The Java Function Functional Interface

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

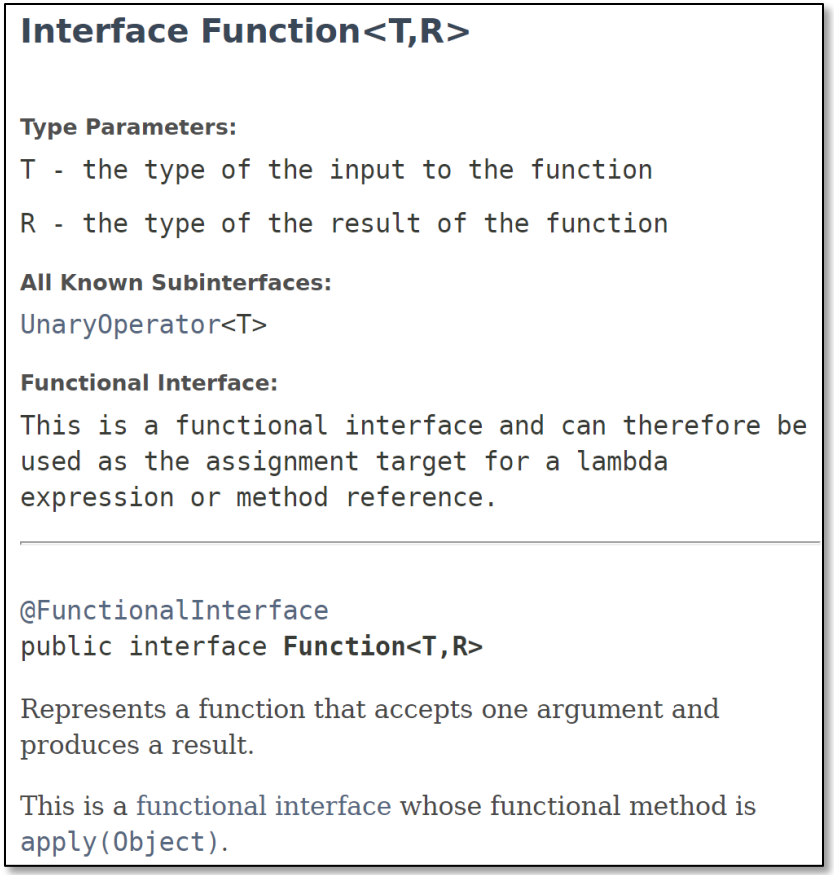
Institute for Software Integrated Systems

Vanderbilt University

Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

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



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).

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

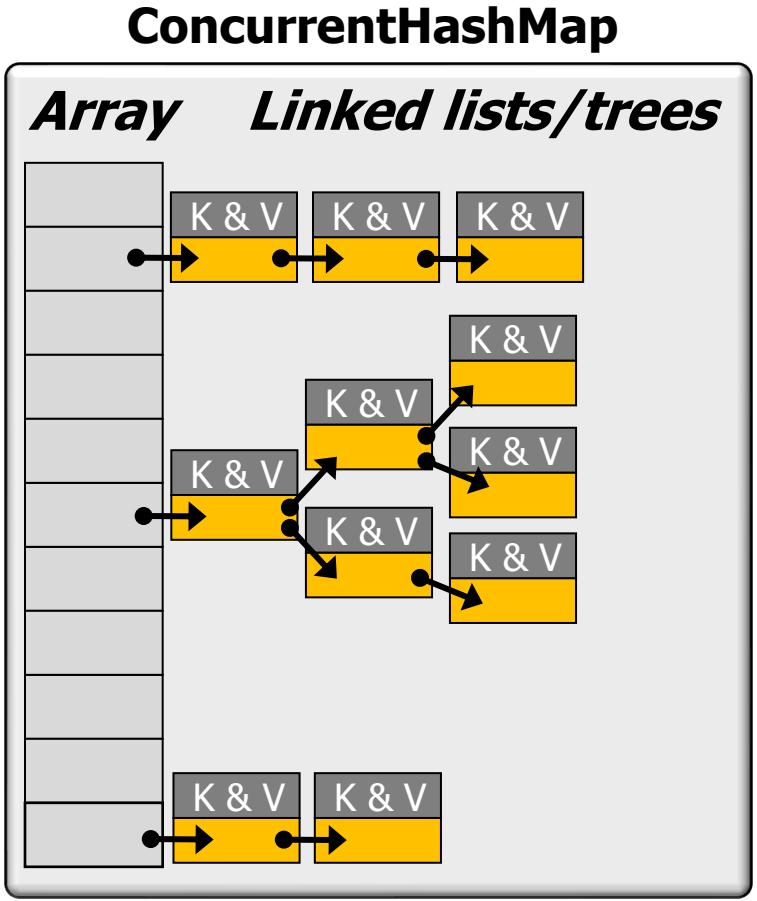
Learning Objectives in this Part of the Lesson

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

See github.com/douglascraigschmidt/ModernJava/tree/main/FP/ex9

Learning Objectives in this Part of the Lesson

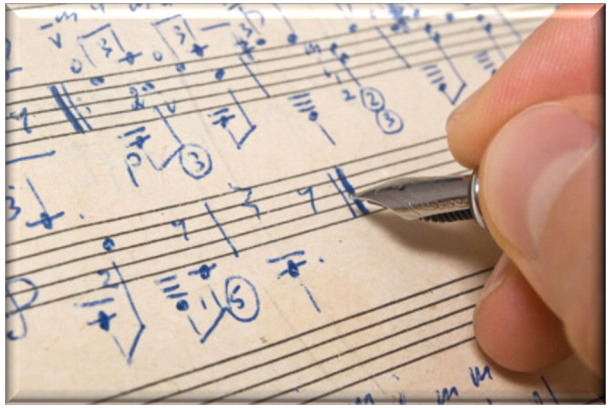
- Understand the Function functional interface in Java & recognize how it can be used in conjunction with lambda expressions & method references
- Know how to apply Java Function in a concise example
  - This example showcases the Java collections framework's ConcurrentHashMap class



See www.nagarro.com/en/blog/post/24/performance-improvement-for-hashmap-in-java-8

Learning Objectives in this Part of the Lesson

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



See tutorials.jenkov.com/java-functional-programming/functional-composition.html

Overview of the Function Functional Interface

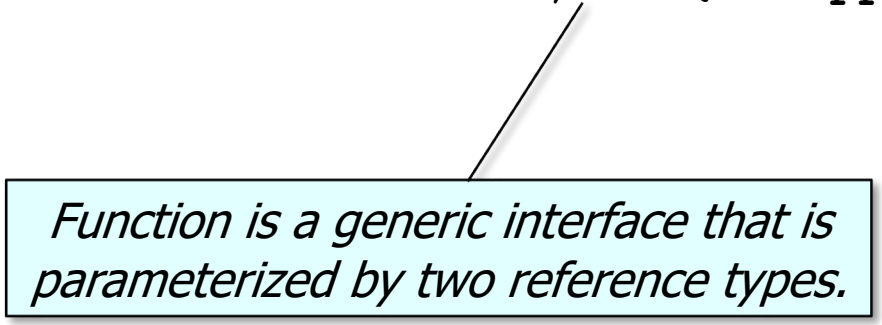
Overview of the Function Functional Interface

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

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

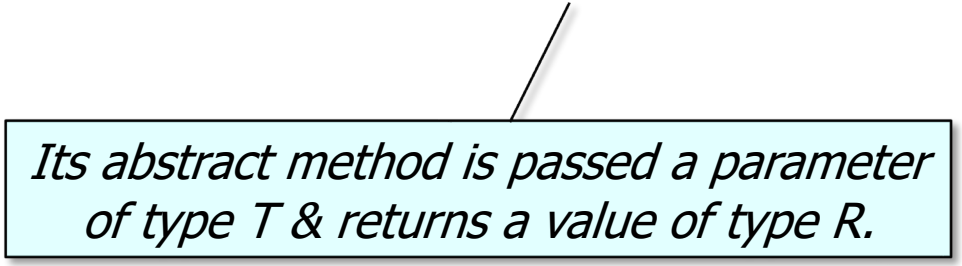
Overview of the Function Functional Interface

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



Overview of the Function Functional Interface

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



Applying the Function Functional Interface

Applying the Function Functional Interface

- This example uses the Java Function functional interface in conjunction with ConcurrentHashMap to compute, cache, & retrieve large prime numbers

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



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

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

See github.com/douglascraigschmidt/ModernJava/tree/main/FP/ex9

Applying the Function Functional Interface

- This example uses the Java Function functional interface in conjunction with ConcurrentHashMap to compute, cache, & retrieve large prime numbers

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

This map caches the results of prime # computations

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

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

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

Applying the Function Functional Interface

- This example uses the Java Function functional interface in conjunction with ConcurrentHashMap to compute, cache, & retrieve large prime numbers

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

This technique is known as "memoization"

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

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

See en.wikipedia.org/wiki/Memoization

Applying the Function Functional Interface

- This example uses the Java Function functional interface in conjunction with ConcurrentHashMap to compute, cache, & retrieve large prime numbers

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

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

Integer isPrime(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

Applying the Function Functional Interface

- This example uses the Java Function functional interface in conjunction with ConcurrentHashMap to compute, cache, & retrieve large prime numbers

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


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

Integer isPrime(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

Applying the Function Functional Interface

- This example uses the Java Function functional interface in conjunction with ConcurrentHashMap to compute, cache, & retrieve large prime numbers

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

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

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

The Function can be provided via a lambda expression

Applying the Function Functional Interface

- This example uses the Java Function functional interface in conjunction with ConcurrentHashMap to compute, cache, & retrieve large prime numbers

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

...
Long smallestFactor = primeCache.computeIfAbsent
    (primeCandidate, PrimeUtils::isPrime);
...

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

The Function can be provided via a method reference

How ConcurrentHashMap Uses the Function Interface

How ConcurrentHashMap Uses the Function Interface

- Here's how the ConcurrentHashMap computeIfAbsent() method uses the *Function* passed to it (atomically)

```
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);
      ...
```

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

How ConcurrentHashMap Uses the Function Interface

- Here's how the ConcurrentHashMap computeIfAbsent() method uses the *Function* passed to it (atomically)

```
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

How ConcurrentHashMap Uses the Function Interface

- Here's how the ConcurrentHashMap computeIfAbsent() method uses the *Function* passed to it (atomically)

```
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);
      ...
```

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

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

How ConcurrentHashMap Uses the Function Interface

- Here's how the ConcurrentHashMap computeIfAbsent() method uses the *Function* passed to it (atomically)

```
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

How ConcurrentHashMap Uses the Function Interface

- Here's how the ConcurrentHashMap computeIfAbsent() method uses the *Function* passed to it (atomically)

```
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);
      ...
```

PrimeUtils::isPrime

The function parameter is bound to PrimeUtils::isPrime method reference

How ConcurrentHashMap Uses the Function Interface

- Here's how the ConcurrentHashMap computeIfAbsent() method uses the *Function* passed to it (atomically)

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

                        if ((val = isPrime(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);
      ...
```

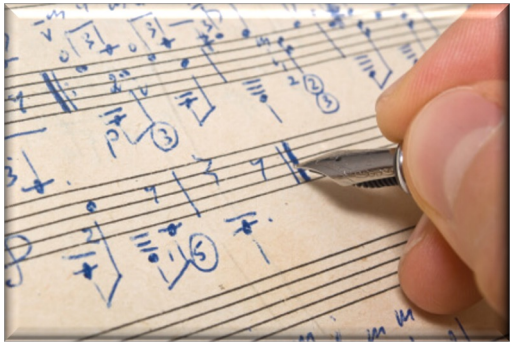
The apply() method is replaced with the PrimeUtils.isPrime() lambda function

Composing Java Function Objects

Composing Java Function Objects

- It's also possible to compose Function objects.
  - `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/ModernJava/tree/main/FP/ex10

Composing Java Function Objects

- It's also possible to compose Function objects.
  - `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"));
```

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

Composing Java Function Objects

- It's also possible to compose Function objects.
  - `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 Java Function Objects

- It's also possible to compose Function objects.
  - `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"));
```

This method composes two Function objects!

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

Composing Java Function Objects

- It's also possible to compose Function objects.
  - `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"));
```

Prints "<HTML><BODY></BODY></HTML>"

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

End of the Java Function Functional Interface