambda Expression at the time of compilation, no .class file fill be generated.	For Anonymous Inner class at the time of compilation, a separate .class file will be generated.
In order to implement a Functional Interface, We must go with Lambda Expressions.	In order to operate on multiple methods, We should go with Anonymous Inner classes.
We can't declare instance variables inside Lambda Expressions and thus they don't have state. Variables	However, we can declare instance variables inside Anonymous Inner classes and thus they have state.

declared acts as local variables.	
We can't instantiate Lambda Expressions.	In contrast, we can instantiate Anonymous Inner classes.
The Performance of the lambda expression is better as it is pure compile time activity and don't incur extra cost during runtime.	However, Performance of anonymous inner class is lower as requires class loading at runtime.

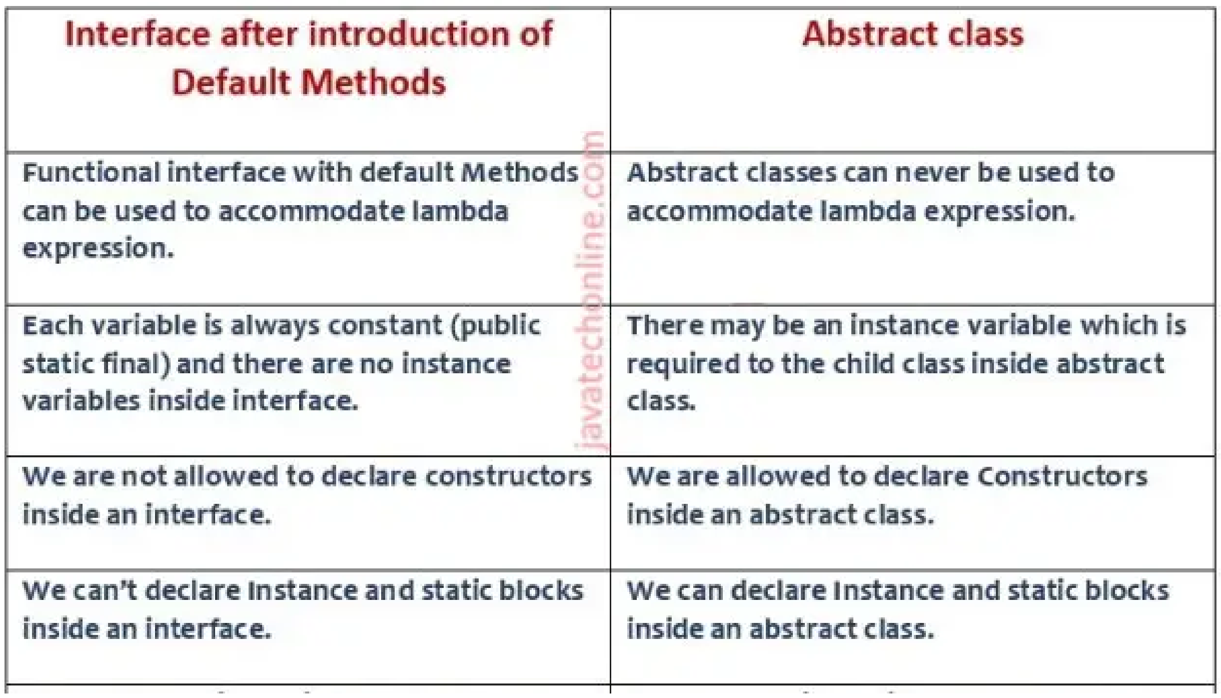
Default Method in Interface

If we have an implemented method inside an interface with default keyword, then we will call it as a Default method. We also call it defender method or virtual extension method. The default method in interface was introduced in JDK 8.

Default methods offer us to add new methods to an interface in future that will be available in the implementations automatically. Thus, there's no need to modify the implementing classes after including a default method in interface.

Difference between Interface & Abstract class after introduction of default Method in Interface

The major difference between the Interface & an abstract class before JDK 1.8 was that all the methods in an Interface are abstract by default while in an abstract class it is not necessary that all methods are abstract. Since JDK 1.8 after the introduction of default method in interface, this difference becomes false. It doesn't mean that both are similar now as we still have many differences between them. Let's go through the table below to see the differences which are still in place.



Interface after introduction of Default Methods	Abstract class
Functional interface with default Methods can be used to accommodate lambda expression.	Abstract classes can never be used to accommodate lambda expression.
Each variable is always constant (public static final) and there are no instance variables inside interface.	There may be an instance variable which is required to the child class inside abstract class.
We are not allowed to declare constructors inside an interface.	We are allowed to declare Constructors inside an abstract class.
We can't declare Instance and static blocks inside an interface.	We can declare Instance and static blocks inside an abstract class.

Important Note on Default Methods

- We can have Default methods (method with default keyword) only inside Interfaces as of Java 8. They are not allowed inside classes, even if it is implemented class of that Interface.

- Any method inside an Interface can’t be declared default & static together. Default methods can’t be static & static methods can’t be default.
- Similar to regular methods, default method in interface is also public implicitly. Hence, we don’t need to specify the public modifier.
- Unlike regular interface methods, default method in interface has its implementation (method body).
- We can have any number of default method in an Interface.

What is a Static method?

A static method is a method that is affiliated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.

What are the Static methods in Interfaces?

Similar to Default Methods in Interfaces, Static Methods also have a method body (implementation). However, we can't override them. Needless to say, static methods can't be overridden.

Why do we need a Static Method in Interface?

If you want to provide some implementation in your interface and you don't want this implementation to be overridden in the implementing class, then you can declare the method as static. Hence, we can secure an implementation by having it in a static method as implementing classes can't override it. Moreover, we define static methods inside classes to use them as general utility methods. Similarly, we can define static methods in the interface to use them as general utility methods.

Since we can have static methods in interface starting from Java 8, we can define the main() method inside an interface without violating any rule of java. Also, we can compile & run this interface successfully. For example, below code is valid.

What is Method Reference(::)?

As we have seen in Lambda expression topic that we use lambda expressions to implement Functional interfaces with minimum lines of code and even to get better code readability. Similarly, we can use Method Reference(::) Java 8 to implement Functional interfaces with even lesser code again than lambda expressions and this time we get the benefit of code re-usability as well, because we don't provide an implementation for functional interface. Instead, we provide reference to already existing method (with similar argument types) to simplify the implementation of the functional interface using a double colon (::) operator. This process of providing reference to pre-existing method is called Method reference.

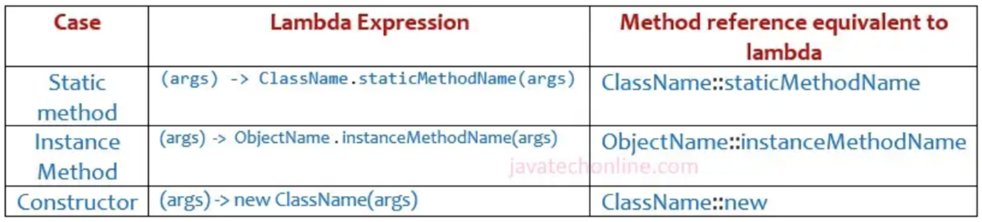
pre-existing methods should have same arguments in number & type and there are no restrictions on their return type as such.

Syntax to write Method References

Before knowing how to write them, let's first categorize the Method Reference(::) Java 8. Here, syntax to write Method Reference(::) Java 8 is different in different cases. In general, we can provide Method references in three ways, sometimes called the types of it.

1. Static Method References
2. Instance method/non-static Method References
3. Constructor References

Now, let's try to observe them one by one from the table below.



Case	Lambda Expression	Method reference equivalent to lambda
Static method	(args) -> ClassName.staticMethodName(args)	ClassName::staticMethodName
Instance Method	(args) -> ObjectName . instanceMethodName(args)	ObjectName::instanceMethodName
Constructor	(args) -> new ClassName(args)	ClassName::new

Types of Method References

Note : In addition, we have two more besides above three:

1. Method References of an arbitrary object of a given type
2. Method References from a super class method.

Method reference to a static method of a class :

```
interface A {

        public boolean checkSingleDigit(int x);

}


class Digit {

        public static boolean isSingleDigit(int x) {

            return x > -10 && x < 10;

        }

}


public class TestStaticMethodReference {


        public static void main(String[] args) {


                //*** Using Lambda Expression ***//

                A a1 = (x) -> { return x > -10 && x < 10;};

                System.out.println(a1.checkSingleDigit(10));
                //*** Using Method Reference ***//
                A a2 = Digit::isSingleDigit;
                System.out.println(a2.checkSingleDigit(9));
        }
}
```

In the above example, we have functional Interface A. We have implemented the single abstract method checkSingleDigit(int x) using Lambda expression in highlighted line. But while using Method reference, we have just referred the similar already existing static method isSingleDigit(int x) of class Digit.

Method reference to an Instance method of a class :

```
interface B {
   public  void add(int x, int y);
}


class Addition {
        public void sum(int a, int b) {
            System.out.println("The sum is :"+(a+b));
        }
}
```

```
public class TestInstanceMethodReference {

    public static void main(String[] args) {

        Addition addition = new Addition();

        //*** Using Lambda Expression ***//

        B b1 = (a,b) -> System.out.println("The sum is :"+(a+b));

        b1.add(10, 14);


        //*** Using Method Reference ***//

        B b2 = addition::sum;

        b2.add(100, 140);

    }

}
```

In the above example, we have functional Interface B. We have implemented the single abstract method add(int x, int y) using Lambda expression in highlighted line. While using Method reference, we have just referred the similar already existing method sum(int a, int b) of class Addition but after creating an object of the class this time.

Constructor Reference

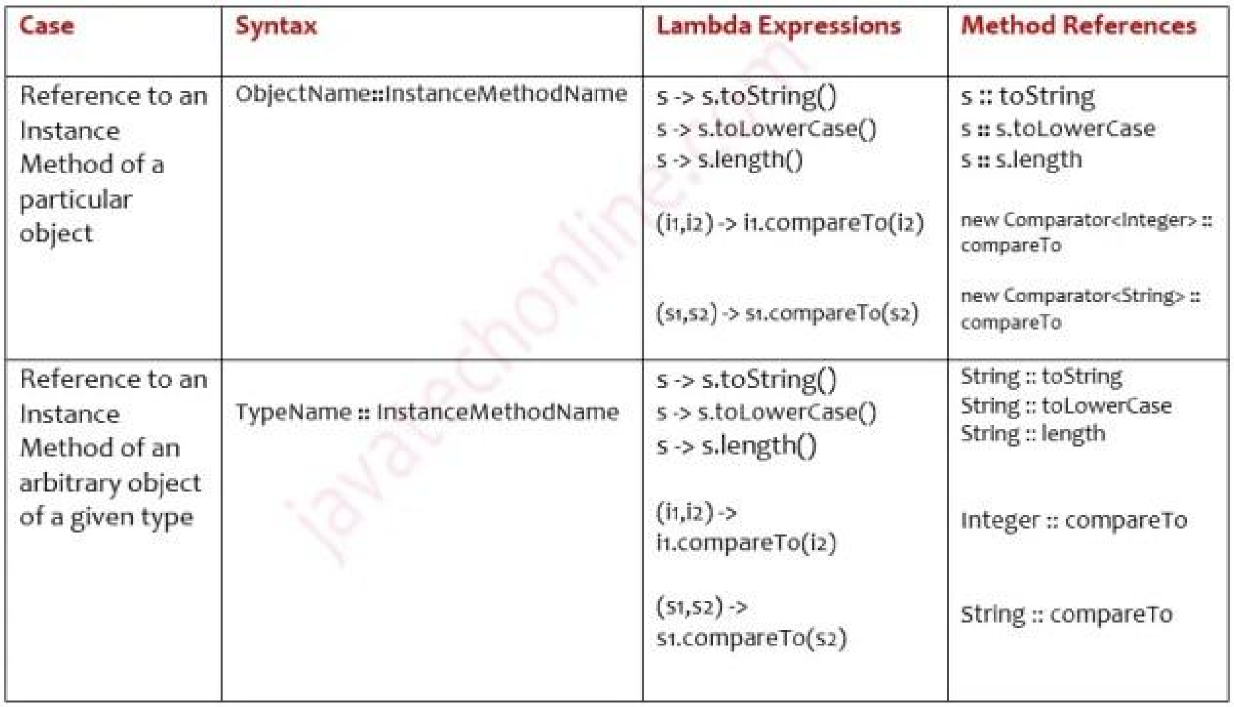
When single abstract method's return type is any Object, we will go with the constructor reference.

```
interface C {

        public Employee getEmployee();

}

interface D {

        public Employee getEmployee(String name, int age);

}


class Employee {

        String eName;

        int eAge;

        public Employee(){}


        public Employee(String eName, int eAge) {

                this.eName = eName;

                this.eAge = eAge;

        }
        public void getInfo() {
                System.out.println("I am a method of class Employee");
        }
}

public class TestConstructorReference {
        public static void main(String[] args) {
                //*** Using Lambda Expression ***//
                C c1 = () -> new Employee();
                c1.getEmployee().getInfo();
                D d1 = (name,age) -> new Employee(name,age);
                d1.getEmployee("Tony", 34).getInfo();

                //*** Using Method Reference ***//
                C c2 = Employee::new;
                c2.getEmployee().getInfo();
                D d2 = Employee::new;
                d2.getEmployee("Tony", 34).getInfo();
        }
}
```

In the above code snippet, we have two Functional interfaces C & D. Interface C has abstract method with no arguments whereas D has with arguments. We will have same constructor reference code in both cases as shown in the highlighted lines of code. Also, if you run the code in your development environment, you will get the same output in each case.

Difference between reference to an Instance method of a particular object & an arbitrary Object of a given type

You might have some doubt on difference between both references, so now it's right time to talk about them. To illustrate, let's observe the examples given in the table below. I am sure you will get the clear idea on differences. There is no need to explain more about this.



Case	Syntax	Lambda Expressions	Method References
Reference to an Instance Method of a particular object	ObjectName::InstanceMethodName	s -> s.toString() s -> s.toLowerCase() s -> s.length() (i1,i2) -> i1.compareTo(i2) (s1,s2) -> s1.compareTo(s2)	s :: toString s :: s.toLowerCase s :: s.length new Comparator<Integer> :: compareTo new Comparator<String> :: compareTo
Reference to an Instance Method of an arbitrary object of a given type	TypeName :: InstanceMethodName	s -> s.toString() s -> s.toLowerCase() s -> s.length() (i1,i2) -> i1.compareTo(i2) (s1,s2) -> s1.compareTo(s2)	String :: toString String :: toLowerCase String :: length Integer :: compareTo String :: compareTo

Method reference Examples of Particular Object Vs an Arbitrary Object

The `Collectors.groupingBy()` method has three overloads within the `Collectors` class - each building upon the other. We'll cover each one in the proceeding sections.

Collectors.groupingBy() with a Classification Function

The first variant of the `Collectors.groupingBy()` method takes only one parameter - a classification function. Its syntax is as follows:

```
Map<String, List<Student>> studentsBySubject = students .stream() .collect(
Collectors.groupingBy(Student::getSubject) );
```

Collectors.groupingBy() with a Classification Function and Downstream Collector

When just grouping isn't quite enough - you can also supply a downstream collector to the `groupingBy()` method:

As the downstream here we'll be using `Collectors.mapping()` method, which takes 2 parameters:

```
Map<String, List<String>> studentsByCity = students.stream()
.collect(Collectors.groupingBy( Student::getCity,
Collectors.mapping(Student::getName, Collectors.toList())));
System.out.println(studentsByCity);
```

Instead of reducing students to their names, we can reduce lists of students to their counts, for instance, which can easily be achieved through `Collectors.counting()` as a wrapper for a reduction operation:

```
Map<Integer, Long> countByAge = students.stream()

            .collect(Collectors.groupingBy(

              Student::getAge,

              Collectors.counting()));
```

Collectors.groupingBy() with a Classification Function, Downstream Collector and Supplier

The third and final overloaded `groupingBy()` method variant takes the same two parameters as before, but with the addition of one more - a *supplier method*.

Map<String, List<String>> namesByCity = students.stream()
.collect(Collectors.groupingBy(Student::getCity, TreeMap::new,
Collectors.mapping(Student::getName, Collectors.toList())));

Sorting a List of Employee

```
lis.stream()
         .sorted(Comparator.comparing(User::getId))
         .collect(Collectors.toList())
         .forEach(System.out::println);

lis.stream()
.sorted(Comparator.comparing(User::getId,Comparator.re
verseOrder()))
         .collect(Collectors.toList())
         .forEach(System.out::println);
```

listOfBooks.sort(Comparator.comparing((Book b) -> b.getAuthor())
.thenComparing((Book b) -> b.getPrice())

```
Comparator<Student> nameAndAgeComparator

              = Comparator.comparing(Student::geFirsttName)

                            .thenComparing(Student::geFirsttName)

                       .thenComparing(Student::getAge,

                                 Comparator.reverseOrder());
```

Sorting a Map of Key-value pair

```
Comparator<Entry<User, String>> comparingByKey =
Map.Entry.comparingByKey(Comparator.comparing(User::getNa
me,Comparator.reverseOrder()));

Comparator<Entry<User, String>> comparingByKey2 =
Map.Entry.comparingByKey(Comparator.comparing(User::getId).
reversed());



map.entrySet()

.stream()
.sorted(comparingByKey.thenComparing(comparingByKey2))
        .collect(Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue));
```