Java 8

Mejoras de la versión.

Temario

francisco.philip@gmail.com

Mejoras

Métodos *default* y *static* en interfaces.

forEach en la interfaz *Iterable*

Interfaces funcionales

Java Time API

Java Stream API

Mejoras en colecciones

Métodos por defecto

francisco.philip@gmail.com

Introducción

Hasta la versión 7, por definición, las interfaces eran clases con todos sus métodos públicos y abstractos. Si bien esto es una limitación, se viene desarrollando código sin inconvenientes.

Con la versión 8, el uso de lambda y la inclusión de métodos como o *forEach* o *stream* motivaron a una mejora, avanzando hacia algo parecido a la herencia múltiple.

francisco.philip@gmail.com

Diferencias con las clases abstractas

Las interfaces no tienen estados.

Las clases abstractas pueden

- métodos *protected*
- tener un estado asociado.

francisco.philip@gmail.com

Interfaces

```
public interface Interface1 {
    void method1(String str);
    default void log(String str){
        System.out.println("I1 logging::"+str);
        print(str);
    }
}
```

```
public interface Interface2 {
    void method2();
    default void log(String str){
        System.out.println("I2 logging::"+str);
    }
}
```

francisco.philip@gmail.com

Implementación

```
public class MyClass implements Interface1, Interface2 {
    @Override public void method2() {
    }
    @Override public void method1(String str) {
    }
 @Override public void log(String str){
        System.out.println("MyClass logging::"+str);
        Interface1.print("abc");
    }
}
```

francisco.philip@gmail.com

Problemas comunes.

```
interface A {
    void method();
}

interface B extends A {
    @Override
    default void method() {
        System.out.println("B");
    }
}
```

```
interface C extends A {
    @Override
    default void method() {
        System.out.println("C");
    }
}
interface D extends B, C {
 }
```

francisco.philip@gmail.com

Solución

```
interface D extends B, C {
    @Override
    default void method() {
        B.super.method();
    }
}
```

francisco.philip@gmail.com

Conclusiones (I)

- Los métodos default pueden ayudarnos a extender interfaces garantizando funcionalidades en las implementaciones.
- Los métodos por defecto funcionan como puente por las diferencias entre las interfaces y clases abstractas.
- Los métodos por defecto nos ayudarán a evitar clases de utilidad, como la clase *Collections* puede ser proporcionada en la propia interfaz.
- Los métodos por defecto nos ayudará en la eliminación de clases de implementación de base, podemos proporcionar implementación por defecto y las clases de implementación pueden elegir cuál de ellos para anular.

francisco.philip@gmail.com

Conclusiones (II)

- Una de las principales razones para la introducción de métodos por defecto es para mejorar la API Colecciones en Java 8 para apoyar las expresiones lambda.
- Los métodos predeterminados también se conocen como *Defender Method* o *Virtual Extension Method*.

Métodos por defecto

Interface static methods

francisco.philip@gmail.com

Definición

Los métodos estáticos son similares a los métodos *default,* salvo que no pueden ser sobrescritos en las implementaciones. Esta funcionalidad nos ayuda a evitar resultados no deseados en las clases.

francisco.philip@gmail.com

Ejemplo

```
public interface MyConsole {
    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;
    }
}
```

francisco.philip@gmail.com

Ejemplo

```
public class MyConsole Impl implements MyConsole {
    public boolean isNull(String str) {
        System.out.println("Impl Null Check");
       return str == null ? true : false;
    }
    public static void main(String args[]){
        MyConsole  obj = new MyConsoleImpl();
        obj.print("");
        obj.isNull("abc");
    }
}
```

Ahora vamos a ver una clase de implementación que está teniendo método *isNull ()* con una mala aplicación.

Como vemos, *isNull(String str)* es un método simple de clase, que no sobrescribe el método de la interfaz. Si sumamos, @Override annotation a este método, el compilara nos reportara errores.

francisco.philip@gmail.com

Salida

Podemos ver que si se ejecuta, obtendremos la siguiente salida

```
Interface Null Check

Impl Null Check
```

Y si la pasamos el método de static a default, la respuesta es la siguiente

```
Impl Null Check

MyConsole Print::

Impl Null Check
```

francisco.philip@gmail.com

Limitaciones

Los métodos estáticos son visibles solamente en la interfaz, si removemos el método isNull() de la clase MyConsoleImpl, no es posible usarlo por el en MyConsoleImpl, sin embargo se puede acceder usando el nombre de la clase.

```
boolean result = MyConsole.isNull("abc");
```

francisco.philip@gmail.com

Conclusiones (I)

- Los métodos estáticos de interfaz son parte de la interfaz, no podemos usarlo para objetos de clase que las implemente implementación.
- Los métodos estáticos de interfaz son buenos para proporcionar métodos de utilidad, por ejemplo cheque nulo, colección de clasificación, etc.
- Los métodos estáticos de interfaz nos ayudan a proveer de seguridad no permitiendo la sobrescritura en la implementación.
- No podemos definir métodos estáticos para métodos de la clase Object, ya que obtendremos error de compilación como "Este método estático no puede ocultar el método de instancia del objeto". Esto se debe a que no está permitido en java, ya que Object es la clase base de todas las clases y no podemos tener método estático un nivel de clase y otro método de instancia con la misma firma

francisco.philip@gmail.com

Conclusiones (II)

- Podemos utilizar métodos de interfaz estáticos para eliminar clases de utilidad, como las colecciones y mover todos es los métodos estáticos a la interfaz correspondiente, que sería fácil de encontrar y utilizar.

Método *forEach* en la *Iterable.java*

francisco.philip@gmail.com

Definición

Java 8 introduce el método *forEach* en la interfaz *java.lang.Iterable* que nos permite enfocarnos en el negocio solamente.

Mediante *java.util.function.Consumer* como argumento del método nos permite reutilizar la lógica de negocio.

Iterable.java

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

francisco.philip@gmail.com

Uso

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

```
myList.forEach(
   new Consumer<Integer>() {
       public void accept(Integer t) {
           System.err.println(t);
       }
```

Interfaces Funcionales

Functional Interface

francisco.philip@gmail.com

Definición

Una interfaz que tiene solo método abstracto puede ser llamada como interfaz funcional

francisco.philip@gmail.com

Definición

Conceptualmente, una interfaz funcional tiene exactamente un método abstracto.

Los casos de interfaces funcionales se pueden crear con las expresiones lambda, referencias de métodos, o referencias de constructor.

francisco.philip@gmail.com

Definición

```
@FunctionalInterface
```

Un tipo de anotación informativa utiliza para indicar que una declaración de tipo de interfaz está destinada a ser una interfaz funcional tal como se define por la especificación del lenguaje Java.

francisco.philip@gmail.com

Ejemplos

```
interface Foo {
          boolean equals(Object obj);
}
```

No es una interfaz funcional, ya que *equals* es un miembro de la clase *Object*

francisco.philip@gmail.com

Ejemplos

```
interface Comparator<T> {
   boolean equals(Object obj);
   int compare(T o1, T o2);
}
```

Al definir en la interfaz un método inherente a la clase Object, se convierte en una interfaz funcional.

francisco.philip@gmail.com

Ejemplos

```
interface X { int m(Iterable<String> arg); }

interface Y { int m(Iterable<String> arg); }

interface Z extends X, Y {}
```

Al ser 2 métodos con la misma firma, es una interfaz funcional

francisco.philip@gmail.com

Ejemplos

```
interface X {
        int m(Iterable<String> arg); }
interface Y {
        int m(Iterable<Integer> arg); }
interface Z extends X, Y {}
```

No es una interfaz funcional, poseen 2 métodos diferentes.

francisco.philip@gmail.com

Ejemplos

```
interface X {
    int m(Iterable<String> arg, Class c); }
interface Y {
    int m(Iterable arg, Class<?> c); }
interface Z extends X, Y {}
```

Al ser 2 métodos con argumentos diferentes, no es una interfaz funcional.

francisco.philip@gmail.com

Ejemplos

```
interface X { long m(); }

interface Y { int m(); }

interface Z extends X, Y {}
```

Error de compilación, ya que no se pueden tener 2 métodos idénticos con clase de retorno de diferente tipo.

francisco.philip@gmail.com

Ejemplos

```
interface Foo<T> { void m(T arg); }

interface Bar<T> { void m(T arg); }

interface FooBar<X, Y> extends Foo<X>, Bar<Y> {}
```

Error de compilación, las firmas del método es diferente.

package java.util.function

francisco.philip@gmail.com

Clases del paquete

Consumer<T> Operación que acepta un argumento y no retorna valor.

Function<T,R> Función que acepta un argumento T y produce un resultado.

Supplier<T> Proveedor de objetos del tipo T.

Predicate<T> Representa un predicado de un solo argumento (Boolean-valued function).

francisco.philip@gmail.com

Clases del paquete

BiConsumer<T,U> Operación que acepta dos argumentos, y no retorna valor.

BiFunction<T,U,R> Función que acepta dos argumentos y produce resultado.

BinaryOperator<T> Operación recibe dos operadores del mismo tipo y produce uno del mismo tipo.

BiPredicate<T,U> Predicado (Boolean-valued function) de dos argumentos.

Gracias