Understand the Java Function Functional Interface

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

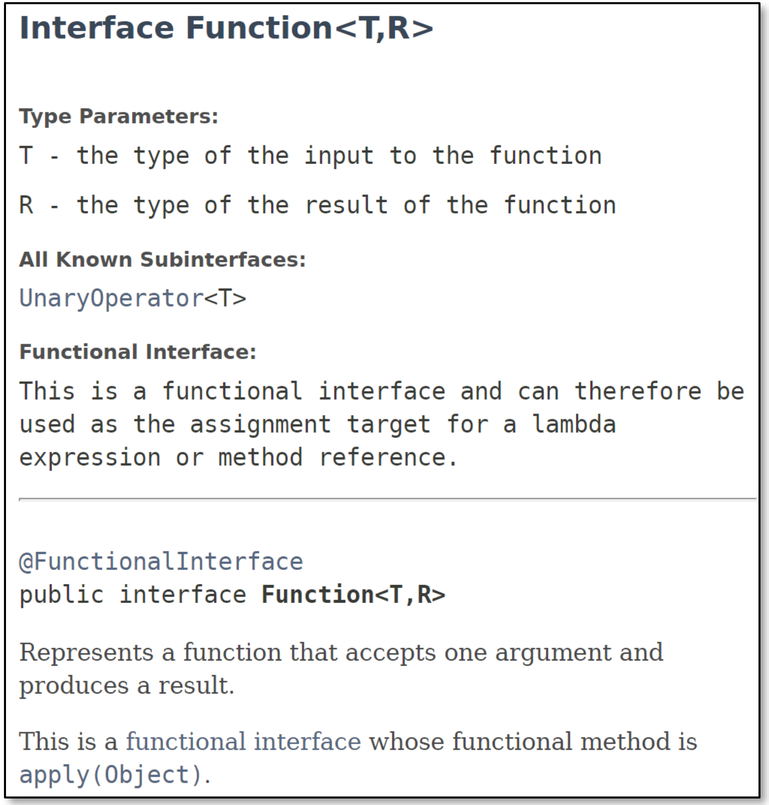
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Lesson

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



Interface Function<T,R>

Type Parameters:

T - the type of the input to the function

R - the type of the result of the function

All Known Subinterfaces:

UnaryOperator<T>

Functional Interface:

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

```
@FunctionalInterface
public interface Function<T,R>
```

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is apply(Object).

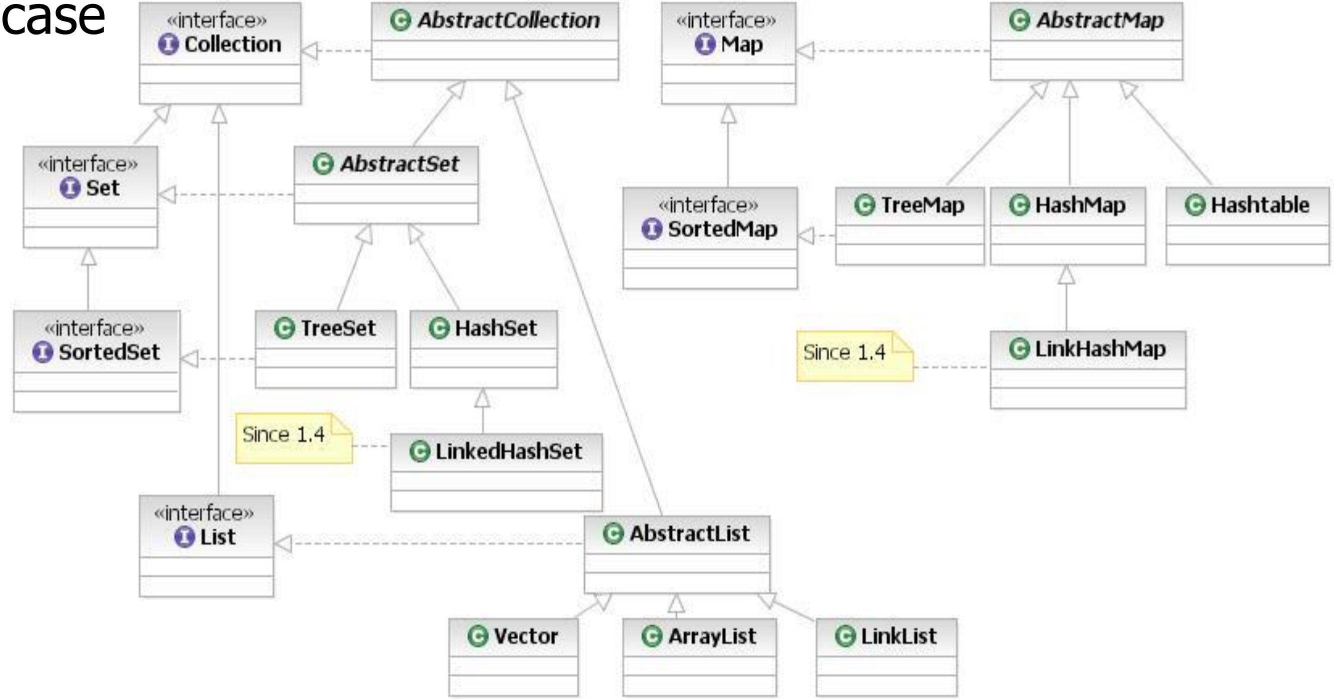
Learning Objectives in this Part of the Lesson

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

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

Learning Objectives in this Part of the Lesson

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



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

Overview of the Function Functional Interface

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

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

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

Function is a generic interface that is parameterized by two reference types.

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

Its abstract method is passed a parameter of type T & returns a value of type R.

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
Map<Integer, Integer> primeCache =
  new ConcurrentHashMap<>();

...
Long smallestFactor = primeCache.computeIfAbsent
    (primeCandidate, (key) -> primeChecker(key));
...

Integer primeChecker(Integer primeCandidate) {
  ... // Returns 0 if a number is prime or the smallest
      // factor if it's not prime
}
```

This map caches the results of prime # computations

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex9

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - **`public interface Function<T, R> { R apply(T t); }`**

```
Map<Integer, Integer> primeCache =
  new ConcurrentHashMap<>();

...
Long smallestFactor = primeCache.computeIfAbsent
    (primeCandidate, (key) -> primeChecker(key));
...

Integer primeChecker(Integer primeCandidate) {
  ... // Returns 0 if a number is prime or the smallest
      // factor if it's not prime
}
```

If key isn't already associated with a value, atomically compute the value using the given mapping function & enter it into the map

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

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
Map<Integer, Integer> primeCache =
  new ConcurrentHashMap<>();

...
Long smallestFactor = primeCache.computeIfAbsent
    (primeCandidate, (key) -> primeChecker(key));
...

Integer primeChecker(Integer primeCandidate) {
  ... // Returns 0 if a number is prime or the smallest
      // factor if it's not prime
}
```

This method provides atomic "check then act" semantics

See dig.cs.illinois.edu/papers/checkThenAct.pdf

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
Map<Integer, Integer> primeCache =
  new ConcurrentHashMap<>();

...
Long smallestFactor = primeCache.computeIfAbsent
    (primeCandidate, (key) -> primeChecker(key));
...

Integer primeChecker(Integer primeCandidate) {
  ... // Returns 0 if a number is prime or the smallest
      // factor if it's not prime
}
```

A lambda expression that calls a function

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
Map<Integer, Integer> primeCache =
  new ConcurrentHashMap<>();

...
Long smallestFactor = primeCache.computeIfAbsent
     (primeCandidate, this::primeChecker);
...

Integer primeChecker(Integer primeCandidate) {
  ... // Returns 0 if a number is prime or the smallest
      // factor if it’s not prime
}
```

Could also be a passed as a method reference

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
class ConcurrentHashMap<K,V> ...
  public V computeIfAbsent(K key,
           Function<? super K, ? extends V> mappingFunction) {



    ...
    if ((f = tabAt(tab, i = (n - 1) & h)) == null)
      ...
      if ((val = mappingFunction.apply(key)) != null)
        node = new Node<K,V>(h, key, val, null);
      ...
```

Here’s how computeIfAbsent() uses the function passed to it (atomically)

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
class ConcurrentHashMap<K,V> ...
  public V computeIfAbsent(K key,
           Function<? super K, ? extends V> mappingFunction) {
    ...
    if ((f = tabAt(tab, i = (n - 1) & h)) == null)
      ...
      if ((val = mappingFunction.apply(key)) != null)
        node = new Node<K,V>(h, key, val, null);
      ...
```

'super' is a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type

See docs.oracle.com/javase/tutorial/java/generics/lowerBounded.html

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
class ConcurrentHashMap<K,V> ...
  public V computeIfAbsent(K key,
           Function<? super K, ? extends V> mappingFunction) {
```

'extends' is an upper bounded wildcard that restricts the unknown type to be a specific type or a subtype of that type

```
    ...
    if ((f = tabAt(tab, i = (n - 1) & h)) == null)
      ...
      if ((val = mappingFunction.apply(key)) != null)
        node = new Node<K,V>(h, key, val, null);
      ...
```

See docs.oracle.com/javase/tutorial/java/generics/upperBounded.html

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
class ConcurrentHashMap<K,V> ...
  public V computeIfAbsent(K key,
           Function<? super K, ? extends V> mappingFunction) {
    ...
    if ((f = tabAt(tab, i = (n - 1) & h)) == null)
      ...
      if ((val = mappingFunction.apply(key)) != null)
        node = new Node<K,V>(h, key, val, null);
      ...
```

'super' & ‘extends' play different roles in Java generics

See en.wikipedia.org/wiki/Generics_in_Java#Type_wildcards

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
class ConcurrentHashMap<K,V> ...
  public V computeIfAbsent(K key,
           Function<? super K, ? extends V> mappingFunction) {
```

`this::primeChecker`

```
    ...
    if ((f = tabAt(tab, i = (n - 1) & h)) == null)
      ...
      if ((val = mappingFunction.apply(key)) != null)
        node = new Node<K,V>(h, key, val, null);
      ...
```

The function parameter is bound to this::primeChecker method reference

Overview of the Function Functional Interface

- A *Function* applies a computation on 1 parameter & returns a result, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
class ConcurrentHashMap<K,V> ...
  public V computeIfAbsent(K key,
           Function<? super K, ? extends V> mappingFunction) {
```

```
if ((val = primeChecker(key)) != null)
```

```
    ...
    if ((f = tabAt(tab, i = (n - 1) & h)) == null)
      ...
      if ((val = mappingFunction.apply(key)) != null)
        node = new Node<K,V>(h, key, val, null);
      ...
```

The apply() method is replaced with the primeChecker() lambda function

Another Function Interface Example

Another Function Interface Example

- Here's another example of applying a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
List<Thread> threads = Arrays.asList(new Thread("Larry"),
                                     new Thread("Curly"),
                                     new Thread("Moe"));
```

Create a list of threads named after the three stooges

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex5

Another Function Interface Example

- Here's another example of applying a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
List<Thread> threads = Arrays.asList(new Thread("Larry"),
                                     new Thread("Curly"),
                                     new Thread("Moe"));
```

A method reference to a Function used to sort threads by name

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

See dzone.com/articles/java-8-comparator-how-to-sort-a-list

Another Function Interface Example

- Here's another example of applying a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
List<Thread> threads = Arrays.asList(new Thread("Larry"),
                                     new Thread("Curly"),
                                     new Thread("Moe"));
```

This method uses the Thread::getName method reference to impose a total ordering on some collection of objects

```
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

Another Function Interface Example

- Here's another example of applying a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
interface Comparator {
  ...
  static <T, U extends Comparable<? super U>> Comparator<T>
           comparing(Function<? super T, ? extends U> keyEx) {
    return ((c1, c2) ->
            keyEx.apply(c1)
                 .compareTo(keyEx.apply(c2)); }
```

Imposes a total ordering on a collection of objects

See docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparing

Another Function Interface Example

- Here's another example of applying a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
interface Comparator {
  ...
  static <T, U extends Comparable<? super U>> Comparator<T>
           comparing(Function<? super T, ? extends U> keyEx) {
    return ((c1, c2) ->
            keyEx.apply(c1)
                 .compareTo(keyEx.apply(c2)); }
```

The comparing() method is passed a Function parameter called keyEx

Another Function Interface Example

- Here's another example of applying a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
interface Comparator {
  ...
  static <T, U extends Comparable<? super U>> Comparator<T>
           comparing(Function<? super T, ? extends U> keyEx) {
    return ((c1, c2) ->
            keyEx.apply(c1)
                 .compareTo(keyEx.apply(c2)); }
```

Thread::getName

The Thread::getName method reference is bound to the keyEx parameter

Another Function Interface Example

- Here's another example of applying a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
interface Comparator {
  ...
  static <T, U extends Comparable<? super U>> Comparator<T>
           comparing(Function<? super T, ? extends U> keyEx) {
    return ((c1, c2) ->
           keyEx.apply(c1)
                .compareTo(keyEx.apply(c2)); }
```

c1 & c2 are thread objects being compared by sort()

Another Function Interface Example

- Here's another example of applying a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
interface Comparator {
  ...
  static <T, U extends Comparable<? super U>> Comparator<T>
           comparing(Function<? super T, ? extends U> keyEx) {
    return ((c1, c2) ->
            keyEx.apply(c1)
                 .compareTo(keyEx.apply(c2)); }
```

The apply() method of the keyEx function is used to compare strings

Another Function Interface Example

- Here's another example of applying a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
interface Comparator {
  ...
  static <T, U extends Comparable<? super U>> Comparator<T>
           comparing(Function<? super T, ? extends U> keyEx) {
    return ((c1, c2) ->
             keyEx.apply(c1)
                  .compareTo(keyEx.apply(c2)); }
```

```
c1.getName().compareTo(c2.getName())
```

The thread::getName method reference is called to compare two thread names

Composing Functions

Composing Functions

- It’s also possible to compose functions.
  - `public interface Function<T, R> { R apply(T t); }`

```
class HtmlTagMaker {
  static String addLessThan(String t) { return "<" + t; }
  static String addGreaterThan(String t) { return t + ">"; }
}


Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
  .andThen(HtmlTagMaker::addGreaterThan);


System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
             + tagger.apply("/BODY") + tagger.apply("/HTML"));
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex3

Composing Functions

- It's also possible to compose functions.
  - `public interface Function<T, R> { R apply(T t); }`

```
class HtmlTagMaker {
  static String addLessThan(String t) { return "<" + t; }
  static String addGreaterThan(String t) { return t + ">"; }
}
```

These methods prepend '<' & append '>' to a string, respectively

```
Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
  .andThen(HtmlTagMaker::addGreaterThan);


System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
                + tagger.apply("/BODY") + tagger.apply("/HTML"));
```

Composing Functions

- It's also possible to compose functions.
  - `public interface Function<T, R> { R apply(T t); }`

```
class HtmlTagMaker {
  static String addLessThan(String t) { return "<" + t; }
  static String addGreaterThan(String t) { return t + ">"; }
}
```

These functions prepend '<' & append '>' to a string

```
Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
  .andThen(HtmlTagMaker::addGreaterThan);


System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
                + tagger.apply("/BODY") + tagger.apply("/HTML"))
```

Composing Functions

- It's also possible to compose functions.
  - `public interface Function<T, R> { R apply(T t); }`

```
class HtmlTagMaker {
  static String addLessThan(String t) { return "<" + t; }
  static String addGreaterThan(String t) { return t + ">"; }
}



Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
  .andThen(HtmlTagMaker::addGreaterThan);
```

This method composes two functions!

```
System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
                 + tagger.apply("/BODY") + tagger.apply("/HTML"));
```

See docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#andThen

Composing Functions

- It’s also possible to compose functions.
  - `public interface Function<T, R> { R apply(T t); }`

```
class HtmlTagMaker {
  static String addLessThan(String t) { return "<" + t; }
  static String addGreaterThan(String t) { return t + ">"; }
}



Function<String, String> lessThan = HtmlTagMaker::addLessThan;
Function<String, String> tagger = lessThan
  .andThen(HtmlTagMaker::addGreaterThan);
```

Prints "<HTML><BODY></BODY></HTML>”

```
System.out.println(tagger.apply("HTML") + tagger.apply("BODY")
                 + tagger.apply("/BODY") + tagger.apply("/HTML"));
```

See docs.oracle.com/javase/8/docs/api/java/util/function/Function.html#andThen

End of Understand the Java Function Functional Interface