HowToDoInJava

Multiple Inheritance in Java

Last Updated: January 29, 2022 | By: Lokesh Gupta | Java Object Oriented Programming | Inheritance

As we have long learned the fact that ***multiple inheritance*** *is not directly supported* in Java, well that was only till Java 7. In Java 8, we can realize the concept of **multiple inheritance** through use of default methods without getting into **diamond problem**.

Let's see how?

Table of Contents

1. What are default methods?

If you been in java programming since enough long time, you may realize that how painful can be adding a new method in an existing interface. You will need to implement that new method in java classes which implement that interface. It's really difficult job. Well, java 8 brought **default methods** to help you in exactly same situation.

> Default methods enable you to add new functionality to the interfaces and ensure backward compatibility for existing classes which implement that

> interface.

As their name implies, default methods in interfaces are methods which will be **invoked by default – if not overridden** in implementing classes. Let's understand with an example.

`Moveable` interface is some existing interface and wants to add a new method `moveFast()`. If it adds `moveFast()` method using old technique, then all classes implemeting `Moveable` will also be changed. So, let's add `moveFast()` method as default method.

```
Moveable.java

public interface Moveable
{
    default void moveFast()
    {
        System.out.println("I am moving fast, buddy !!");
    }
}
```

If all classes implementing `Moveable` interface do not need change themselves (until some class specifically wants to override `moveFast()` method to add custom logic). All classes can directly call `instance.moveFast()` method.

```
Animal.java

public class Animal implements Moveable
{
    public static void main(String[] args)
    {
        Animal tiger = new Animal();

        //Call default method using instance reference
        tiger.moveFast();
    }
}
```

2. How multiple inheritance is achieved via default methods?

> Multiple inheritance is a feature of some object-oriented computer programming languages in which an object or class can inherit characteristics and behavior from more than one parent object or parent class.

We know that in java (until jdk 7), inheritence in java was supported by `extends` keyword which is used to create a child class from a parent class. You cannot extend from two classes.

Until java 7, interfaces were only for declaring the contracts which implementing classes MUST implement (except the implementing class in not `abstract` itself). So there was no specific behavior attached with interfaces which a class can inherit. So, even after a class was capable of implementing as many interfaces as it want, it was not appropriate to term as multiple inheritance.

But since java 8's default methods, interfaces have behavior as well. So now **if a class implement two interfaces and both defines default methods, then it is essentially inheriting behaviors from two parents which is multiple inheritance**.

For example, in below code `Animal` class does not define any of it's own behavior; rather it is inheriting behavior from parent interfaces. That's multiple inheritance.

```
package com.howtodoinjava.examples;

interface Moveable
{
    default void moveFast(){
        System.out.println("I am moving fast, buddy !!");
    }
}

interface Crawlable
{
    default void crawl(){
        System.out.println("I am crawling !!");
    }
}

public class Animal implements Moveable, Crawlable
{
```

```
    public static void main(String[] args)
    {
        Animal self = new Animal();

        self.moveFast();
        self.crawl();
    }
}
```

3. Possible conflicts during mutiple inheritance

In above example, we have two different interfaces and two different methods – so there is no conflict. What if **both interfaces decide to define a new method with same name**. Well they can define without problem. But which method will be invoked when `Animal` instance will call it's name.

That's conflicting situation.

```
package com.howtodoinjava.examples;

interface Moveable
{
    default void run(){
        System.out.println("I am running, kid !!");
    }
}

interface Crawlable
{
    default void run(){
        System.out.println("I am running, daddy !!");
    }
}

public class Animal implements Moveable, Crawlable
{
    public static void main(String[] args)
    {
        Animal self = new Animal();

        //What will happen when below statement will execute
        //self.run();
    }
```

```
}
```

So solve above conflict, caller class must decide which `run()` method it want to invoke and then call **using interface's reference** like below.

```
Moveable.super.run();    //Call Moveable's run() method

//or

Crawlable.super.run();   //Call Crawlable's run() method
```

That's all you should know about multiple inheritance feature Java 8, using default methods.

Happy Learning !!

Recommended Reading:

1. Guide to Inheritance
2. Java group by sort – multiple comparators example
3. Chaining Multiple Predicates in Java
4. Java Stream reuse – traverse stream multiple times?
5. Sorting a Stream by Multiple Fields in Java

6. Applying Multiple Conditions on Java Streams
7. Java Enum with Multiple Values
8. Control concurrent access to multiple copies of a resource using Semaphore
9. Spring mvc multiple file upload with progress bar in ajax and jquery
0. Spring Batch MultiResourceItemReader – Read Multiple CSV Files Example

10 thoughts on “Multiple Inheritance in Java”

Jb

October 7, 2016 at 4:13 am

Even this code (note how I took the calls out of the static method by adding a Test class) still fails to pass the IDE’s error checking and you get an error on the Animal class with something like:

Duplicate default methods named run with the parameters () and () are inherited from the types Crawlable and Moveable

```
package com.foo.bar;



  interface Moveable
  {
      default void run(){
          System.out.println("I am running, kid !!");
      }
  }

  interface Crawlable
  {
      default void run(){
          System.out.println("I am running, daddy !!");
      }
```

```
}

public class Animal implements Moveable, Crawlable
{
    public void move() {
      Moveable.super.run();
        Crawlable.super.run();
    }
}

public class Test
{
    public static void main(String[] args)
    {
      Animal animal = new Animal();
      animal.move();
        //What will happen when below statement will execute
        //self.run();
    }
}
```

Reply

Lokesh Gupta

October 7, 2016 at 10:36 am

> Yeh, it's problem that `class.super.method()` solve.

Reply

jb

October 11, 2016 at 2:39 am

> Maybe I misunderstand, but class.super.method() does not solve the problem in my example above. I still get errors and I'm not in a static context…

It's a cool idea and I'd like to understand. To help me, please post some code that will not generate errors either in the IDE or at compile time? Thanks!

Reply

Lokesh Gupta

October 11, 2016 at 10:20 am

```
interface Moveable
{
    default void run(){
        System.out.println(&amp;quot;I am running, kid !!&amp;quot;);
    }
}

interface Crawlable
{
    default void run(){
        System.out.println(&amp;quot;I am running, daddy !!&amp;quot;);
    }
}

public class Animal implements Moveable, Crawlable
{
    public static void main(String[] args)
    {
        Animal self = new Animal();
        self.run();
    }

    public void run()
    {
      Moveable.super.run();
    Crawlable.super.run();
  }
}
```

Reply

jb

October 12, 2016 at 5:29 am

> Many thanks! I see my error now – not having a method named “run()” to match the interface.
>
> The error message generated by that mistake was not helpful. It referenced “duplicate default methods” when in fact, the problem was the lack of a run() method in my Animal class…
>
> Great example – thanks for posting!

Jb

October 7, 2016 at 3:52 am

> This sounded great until I tried to put it into an IDE. Unless I made a mistake, the following code will not compile but will instead produce the error: “Cannot use super in a static context”

```
package com.foo.bar;

  interface Moveable
  {
      default void run(){
          System.out.println("I am running, kid !!");
      }
  }

  interface Crawlable
  {
      default void run(){
          System.out.println("I am running, daddy !!");
      }
  }
```

```
public class Animal implements Moveable, Crawlable
{
    public static void main(String[] args)
    {
        Animal self = new Animal();
        Moveable.super.run();
        Crawlable.super.run();
    }
}
```

Please let me know if I got it wrong…

Reply

Lokesh Gupta

October 7, 2016 at 10:33 am

You are right. My point is to use `Moveable.super.run()` style method calls to avoid conflict. You can definitely use it outside static methods. Or you can test by removing static keyword from main mathod.

Reply

Aurangzeb khan

September 19, 2016 at 10:31 pm

This article very important for java developer and java new concepts

Reply

Dieblich

September 4, 2015 at 12:12 pm

It goes even further…
Peter Verhas showed that you could use the default methods to induce attributes:
https://javax0.wordpress.com/2014/04/02/how-not-to-use-java-8-default-methods/

I tried to adopt his idea for observability, but it sees somehow “wrong”
https://codereview.stackexchange.com/questions/102596/observe-closeable-with-default-methods

Reply

Lokesh Gupta

September 4, 2015 at 2:17 pm

Sorry, I am making a comment without knowing complete context. Have you considered Closeable and Observable separate; rather than try to combine them?

Reply

Leave a Comment

Name *

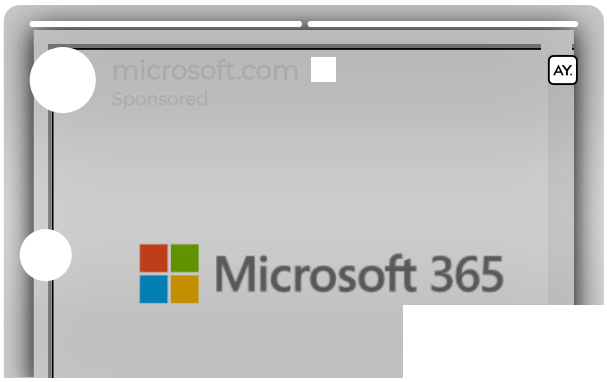
Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...




See More

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial