CitiusTech

Lambda Method Enhancements

Session 3

Version 1.0

January 2019

CitiusTech has prepared the content contained in this document based on information and knowledge that it reasonably believes to be reliable. Any recipient may rely on the contents of this document at its own risk and CitiusTech shall not be responsible for any error and/or omission in the preparation of this document. The use of any third party reference should not be regarded as an indication of an endorsement, an affiliation or the existence of any other kind of relationship between CitiusTech and such third party

Agenda

- **Default methods**
- Default Methods over Abstract Classes
- Multiple Inheritance for Default Methods
- Static methods
- Points for Static Methods

Default Methods (1/6)

- Java 8 introduces "**Default Method**" or (Defender methods) new feature, which allows developer to add new methods to the interfaces without breaking the existing implementation of these interface.
- It provides flexibility to allow interface define implementation which will use as default in the situation where a concrete class fails to provide an implementation for that method.
- Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Default Methods (2/6)

- Following code snippet shows an interface named TimeClient:

```
public interface TimeClient {
   void setTime(int hour, int minute, int second);
   void setDate(int day, int month, int year);
   void setDateAndTime(int day, int month, int year, int hour, int minute, int
                                                                        second);
   LocalDateTime getLocalDateTime();
}
```

- Suppose that you want to add new functionality to the **TimeClient** interface, such as the ability to specify a time zone through a **getZondDateTime()**.
- Now, in case, if you modify the **TimeClient** interface, you will have to modify the class implementing the interface.
- However, rather than leaving **getZonedDateTime()** as abstract, we can define a **default implementation**.

Default Methods (3/6)

- Following code snippet shows the default method definition in the **TimeClient** interface:

```
public interface TimeClient {
   void setTime(int hour, int minute, int second);
   void setDate(int day, int month, int year);
   void setDateAndTime(int day, int month, int year, int hour, int minute, int
                                                                        second);
   LocalDateTime getLocalDateTime();

// default method
default ZonedDateTime getZonedDateTime(String zoneString) {
      return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
 }
}
```

- You can specify that a method definition in an interface is a default method with the **default** keyword at the beginning of the method signature.
- All method declarations in an interface, including default methods, are **implicitly public**, so you can omit the public modifier.

Default Methods (4/6)

- When you extend an interface that contains a default method, you can do the following:
  - Do not mention the default method at all, which lets your extended interface inherit the default method.
  - Re-declare the default method, which makes it abstract.
  - Redefine the default method, which overrides it.
- When you extend the interface with default method, the extended interface will get the default implementation.
- Example:

```
public interface AnotherTimeClient extends TimeClient { }
```

- Any class that implements the interface **AnotherTimeClient** will have the implementation specified by the default method **TimeClient.getZonedDateTime()**.

Default Methods (5/6)

- Suppose that you extend the interface **TimeClient** as follows:

```
public interface AbstractZoneTimeClient extends TimeClient {

   public ZonedDateTime getZonedDateTime(String zoneString);
}
```

- Any class that implements the interface **AbstractZoneTimeClient** will have to implement the method **getZonedDateTime()**.
- This method is an abstract method like all other non-default (and non-static) methods in an interface.

Default Methods (6/6)

- For Java 8, the JDK collections have been extended and **forEach** method is added to the entire collection (which work in conjunction with lambdas).

```
public interface Iterable<T>
{
  default void forEach(Consumer<? super T> action)
 {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
 }

}
```

Agenda

- Default methods
- **Default Methods over Abstract Classes**
- Multiple Inheritance for Default Methods
- Static methods
- Points for Static Methods

When to use Default Method over Abstract Classes?

- It seems that interfaces and abstract classes are same. However, they are still different concept in Java 8.
- Abstract class can define constructor. They are more structured and can have a state associated with them.
- While in contrast, *default method* can be implemented only in the terms of invoking other interface methods, with no reference to a particular implementation's state.
- The good thing about this new feature is that, before you were forced to use an abstract class for the convenience methods, thus constraining the implementer to single inheritance, now you can have a really clean design with just the interface and a minimum of implementation effort forced on the programmer.
- Hence, both are used for different purposes and choosing between them really depends on the scenario.

Agenda

- Default methods
- Default Methods over Abstract Classes
- **Multiple Inheritance for Default Methods**
- Static methods
- Points for Static Methods

Multiple Inheritance with Default Methods (1/2)

- As java class can implement multiple interfaces and each interface can define default method with same method signature, therefore, the inherited methods can conflict with each other.

```
Interface InterfaceA{
   default void defaultMethod() {
      System.out.println("Interface A Default Method");
   }
}

Interface InterfaceB{
    default void deafultMethod() {
       System.out.println("Interface B Default Method");
   }
}

class Impl implements InterfaceA, InterfaceB {
}
```

- The above code will fail to compile with the following error,

  java: class Impl inherits unrelated defaults for defaultMethod() from types InterfaceA and InterfaceB

Multiple Inheritance with Default Methods (2/2)

- In order to fix this class, we need to provide *default method* implementation:

```
class Impl implements InterfaceA, InterfaceB {
    @Override
     public void defaultMethod(){
     }
}
```

- Further, if we want to invoke default implementation provided by any of the super interface rather than our own implementation, we can achieve it as follows,

  InterfaceA.super.defaultMethod();

Agenda

- Default methods
- Default Methods over Abstract Classes
- Multiple Inheritance for Default Methods
- **Static methods**

Static Methods (1/2)

- In addition to default methods, you can define static methods in interfaces.
- Java interface static method is similar to default method except that we can't override them in the implementation classes.
- This feature helps us in avoiding undesired results incase of poor implementation in implementation classes.
- This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class.
- Following code snippet demonstrates the static method:

```
public interface MyData {
 default void print(String str) {
     if (!isNull(str))
          System.out.println("MyData Print::" + str);
   }

 static boolean isNull(String str) {
  System.out.println("Interface Null Check");
   return str == null ? true : "".equals(str) ? true : false;
  }
}
```

Static Methods (2/2)

- Following code snippet shows an implementation class that is having **isNull()** method with poor implementation:

```
public class MyDataImpl implements MyData {
  public boolean isNull(String str) {
     System.out.println("Impl Null Check");
    return str == null ? true : false;
  }

   public static void main(String args[]){
                MyDataImpl obj = new MyDataImpl();
                obj.print("");
                obj.isNull("abc");
   }
}
// prints:
            Interface Null Check
            Impl Null Check
```

- **isNull(String str)** is a simple class method, it's not overriding the interface method.
- Adding **@Override** annotation will result in compiler error.

Points for Static Method

- Java interface static method is part of interface, we can't use it for implementation of class objects.
- Java interface static methods are good for providing utility methods, for example null check, collection sorting, and so on.
- Java interface static method helps us in providing security by not allowing implementation classes to override them.
- We can't define interface static method for Object class methods, we will get compiler error as "This static method cannot hide the instance method from Object".
- This is because it's not allowed in java, since Object is the base class for all the classes and we can't have one class level static method and another instance method with same signature.
- We can use java interface static methods to remove utility classes such as Collections and move all of it's static methods to the corresponding interface, that would be easy to find and use.

Thank You

CitiusTech Markets

CitiusTech Services

CitiusTech Platforms

Accelerating Innovation

CitiusTech Contacts

Email ct-univerct@citiustech.com

www.citiustech.com