Java8

## Interface changes

Type of combability:

* Binary – the applications work after compiling with changes. Adding new method to existing interfaces breaks this.
* Source – the source code works with the changes. Adding new method to existing interfaces breaks this.
* Behaviour – The output of the application on the same input remains same even after applying the change. Adding new method to existing interfaces does not break this because the new method is not called from existing code.

Why default methods are introduced?

The main reason is backward compatibility. Adding new method in a existing interface means all other projects using the interface needs to be modified by providing a new implementation for the new method(s). This is something java developers can’t afford as the products/projects running on Java is huge. So, they comeup with the idea of having default methods in interfaces.

It helps is achieving additional functionality of interfaces with the need to modify their implementation classes. This is huge advantage for library or framework developers.

Changes in Java8 in terms of interfaces

* Default method
  + Sort() : List interface
  + Stream
  + spliterator() – Collection interface
* Static method
  + Comparator.naturalOrder
  + They may replace the companion or utility classes (having only static methods)
    - Collections class

|  |  |
| --- | --- |
| Abstract | Interface |
| Only one abstract class is allowed to extent | Multiple interfaces can be implemented by a class |
| It can contain instance fields | Fields are not possible. By the default the variables defined in an interface is public static final. |

Default method use cases

* Optional methods –
  + In case we implement an interface and u don’t require all the functionalities. You still have to give a concrete implementation (empty code block for examples)
  + Default methods reduces these boilerplate code and helps developer to implement the functionality they need.
* Multiple inheritance behaviour –
  + Reuse code from multiple classes
  + Class can inherit the default behaviour of multiple classes

Why inheritance is bad ?

Inheriting from a class with 100 methods just to reuse 1 method is a bad idea. Use aggregation/composition technique.

* Classes are marked as final just to avoid this kind of scenarios.

what if a class implements two interfaces that have the same default method signature?

**Rules**

* Classes always win – in case of conflict jvm chooses the class or its super class method
* Sub-interface wins – if multiple interfaces are implemented and conflict is detected in super and derived interfaces then sub interface method is invoked
* If the choice is still ambiguous the developer should create a method and call the desired method explicitly (compile time error is shown - Error: class C inherits unrelated defaults for hello() from types B and A.).

Solution:

* Implement the method and call the method from the interface you want to inherit the behaviour using <<interface>>.super.<<method>> syntax.

Dimond problem:

Case 1: In this case the resolution is easy since method is declared in A.

|  |
| --- |
| interface A {  default void print() {  System.*out*.println("A");  } } interface B extends A {  } interface C extends A {  }  class D implements B, C {  public static void main(String[] args) {  D d = new D();  d.print();  } } |

Case 1: What if B/C also implements the method. In this case the resolution follows Rule#2 and invokes B->print.

|  |
| --- |
| interface A {  default void print() {  System.*out*.println("A");  } } interface B extends A {  default void print() {  System.*out*.println("A");  }  } interface C extends A {  }  class D implements B, C {  public static void main(String[] args) {  D d = new D();  d.print();  } } |

Case 1: What if B and C botj implements the method. In this case the resolution follows Rule#3.

|  |
| --- |
| interface A {  default void print() {  System.*out*.println("A");  } } interface B extends A {  default void print() {  System.*out*.println("A");  }  } interface C extends A {  default void print() {  System.*out*.println("A");  } }  class D implements B, C {  void print() {  Super.B.print();  }  public static void main(String[] args) {  D d = new D();  d.print();  } } |

Following are the key concepts you should take away from this chapter:

 Interfaces in Java 8 can have implementation code through default methods and static methods.

 Default methods start with a default keyword and contain a body like class methods do.

 Adding an abstract method to a published interface is a source incompatibility.

 Default methods help library designers evolve APIs in a backward-compatible way.

 Default methods can be used for creating optional methods and multiple inheritance of behavior.

 There are resolution rules to resolve conflicts when a class inherits from several default methods with the same signature.

 A method declaration in the class or a superclass takes priority over any default method declaration. Otherwise, the method with the same signature in the most specific default-providing interface is selected.

 When two methods are equally specific, a class can explicitly override a method and select which one to call.