

# 214-08-2.Java8::言語仕様変更

デフォルトメソッド、インタフェースのstaticメソッド、  
メソッド参照、実質的final

デフォルトメソッド

インタフェースのstaticメソッド

メソッド参照

実質的final

関数型プログラミングまとめ

## 目的：

- デフォルトメソッドとは何かを学び、その活用方法について学ぶ。
- どういった場合にメソッド参照の記述を行えるか学ぶ。

## ゴール：

- 利点を理解し、インターフェースにデフォルトメソッドを実装することが出来る。
- メソッド参照での実際の処理の流れについて理解し、記述を行うことが出来る。

## デフォルトメソッド

Java8からインタフェースに処理本体が記述できるメソッド（抽象でないメソッド）が定義できるようになった。

### 構文

```
public interface インタフェース名 {  
    アクセス修飾子 default 戻り型 メソッド名(引数,...) {  
        //処理～  
    }  
}
```

デフォルトメソッドだけのインタフェースも記述可能。

デフォルトメソッドをオーバーライドすることも可能。

同じシグネチャー（メソッド名と引数の型が同じ）を持つ別々のインタフェースを継承しようとする、そのままではコンパイルエラーになる。オーバーライドして定義し直すか、どちらかのデフォルトメソッドを呼ぶか定義すればよい。

「親インタフェース名.super.メソッド名」で呼び出し可能。ただし親の親メソッドは呼び出せない。

## 継承の3タイプ

- 状態(state)の多重継承：言語仕様上の選択としてサポートしない
- 実装(implementation)の多重継承：Java8で導入されたデフォルトメソッドでサポート
- 型(type)の多重継承：インタフェース継承でサポート

参照

<https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html>

<https://www.cs.dartmouth.edu/~mckeeman/cs118/references/OriginalJavaWhitepaper.pdf>

## インタフェースのstaticメソッド

Java8からインタフェースにstaticメソッドが記述できるようになった。

### 構文

```
アクセス修飾子 static 戻り型 メソッド名(引数,...) {  
    ~  
}
```

既存のメソッドを(::)で呼び出すことでラムダ式化が出来る。

(関数型インタフェースの変数にメソッドを参照して代入することが出来る。

その場合、引数は書かない)

```
List<String> list = Arrays.asList("a", "b", "c", "d");  
list.forEach(System.out::println);
```

```
//下記の記述をメソッド参照にした  
//list.forEach((str) -> {  
//    System.out.println(str);  
//});
```

List

default void forEach(Consumer<? super T> action)

Consumer

void accept(T t)

## メソッド参照

メソッド単体を参照できる

対象	文法	例
クラスメソッド	クラス名::<クラスメソッド名	String::toString System.out::println
インスタンスメソッド	オブジェクト参照::インスタンスメソッド名	str::length
コンストラクタ	クラス名::new	String::new



## 実質的final

ラムダ式からクラスのフィールドへのアクセスする事ができます。

ローカル変数はfinalな変数のみアクセスする事ができます。

ただし、再代入が行われていないローカル変数はfinalとみなされます。

これを実質的finalといいます。

※エンクロージング環境のローカル変数に匿名関数がアクセスするにはfinalが必要

## 関数型プログラミングで注意する事

### 匿名クラスよりラムダ式を使う

Java8より前は匿名クラスが関数オブジェクトの生成する手段でしたが、Java8以降は匿名クラスを使うメリットはありません。

### ラムダ式よりメソッド参照をつかう

ラムダ式よりも簡潔に記述する事ができる。

### 標準の関数型インタフェースを使う



/\*memo 主な関数型インタフェース \*/

	→ T	→ int	→ long	→ double	→ boolean	→ void	
() →	Supplier<T>	IntSupplier	LongSupplier	DoubleSupplier	BooleanSupplier	Runnable	get getAsInt getAsLong getAsDouble getAsBoolean
	→ R	→ int	→ long	→ double	→ boolean	→ void	
(T) →	Function<T,R> UnaryOperator<T>	ToIntFunction<T>	ToLongFunction<T>	ToDoubleFunction<T>	Predicate<T>	Consumer<T>	run
(int) →	IntFunction<R>	IntUnaryOperator	IntToLongFunction	IntToDoubleFunction	IntPredicate	IntConsumer	
(long) →	LongFunction<R>	LongToIntFunction	LongUnaryOperator	LongToDoubleFunction	LongPredicate	LongConsumer	
(double) →	DoubleFunction<R>	DoubleToIntFunction	DoubleToLongFunction	DoubleUnaryOperator	DoublePredicate	DoubleConsumer	
	→ R	→ int	→ long	→ double	→ boolean	→ void	
(T, U) →	BiFunction<T,U,R> BinaryOperator<T>	ToIntBiFunction<T,U>	ToLongBiFunction<T,U>	ToDoubleBiFunction<T,U>	BiPredicate<T,U>	BiConsumer<T,U>	apply applyAsInt applyAsLong applyAsDouble
(int, int) →		IntBinaryOperator				(T, int) → ObjIntConsumer<T>	
(long, long) →			LongBinaryOperator			(T, long) → ObjLongConsumer<T>	
(double, double) →				DoubleBinaryOperator		(T, double) → ObjDoubleConsumer<T>	test
							accept

Java 8 Functional Interface Naming Guide  
by Esko Luontola, www.orfjackal.net

引用 : <http://blog.orfjackal.net/2014/07/java-8-functional-interface-naming-guide.html>