



Victoria University
of Wellington, New Zealand
*Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa*



SWEN221: Software Development

21: Java 8 (1)

Outline

- Default and Static Interface Methods
- Lambda
- Functional Interfaces

New Interfaces in Java 8

- Before Java 8
 - Only method declaration in interfaces
 - Tough to change interface (change all the classes implementing it)
- After Java 8
 - Allow method implementation in interfaces!
 - Default methods
 - Static methods

Default Methods

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

- Easy to add new methods to interface without changing other classes

Default Methods

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

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

- Two interfaces can have the same default method, but different implementations

Default Methods

```
class MyClass implements Interface1, Interface2 {  
    @Override void method1(String str) { ... }  
    @Override void method2(){ ... }  
    // no need to override default methods, e.g. log(str)  
}
```

- What's the problem?

Default Methods

```
class MyClass implements Interface1, Interface2 {  
    @Override void method1(String str) { ... }  
    @Override void method2(){ ... }  
    // no need to override default methods  
}
```

- Which **default** `log(String str)` to choose from?
- `Interface1` or `Interface2`?
- The common default method(s) must be overridden

Default Methods

```
class MyClass implements Interface1, Interface2 {  
    @Override void method1(String str) { ... }  
    @Override void method2(){ ... }  
    // must override common default method(s)  
    @Override default void log(String str) {  
        System.out.println("MyClass logging:" + str);  
    }  
}
```


Static Methods

- Similar to default methods, except that we **cannot override** them in the implementation classes

What Will Be Printed?

```
interface MyData {  
    default void print(String str) {  
        if (!isNull(str)) System.out.println("MyData:" + str);  
    }  
  
    static boolean isNull(String str) {  
        System.out.println("Interface Null Check");  
        return str == null ? true : "".equals(str) ? true : false;  
    }  
}
```

```
class MyDataImpl implements MyData {  
    boolean isNull(String str) {  
        System.out.println("Impl Null Check");  
        return str == null ? true : false;  
    }  
}
```

```
MyDataImpl obj = new MyDataImpl();  
obj.print("");  
obj.isNull("abc");
```

What Will Be Printed?

A) Interface Null Check
Impl Null Check

B) Interface Null Check
MyData:
Impl Null Check

C) Interface Null Check
Interface Null Check

What Will Be Printed?

A) Interface Null Check
Impl Null Check



B) Interface Null Check
MyData:
Impl Null Check



C) Interface Null Check
Interface Null Check



What Will Be Printed?

```
interface MyData {  
    default void print(String str) {  
        if (!isNull(str)) System.out.println("MyData:" + str);  
    }  
  
    static boolean isNull(String str) {  
        System.out.println("Interface Null Check");  
        return str == null ? true : "".equals(str) ? true : false;  
    }  
}
```

```
Class MyDataImpl implements MyData {  
    boolean isNull(String str) {  
        System.out.println("Impl Null Check");  
        return str == null ? true : false;  
    }  
}
```

```
MyDataImpl obj = new MyDataImpl();
```

```
obj.print("");
```

```
obj.isNull("abc");
```

Interface Null Check

Impl Null Check

Interface vs Abstract Class

- Interface

fields



constructors



privates



many



- Abstract class

fields



constructors



privates



many



Lambdas

- Anonymous single-method class can be unnecessarily long and cumbersome

```
Collections.sort(ls, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.compareToIgnoreCase(s2);  
    }  
});
```

- Use Lambdas to make it more compact

```
Collections.sort(ls, (s1, s2) -> s1.compareToIgnoreCase(s2));
```



Parameters

Method body

Syntax of Lambda Expression

- A comma-separated list of formal parameters in parentheses
 - Can omit the data type of the parameters
 - Can omit the parentheses if only one parameter
- An arrow token "`->`"
- A body
 - A single expression
 - A statement block
 - A void method with no brace

```
s -> s == ""
```

```
s -> {return s == "" ;}
```

```
s -> System.out.println(s)
```


Extensive Use for Event Handler

- Normal

```
JButton b = new JButton("Press Me");  
b.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Button pressed");  
    }  
});
```

- Lambda

```
JButton b = new JButton("Press Me");  
b.addActionListener(  
    e -> System.out.println("Button pressed"));
```

Lambda: More Examples

```
person -> person.getAge()
```


```
(p1,p2) -> p1.getAge() > p2.getAge()
```


```
() -> System.currentTimeMillis()
```

```
(customer, product) -> {  
    if (customer.getAge() < 25 && product.hasAlcohol())  
        return "Please show your ID!"  
    return "Do you need a receipt?"  
}
```

Functional Interfaces

- The following codes have huge repetitions
- Can we make it more compact?

```
static int sum(List<Integer> list) {  
    int res = list.get(0);  
  
    for(int i=1; i<list.size(); i++) res = res  list.get(i);  
    return res;  
}
```

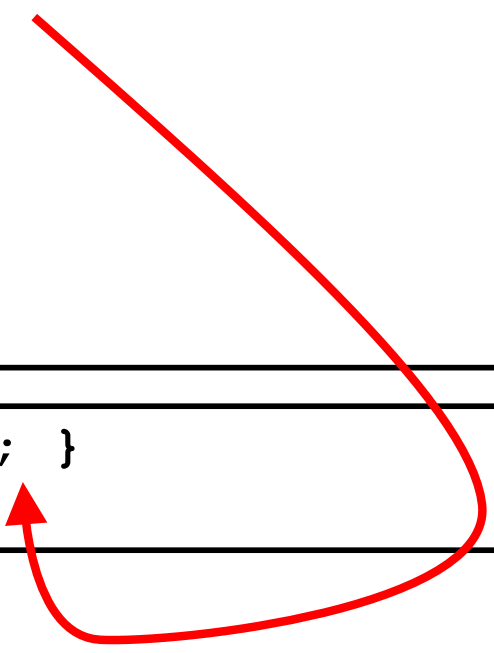
```
static int mul(List<Integer> list) {  
    int res = list.get(0);  
  
    for(int i=1; i<list.size(); i++) res = res  list.get(i);  
    return res;  
}
```

Functional Interfaces

- Parameterise the functions (sum, mul, ...)

```
static int reduce(List<Integer> list, Func<Integer> func) {  
    int res = list.get(0);  
  
    for(int i=1; i<list.size(); i++)  
        res = func.apply(res, list.get(i));  
    return res;  
}
```

```
interface Func<T> { T apply(T a, T b); }
```



- Need another class/interface for reduce()
- Why not put into Func<T>?

Functional Interfaces

```
interface Func<T> {  
    T apply(T a, T b);  
  
    static <T> T reduce(List<T> list, Func<T> func) {  
        T res = list.get(0);  
  
        for(int i=1; i<list.size(); i++)  
            res = func.apply(res, list.get(i));  
        return res;  
    }  
}
```

```
public static void main(String[] args){  
    List<Integer> list = Arrays.asList(2,1,5,7,4,3);  
    System.out.println(Func.reduce(list, (a,b)->a+b));  
    System.out.println(Func.reduce(list, (a,b)->a*b));  
}
```

Functional Interfaces

- Interfaces with **one and only one** abstract method
- Decorated with **@FunctionalInterface**
- Can be represented as a **lambda expression**
- A lot of functional interfaces in Java 8
 - Runnable
 - Callable
 - Comparator
 - ActionListener
 - ...
- *<http://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>*

Example: Function

- A function takes $(T \ t)$ and returns $(R \ r)$

```
interface Function<T,R> {  
    R apply(T t);  
  
    // identity function always returns the input  
    static <T> Function<T,T> identity() { return t -> t; }  
  
    // first apply "before", then apply this function  
    default <V> Function<V,R>  
        compose(Function<? super V,? extends T> before){  
            return (V v) -> apply(before.apply(v));  
        }  
  
    // first apply this function, then apply "after"  
    default <V> Function<T,V>  
        andThen(Function<? super R,? extends V> after){  
            return (T t) -> after.apply(apply(t));  
        }  
}
```

Example: Function

- What does the following output?

```
Function<Integer,Integer> mul2 = x -> x*2;  
Function<Integer,Integer> add2 = x -> x+2;  
  
System.out.println(  
    mul2.andThen(add2).apply(1));  
System.out.println(  
    add2.andThen(mul2).apply(1));  
System.out.println(  
    add2.compose(mul2).apply(1));
```


Example: Function

- What does the following output?

```
Function<Integer,Integer> mul2 = x -> x*2;
Function<Integer,Integer> add2 = x -> x+2;

System.out.println(
    mul2.andThen(add2).apply(1));
System.out.println(
    add2.andThen(mul2).apply(1));
System.out.println(
    add2.compose(mul2).apply(1));
```

```
4          // (1*2)+2
6          // (1+2)*2
4          // (1*2)+2
```

Example: Predicate

- Test whether $(T \ t)$ is true or false

```
interface Predicate<T> {  
    boolean test(T t);  
  
    // a predicate to test if two arguments are equal  
    static <T> Predicate<T> isEqual(Object targetRef) { ... }  
  
    default Predicate<T> negate() { return (t) -> !test(t); }  
  
    default Predicate<T> or(Predicate<? super T> other) {  
        return (t) -> test(t) || other.test(t);  
    }  
  
    default Predicate<T> and(Predicate<? super T> other) {  
        return (t) -> test(t) && other.test(t);  
    }  
}
```

Example: Predicate

- Fill in [???

```
Predicate<Person> males = p -> p.getGender() == "male";  
Predicate<Person> young = p -> p.getAge() < 18;  
  
Predicate<Person> youngFemales = [???
```

Example: Predicate

- Fill in [???

```
Predicate<Person> males = p -> p.getGender() == "male";  
Predicate<Person> young = p -> p.getAge() < 18;  
  
Predicate<Person> youngFemales = [???
```

```
Predicate<Person> youngFemales = young.and(males.negate());
```

Summary

- Interface with default & static methods
 - Override common default methods
 - Cannot override static methods
- Lambdas
 - Represent anonymous classes with a single abstract method
- Functional interface
 - `Function`
 - `Predicate`
 - ...