



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



SWEN221: Software Development

21: Java 8 (2)

Outline

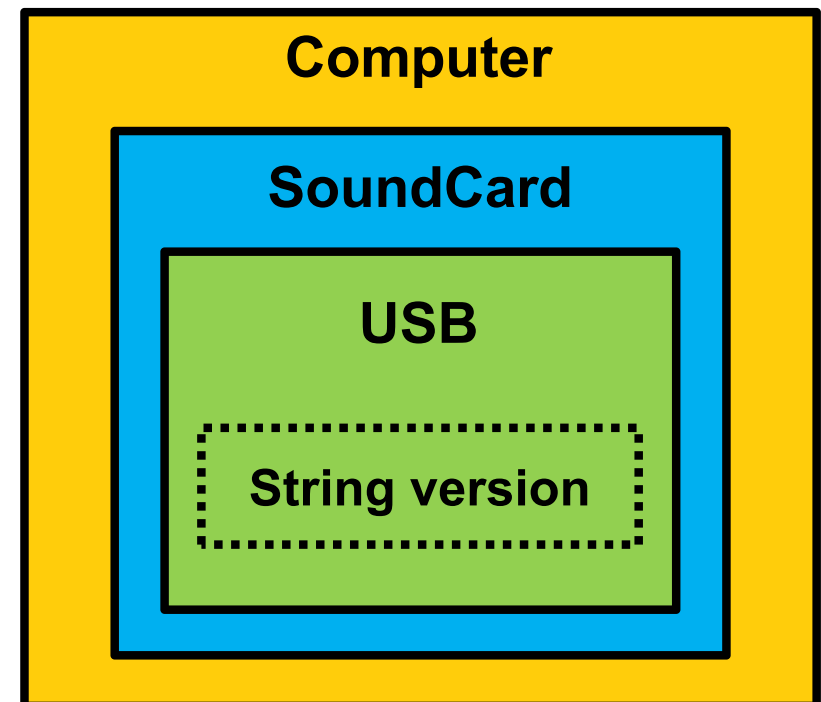
- Optional
- Streams

Why Optional?

- **Get a String version of the USB of the SoundCard of the Computer**

```
String ver = computer.getSoundCard().getUSB().getVersion();
```

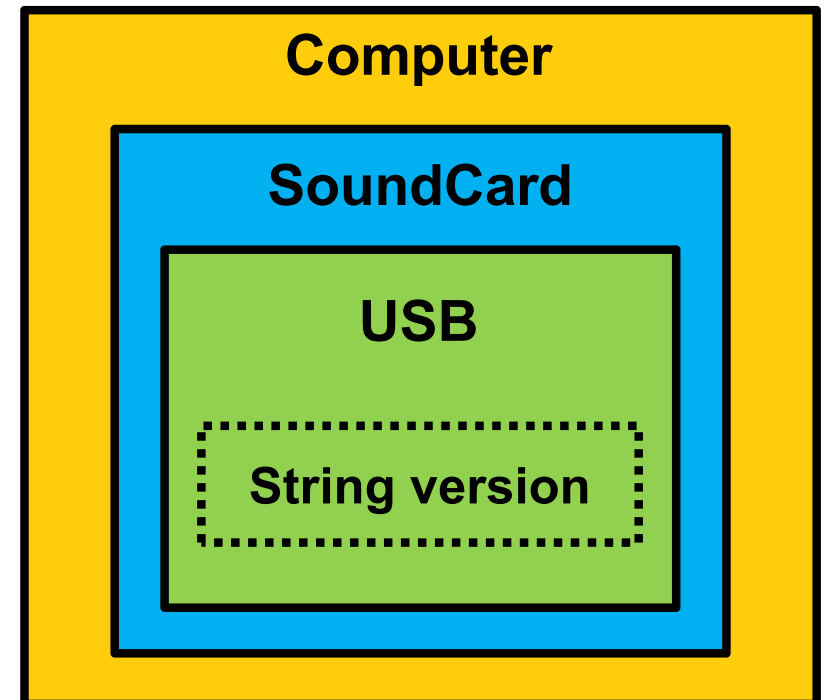
- Many computers don't have sound card
- Sound card may not have USB
- `NullPointerException()`



Why Optional?

```
String ver = "UNKNOWN";  
if(computer != null){  
    Soundcard soundcard = computer.getSoundcard();  
    if(soundcard != null){  
        USB usb = soundcard.getUSB();  
        if(usb != null){ ver = usb.getVersion(); }  
    }  
}
```

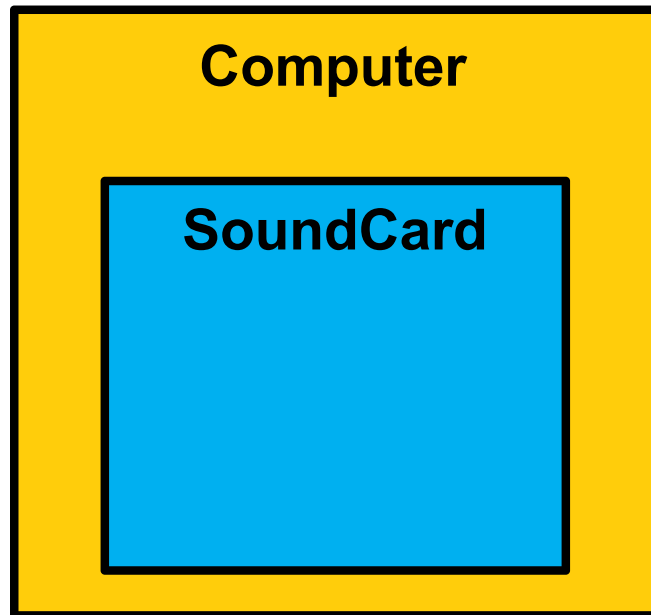
- So many null checks
- Get rid of the ugly code



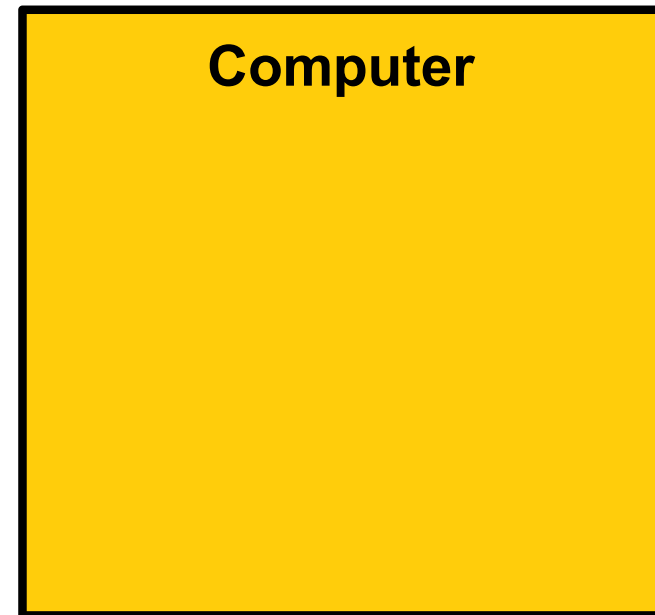
Optional

- A better alternative to null

```
class Computer {  
    private Optional<Soundcard> soundcard;  
    public Optional<Soundcard> getSoundcard() {  
        return soundcard;  
    }  
}
```



or



Creating Optional

- Empty:

```
Optional<Soundcard> sc = Optional.empty();
```

- From non-null value:

```
Soundcard card = new SoundCard();  
Optional<Soundcard> sc = Optional.of(card);
```

- From any value (including null):

```
Soundcard card = ...;  
Optional<Soundcard> sc = Optional.ofNullable(card);
```

Do Something If Present

- Null check

```
SoundCard card = ...;  
if (card != null) { System.out.println(card); }
```

- Optional

```
Optional<SoundCard> optCard = ...;  
if (optCard.isPresent()) {  
    System.out.println(optCard.get());  
}
```

Default Value If Empty

- `T orElse(T default)`
 - Return the default value if the `Optional` is empty

```
Optional<Soundcard> maybeSoundcard = ...;  
Soundcard soundcard =  
    maybeSoundcard.orElse(new Soundcard("default"));
```


What Will be Printed?

```
class Person {  
    public String name;  
    public Person(String name){ this.name = name; }  
}  
  
Optional<Person> op1 = Optional.ofNullable(null);  
Optional<Person> op2 = Optional.of(new Person("Eva"));  
Person p1 = op1.orElse(new Person("Allen"));  
Person p2 = op2.orElse(new Person("Allen"));  
  
if(op1.isPresent()) System.out.println(p1.name);  
if(op2.isPresent()) System.out.println(p2.name);
```

A)

Eva
Allen

B)

Eva
Eva

C)

Eva

D) Cannot compile

What Will be Printed?

```
class Person {  
    public String name;  
    public Person(String name){ this.name = name; }  
}
```

```
Optional<Person> op1 = Optional.ofNullable(null);  
Optional<Person> op2 = Optional.of(new Person("Eva"));  
Person p1 = op1.orElse(new Person("Allen"));  
Person p2 = op2.orElse(new Person("Allen"));  
  
if(op1.isPresent()) System.out.println(p1.name);  
if(op2.isPresent()) System.out.println(p2.name);
```

A)

Eva
Allen



B)

Eva
Eva



C)

Eva



D) Cannot
compile



Java 8 Stream

- Given a collection of transactions
 - ID and Value
- Return the list of the **IDs** of the large transactions (**value >= 80**), **sorted** in the decreasing order of value

ID: 1 Value: 100	ID: 3 Value: 80	ID: 6 Value: 120	ID: 7 Value: 40	ID: 10 Value: 50
---------------------	--------------------	---------------------	--------------------	---------------------

Java 8 Stream

- **Not** InputStream/OutputStream
- Rich library to query and process collections

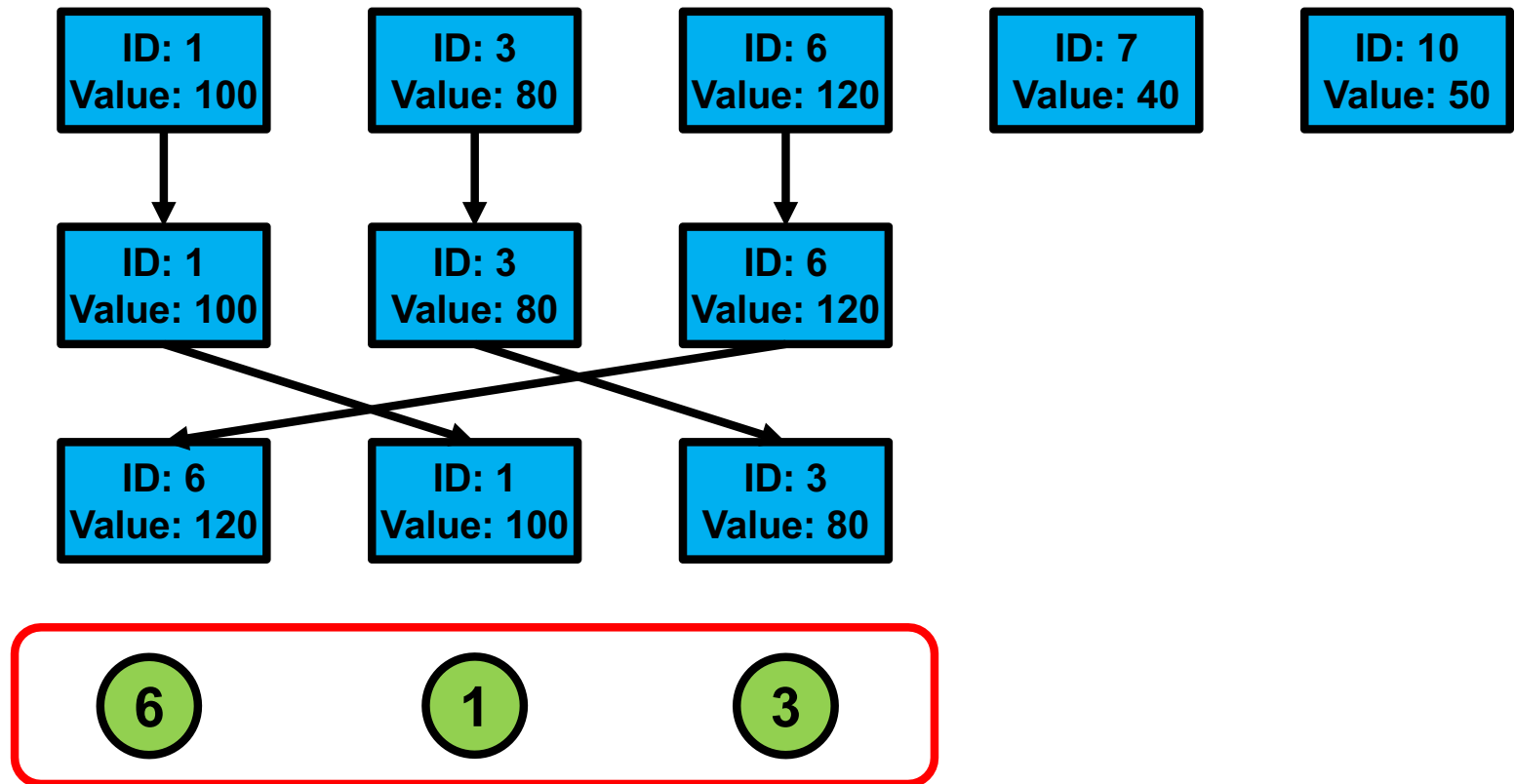
Transactions
Stream

Filter:
value >= 80

Sort:
Value down

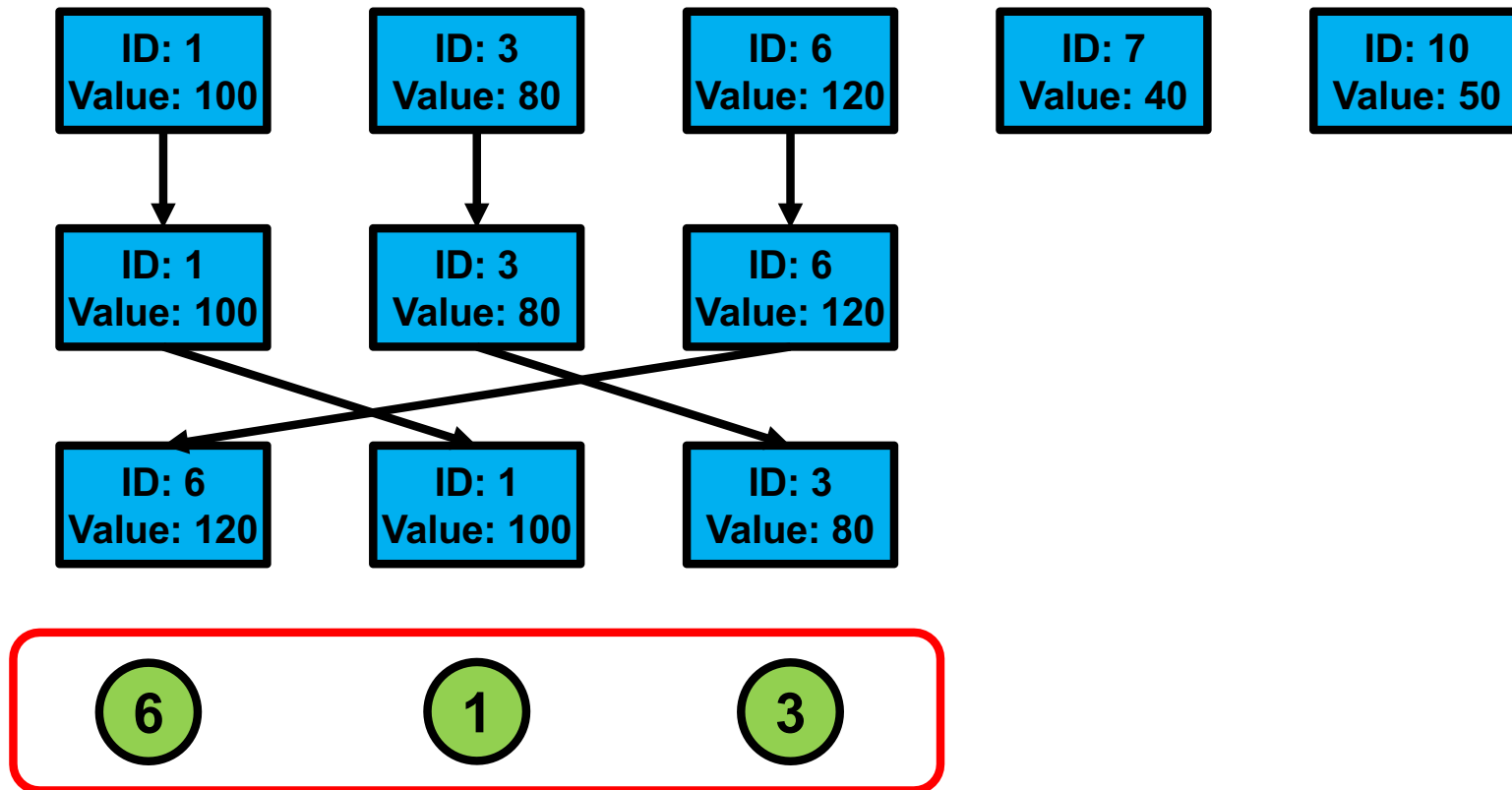
Map:
Get ID

Collect:
To list



Java 8 Stream

```
List<Transaction> transactions = ...  
  
List<Integer> res = transactions.stream()  
    .filter(t -> t.value >= 80)  
    .sorted((t1,t2) -> t2.value-t1.value)  
    .map(t -> t.id).collect(Collectors.toList());
```



Java 8 Stream

- First, turn collection into a stream
 - `list.stream()`
- Then, lots of operations (lambda arguments)
 - `.filter(Predicate)`
 - `.sorted(Comparator)`
 - `.map(Function)`
 - `.collect(Collector)`
 - `.reduce(BinaryOperator)`
 - `.distinct()`
 - ...

What Will Be Printed?

```
List<String> myList =  
    Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");  
myList = myList.stream()  
    .filter(s -> s.startsWith("c"))  
    .map(s -> s.trim())  
    .collect(Collectors.toList());  
for (String s : myList) System.out.println(s);
```

A)

"c1"

B)

" c2 "
"c1"

C)

"c2"
"c1"

What Will Be Printed?

```
List<String> myList =  
    Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");  
myList = myList.stream()  
    .filter(s -> s.startsWith("c"))  
    .map(s -> s.trim())  
    .collect(Collectors.toList());  
for (String s : myList) System.out.println(s);
```

A)

"c1"



B)

" c2 "
"c1"



C)

"c2"
"c1"



What Will Be Printed?

```
List<String> myList =  
    Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");  
myList = myList.stream()  
    .map(s -> s.trim())  
    .filter(s -> s.startsWith("c"))  
    .collect(Collectors.toList());  
for (String s : myList) System.out.println(s);
```

A)

"c1"

B)

" c2 "
"c1"

C)

"c2"
"c1"

What Will Be Printed?

```
List<String> myList =  
    Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");  
myList = myList.stream()  
    .map(s -> s.trim())  
    .filter(s -> s.startsWith("c"))  
    .collect(Collectors.toList());  
  
for (String s : myList) System.out.println(s);
```

Order makes difference!

A)

"c1"



B)

" c2 "
"c1"



C)

"c2"
"c1"



Create Stream From Value(s)

- `Collection.stream()`
- `Stream.of(T... values)`
- `Stream.of(T value)`

```
List<String> myList = Arrays.asList("a", "b", "c");  
Stream<String> s1 = myList.stream();  
Stream<String> s2 = Stream.of("a", "b", "c");  
Stream<String> s3 = Stream.of("a");
```

Stream Reduce

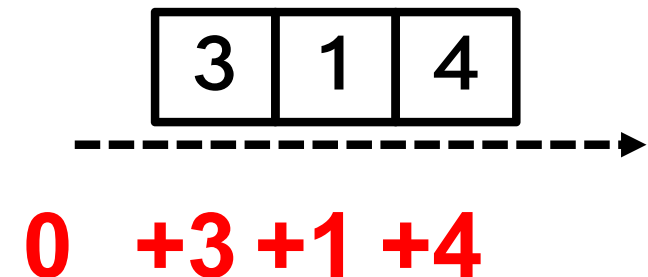
- `T reduce(T identity, BinaryOperator<T> accumulator)`

```
List<Integer> myList = Arrays.asList(3, 1, 4);  
int result = 0;  
for (Integer element : myList)  
    result = result + element;  
return result;
```

```
List<Integer> myList = Arrays.asList(3, 1, 4);  
int result = myList.stream()  
    .reduce(0, (s1,s2) -> s1+s2);
```

Initial value
of result

Accumulator
for sum



Stream in Parallel

- `.parallelStream()`
- Partition into substreams, process in parallel, and combine the results

```
List<String> myList =  
    Arrays.asList("a1 ", " a2 ", "b1", " c2 ", "c1");  
myList = myList.parallelStream()  
    .map(s -> s.trim())  
    .filter(s -> s.startsWith("c"))  
    .collect(Collectors.toList());
```

Reduce in Parallel

- `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`
 - `identity`: **initial value** of the result for each sub-stream
 - `accumulator`: **accumulator** for each sub-stream
 - `combiner`: **combine** the sub-stream results together
-
- Example: calculate sum age of persons

```
int ageSum = persons.parallelStream()
    .reduce(0,
        (sum, p) -> sum + p.age,
        (sum1, sum2) -> sum1 + sum2);
```

Reduce in Parallel

[("Marco", 34); ("Mario", 20); ("Alice", 9)] $0+34+20+9 = 63$

[("Jack", 26); ("John", 10); ("Yoko", 20)] $0+26+10+20 = 56$

[("Steve", 35); ("Teddy", 5)] $0+35+5 = 40$

63
+
56
+
40
=
159

```
int ageSum = persons.parallelStream()
    .reduce(0,
        (sum, p) -> sum + p.age,
        (sum1, sum2) -> sum1 + sum2);
```

Summary

- Optional
 - Explicitly show which fields are optional
- Stream
 - Rich library to process collections
 - A variety of operations with lambda arguments (functional interfaces)
 - Parallelism