**13 Rules about Interfaces in Java**

### 1) Fields in an interface are public, static and final implicitly:

### 2) Methods in an interface are public implicitly:

### 3) A class can implement multiple interfaces:

### 4) The overriding methods cannot have more restrict access specifiers:

### 5) Non-abstract classes must override all methods declared in the super interfaces:

### 6) Abstract classes are not forced to override all methods from their super interfaces. The first concrete class in the inheritance tree must override all methods:

Let’s see an example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | interface Animal {      void eat();      void sleep();  }  abstract class Felidae implements Animal {      public void sleep() {          // sleeping      }        // this abstract class doesn't override the eat() method  }    class Cat extends Felidae {      public void eat() {          // cat eats...      }  } |

Here, the Felidae class is abstract so that it is not forced to override all methods from the Animal interface. However, Cat is a concrete class so it must override the remaining method eat() from the super interface.

### 7) An interface cannot extend another class

### 8) An interface can extend from multiple interfaces:

Let’s see an example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | interface Animal { }    interface Predator extends Animal { }    interface Herbivore extends Animal { }    interface Human extends Predator, Herbivore { } |

### 9) An interface can be nested within a class:

For example:

|  |  |
| --- | --- |
| 1  2  3 | class A {      interface B { }  } |

### 10) An interface can be nested within another interface:

For example:

|  |  |
| --- | --- |
| 1  2  3 | interface C {      interface D { }  } |

### 11) Earlier, Methods in an interface cannot be static and final

### 12) Since Java 8, an interface can have default methods and static methods:

The purpose of default methods is to make changes to existing interfaces without breaking its existing subclasses – for backward compatibility. That means you should not use default methods for completely new code. Use default methods to make changes to old code - existing interfaces – to avoid changes to existing implementation classes.

When a class implements an interface that contains default methods, it can:

* Do nothing: inherit the default methods from the super interface.
* Re-declare the default methods as abstract – causing its subclasses to implement them.
* Override the default methods with new implementation.

### 13) Functional interface is an interface that has only one method:

## Default method and multiple inheritance in Java:

There can be a case in which a class implements two different interfaces having the same default method, for example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public interface X {      void foo();        default void bar() {          // code      }  }    public interface Y {      void doo();        default void bar() {          // code      }  }    public class XYImpl implements X, Y {    } |

In this case, to avoid ambiguity, the sub class must override the common default method, for example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class XYImpl implements X, Y {      public void foo() {          // implement from X      }        public void doo() {          // implement from Y      }        public void bar() {          // override from X, Y      }  } |

## Default method and static method in interface in Java:

Also since Java 8, we can write static methods (with code body) in an interface. Imagine an interface has two default methods that use a same portion of code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public interface ABC {        void foo(); // abstract method        default void bar() {          // code snippet #1            // other bar's code      }        default void doo() {          // code snippet #1            // other doo's code      }  } |

Here, both the default methods bar() and doo() uses a same portion of code code snippet #1. We want to avoid code duplication – so how can we do?

Introduce a new default method that contains only the code to be reused? That’s possible. But the intermediate default method is also inherited by subclasses – adding more complexity to the inheritance tree - which we don’t want.

So instead of using an intermediate default method, we can write the shared code in a static method like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public interface ABC {        void foo(); // abstract method        static void code1() {          // code snippet #1      }        default void bar() {            code1();            // other bar's code      }        default void doo() {          code1();            // other doo's code      }    } |

Since [static methods](https://www.codejava.net/java-core/the-java-language/what-is-static-method-in-java) do not belong to instances of class and are not inherited, they are perfect to be reused by default methods, as seen in the above example.

**Overloading an abstract Method Declaration**

interface PointInterface {

void move(int dx, int dy);

}

interface RealPointInterface extends PointInterface {

void move(float dx, float dy);

void move(double dx, double dy);

}

*Here, the method named move is overloaded in interface RealPointInterface with three different signatures, two of them declared and one inherited. Any non-abstract class that implements interface RealPointInterface must provide implementations of all three method signatures.*

**Ambiguous Inherited Fields**

*If two fields with the same name are inherited by an interface because, for example, two of its direct superinterfaces declare fields with that name, then a single ambiguous member results. Any use of this ambiguous member will result in a compile-time error. In the program:*

interface BaseColors {

int RED = 1, GREEN = 2, BLUE = 4;

}

interface RainbowColors extends BaseColors {

int YELLOW = 3, ORANGE = 5, INDIGO = 6, VIOLET = 7;

}

interface PrintColors extends BaseColors {

int YELLOW = 8, CYAN = 16, MAGENTA = 32;

}

interface LotsOfColors extends RainbowColors, PrintColors {

int FUCHSIA = 17, VERMILION = 43, CHARTREUSE = RED+90;

}

*the interface LotsOfColors inherits two fields named YELLOW. This is all right as long as the interface does not contain any reference by simple name to the field YELLOW. (Such a reference could occur within a variable initializer for a field.)*

*Even if interface PrintColors were to give the value 3 to YELLOW rather than the value 8, a reference to field YELLOW within interface LotsOfColors would still be considered ambiguous.*

### Inheritance and Overriding

interface Top {

default String name() { return "unnamed"; }

}

interface Left extends Top {

default String name() { return getClass().getName(); }

}

interface Right extends Top {}

interface Bottom extends Left, Right {}

*Right inherits name() from Top, but Bottom inherits name() from Left, not Right. This is because name() from Left overrides the declaration of name() in Top.*

An interface does not inherit static methods from its superinterfaces.

1) To achieve security - hide certain details and only show the important details of an object (interface).