

Understand the Java BiFunction Functional Interface

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

**Institute for Software
Integrated Systems**

**Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Part of the Lesson

- Understand foundational functional programming features in Java, e.g.,
 - Lambda expressions
 - Method & constructor references
 - Key functional interfaces
 - Predicate
 - Function
 - BiFunction

Interface BiFunction<T,U,R>

Type Parameters:

T - the type of the first argument to the function

U - the type of the second argument to the function

R - the type of the result of the function

All Known Subinterfaces:

`BinaryOperator<T>`

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

Learning Objectives in this Part of the Lesson

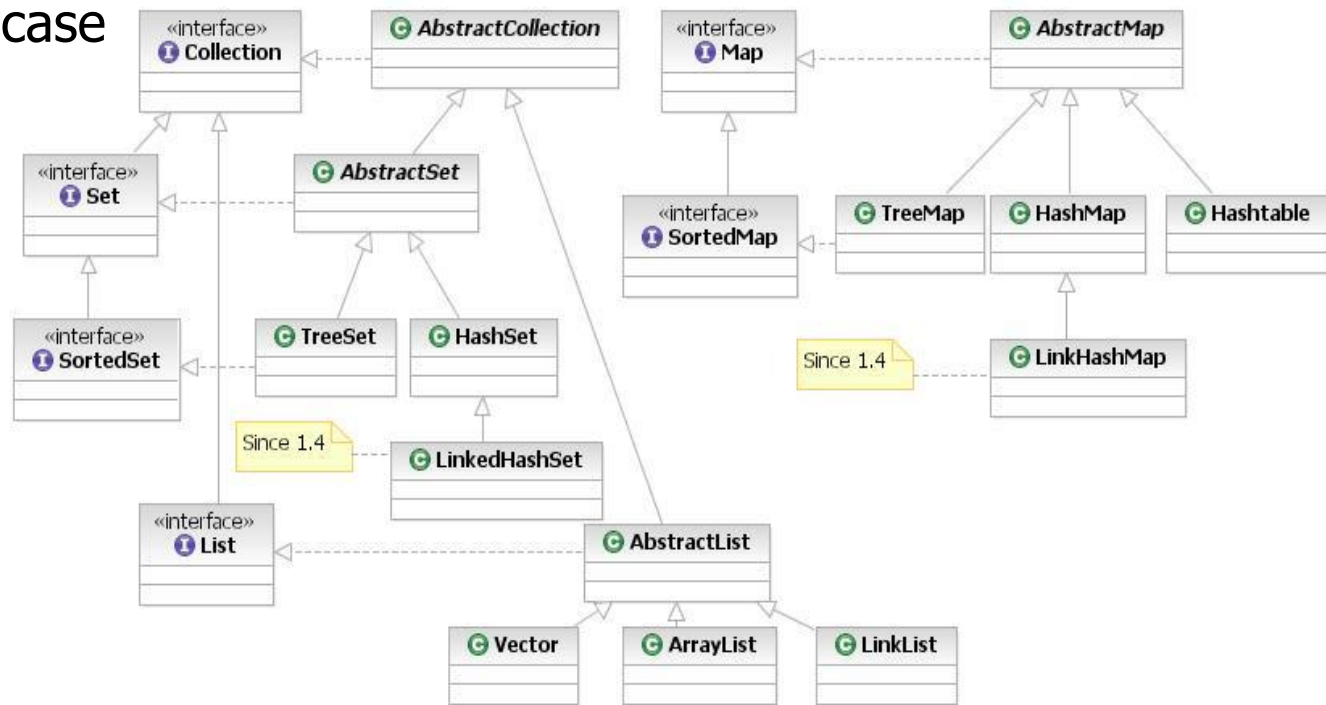
- Understand foundational functional programming features in Java
- Learn how to apply Java bifunctions in concise example programs



See github.com/douglasraigschmidt/LiveLessons/tree/master/Java8

Learning Objectives in this Part of the Lesson

- Understand foundational functional programming features in Java
- Learn how to apply Java bifunctions in concise example programs
- The examples showcase the Java collections framework



See docs.oracle.com/javase/8/docs/technotes/guides/collections

Overview of Functional Interfaces: BiFunction

Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
 - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
 - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`



BiFunction is a generic interface that is parameterized by three reference types

Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
 - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

Its abstract method is passed two parameters of type T & U & returns a value of type R

Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
 - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

```
Map<String, Integer> iqMap =  
    new ConcurrentHashMap<String, Integer>() {  
        { put("Larry", 100); put("Curly", 90); put("Moe", 110); }  
    };  
    
```

Create a map of "stooges" & their IQs!

```
for (Map.Entry<String, Integer> entry : iqMap.entrySet())  
    entry.setValue(entry.getValue() - 50);  
    
```

VS.

```
iqMap.replaceAll((k, v) -> v - 50);
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex4

Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
 - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

```
Map<String, Integer> iqMap =  
    new ConcurrentHashMap<String, Integer>() {  
        { put("Larry", 100); put("Curly", 90); put("Moe", 110); }  
    };  
  
for (Map.Entry<String, Integer> entry : iqMap.entrySet())  
    entry.setValue(entry.getValue() - 50);
```

vs.

Conventional way of subtracting 50 IQ points from each stooge in map

```
iqMap.replaceAll((k, v) -> v - 50);
```

Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
 - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

```
Map<String, Integer> iqMap =  
    new ConcurrentHashMap<String, Integer>() {  
        { put("Larry", 100); put("Curly", 90); put("Moe", 110); }  
    };  
  
for (Map.Entry<String, Integer> entry : iqMap.entrySet())  
    entry.setValue(entry.getValue() - 50);
```

vs.

BiFunction lambda subtracts 50 IQ points from each stooge in map

```
iqMap.replaceAll((k, v) -> v - 50);
```

Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,
 - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

```
Map<String, Integer> iqMap =  
    new ConcurrentHashMap<String, Integer>() {  
        { put("Larry", 100); put("Curly", 90); put("Moe", 110); }  
    };  
  
for (Map.Entry<String, Integer> entry : iqMap.entrySet())  
    entry.setValue(entry.getValue() - 50);
```

vs.

Unlike Entry operations, replaceAll() operates in a thread-safe manner!

```
iqMap.replaceAll((k, v) -> v - 50);
```

Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

- ```
public interface BiFunction<T, U, R> { R apply(T t, U u); }
```

```
class ConcurrentHashMap<K,V> {
```

```
 ...
```

```
 public void replaceAll
```

```
 (BiFunction<? super K, ? super V, ? extends V> function) {
```

```
 ...
```

```
 for (Node<K,V> p; (p = it.advance()) != null;) {
```

```
 V oldValue = p.val;
```

```
 for (K key = p.key;;) {
```

```
 V newValue = function.apply(key, oldValue);
```

```
 ...
```

```
 replaceNode(key, newValue, oldValue)
```

```
 ...
```

The `replaceAll()` method uses the bifunction passed to it in a thread-safe way

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

- `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

```
class ConcurrentHashMap<K,V> {
```

```
...
```

```
public void replaceAll
```

```
 (BiFunction<? super K, ? super V, ? extends V> function) {
```

```
...
```

```
for (Node<K,V> p; (p = it.advance()) != null;) {
```

```
 V oldValue = p.val;
```

```
 for (K key = p.key;;) {
```

```
 V newValue = function.apply(key, oldValue);
```

```
 ...
```

```
 replaceNode(key, newValue, oldValue)
```

```
 ...
```

(k, v) -> v - 50

The bifunction parameter is bound to the lambda expression `v - 50`

# Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 parameters & returns a result, e.g.,

- ```
public interface BiFunction<T, U, R> { R apply(T t, U u); }
```

```
class ConcurrentHashMap<K,V> {  
    ...  
    public void replaceAll  
        (BiFunction<? super K, ? super V, ? extends V> function) {  
        ...  
        for (Node<K,V> p; (p = it.advance()) != null; ) {  
            V oldValue = p.val;  
            for (K key = p.key;;) {  
                V newValue = function.apply(key, oldValue);  
                ...  
                replaceNode(key, newValue, oldValue)  
                ...  
            }  
        }  
    }  
}
```

V newValue =
oldValue - 50

The apply() method is replaced by the v – 50 bifunction lambda

End of Understand the Java BiFunction Functional Interface