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 the BiFunction functional interface in Java & recognize how it can be used in conjunction with lambda expressions & method references

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.

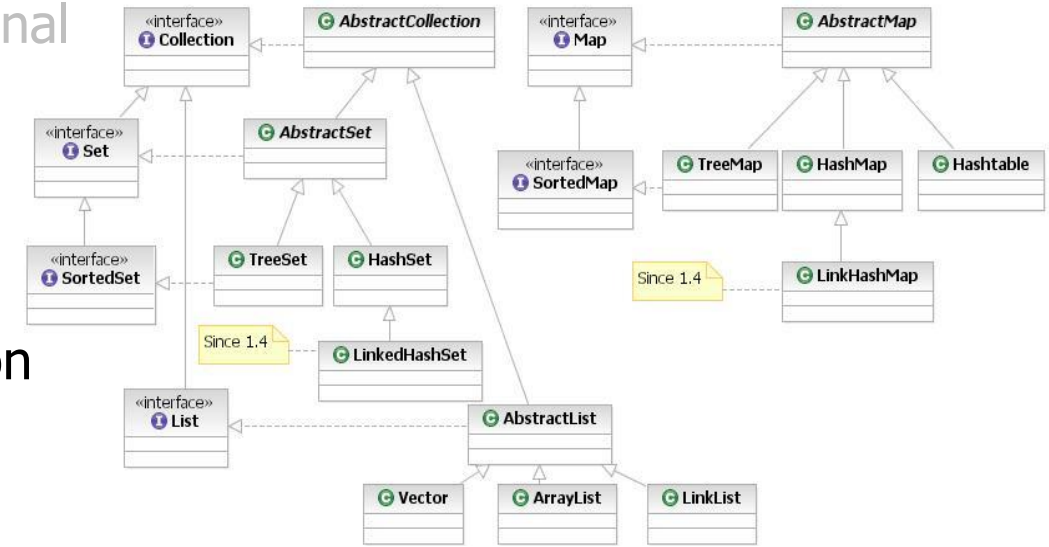
Learning Objectives in this Part of the Lesson

- Understand the BiFunction functional interface in Java & recognize how it can be used in conjunction with lambda expressions & method references
- Know how to apply Java BiFunction in a concise example

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

Learning Objectives in this Part of the Lesson

- Understand the BiFunction functional interface in Java & recognize how it can be used in conjunction with lambda expressions & method references
- Know how to apply Java BiFunction in a concise example
  - This example showcases the Java collections framework's ConcurrentHashMap class & Set interface



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

Overview of Functional Interfaces: BiFunction

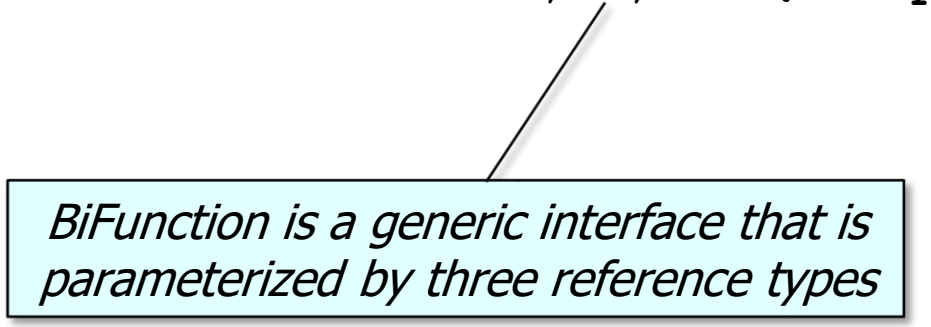
Overview of Common Functional Interfaces: BiFunction

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

See docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html

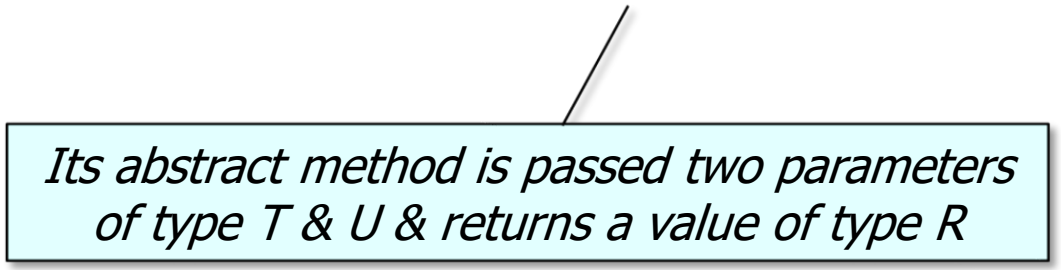
Overview of Common Functional Interfaces: BiFunction

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

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


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

vs.

```
stooges.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 params & returns a result
  - `public interface BiFunction<T, U, R> { R apply(T t, U u); }`

```
Map<String, Integer> stooges =
  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 : stooges.entrySet())
   entry.setValue(entry.getValue() - 50);
```

vs.

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

See en.wikipedia.org/wiki/The_Three_Stooges

Overview of Common Functional Interfaces: BiFunction

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

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

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

vs.

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

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

Overview of Common Functional Interfaces: BiFunction

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

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


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

vs.

BiFunction lambda subtracts 50 IQ points from each stooge in map

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

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#replaceAll

Overview of Common Functional Interfaces: BiFunction

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

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

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

vs.

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

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

See winterbe.com/posts/2015/05/22/java8-concurrency-tutorial-atomic-concurrent-map-examples

Overview of Common Functional Interfaces: BiFunction

- A *BiFunction* applies a computation on 2 params & returns a result
  - `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 params & returns a result
  - `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 params & returns a result
  - `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 the Java BiFunction Functional Interface