Overview of Java 8 Functional Interfaces

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

- Recognize foundational functional programming features in Java 8, e.g.,
  - Lambda expressions
  - Method & constructor references
  - Functional interfaces

Learning Objectives in this Lesson

- Recognize foundational functional programming features in Java 8, e.g.,
  - Lambda expressions
  - Method & constructor references
  - Functional interfaces

These features are the foundation for Java 8's concurrency/parallelism frameworks

Learning Objectives in this Lesson

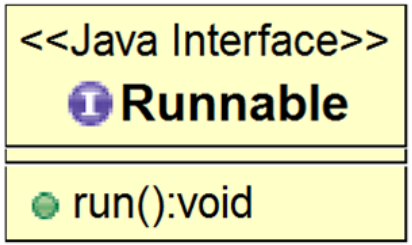
- Recognize foundational functional programming features in Java 8
- Understand how these Java 8 features are applied in concise example programs

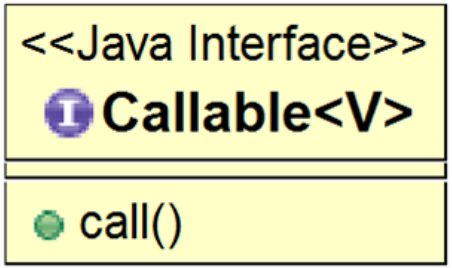
See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8

Overview of Common Functional Interfaces

Overview of Common Functional Interfaces

- A *functional interface* contains only one abstract method





See www.oreilly.com/learning/java-8-functional-interfaces

Overview of Common Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument

```
<T> void runTest(Function<T, T> fact, T n) {
  System.out.println(n + " factorial = " + fact.apply(n));
}

runTest(ParallelStreamFactorial::factorial, n);

...
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex16

Overview of Common Functional Interfaces

- A functional interface is the type used for a parameter when a lambda expression or method reference is passed as an argument

```
<T> void runTest(Function<T, T> fact, T n) {
  System.out.println(n + " factorial = " + fact.apply(n));
}

runTest(ParallelStreamFactorial::factorial, n);

...
```

```
static BigInteger factorial(BigInteger n) { return LongStream
    .rangeClosed(1, n)
    .parallel()
    .mapToObj(BigInteger::valueOf)
    .reduce(BigInteger.ONE, BigInteger::multiply);
}
```

Overview of Common Functional Interfaces

- Java 8 defines many types of functional interfaces

Package java.util.function

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

Interface	Description
BiConsumer<T,U>	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T,U,R>	Represents a function that accepts two arguments and produces a result.
BinaryOperator<T>	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.
BiPredicate<T,U>	Represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	Represents a supplier of `boolean`-valued results.
Consumer<T>	Represents an operation that accepts a single input argument and returns no result.
DoubleBinaryOperator	Represents an operation upon two `double`-valued operands and producing a `double`-valued result.
DoubleConsumer	Represents an operation that accepts a single `double`-valued argument and returns no result.
DoubleFunction<R>	Represents a function that accepts a double-valued argument and produces a result.
DoublePredicate	Represents a predicate (boolean-valued function) of one `double`-valued argument.
DoubleSupplier	Represents a supplier of `double`-valued results.
DoubleToIntFunction	Represents a function that accepts a double-valued argument and produces an int-valued result.
DoubleToLongFunction	Represents a function that accepts a double-valued argument and produces a long-valued result.
DoubleUnaryOperator	Represents an operation on a single `double`-valued operand that produces a `double`-valued result.
Function<T,R>	Represents a function that accepts one argument and produces a result.

See docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Overview of Common Functional Interfaces

- Java 8 defines many types of functional interfaces
  - This list is large due to the need to support reference types & primitive types..

Package java.util.function

Functional interfaces provide target types for lambda expressions and method references.

See: Description

Interface Summary

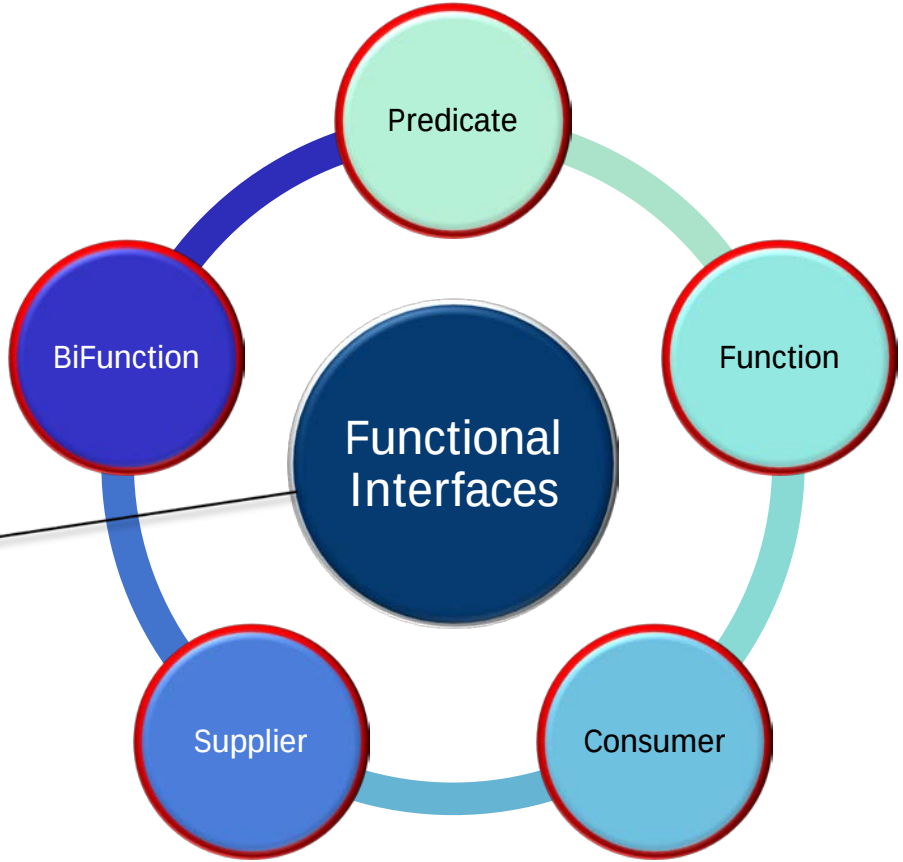
Interface	Description
IntConsumer	Represents an operation that accepts a single int-valued argument and returns no result.
IntFunction<R>	Represents a function that accepts an int-valued argument and produces a result.
IntPredicate	Represents a predicate (boolean-valued function) of one int-valued argument.
IntSupplier	Represents a supplier of int-valued results.
IntToDoubleFunction	Represents a function that accepts an int-valued argument and produces a double-valued result.
IntToLongFunction	Represents a function that accepts an int-valued argument and produces a long-valued result.
IntUnaryOperator	Represents an operation on a single int-valued operand that produces an int-valued result.
LongBinaryOperator	Represents an operation upon two long-valued operands and producing a long-valued result.
LongConsumer	Represents an operation that accepts a single long-valued argument and returns no result.
LongFunction<R>	Represents a function that accepts a long-valued argument and produces a result.
LongPredicate	Represents a predicate (boolean-valued function) of one long-valued argument.
LongSupplier	Represents a supplier of long-valued results.
LongToDoubleFunction	Represents a function that accepts a long-valued argument and produces a double-valued result.
LongToIntFunction	Represents a function that accepts a long-valued argument and produces an int-valued result.
LongUnaryOperator	Represents an operation on a single long-valued operand that produces a long-valued result.
ObjDoubleConsumer<T>	Represents an operation that accepts an object-valued and a double-valued argument, and returns no result.
ObjIntConsumer<T>	Represents an operation that accepts an object-valued and a int-valued argument, and returns no result.

See dzone.com/articles/whats-wrong-java-8-part-ii

Overview of Common Functional Interfaces

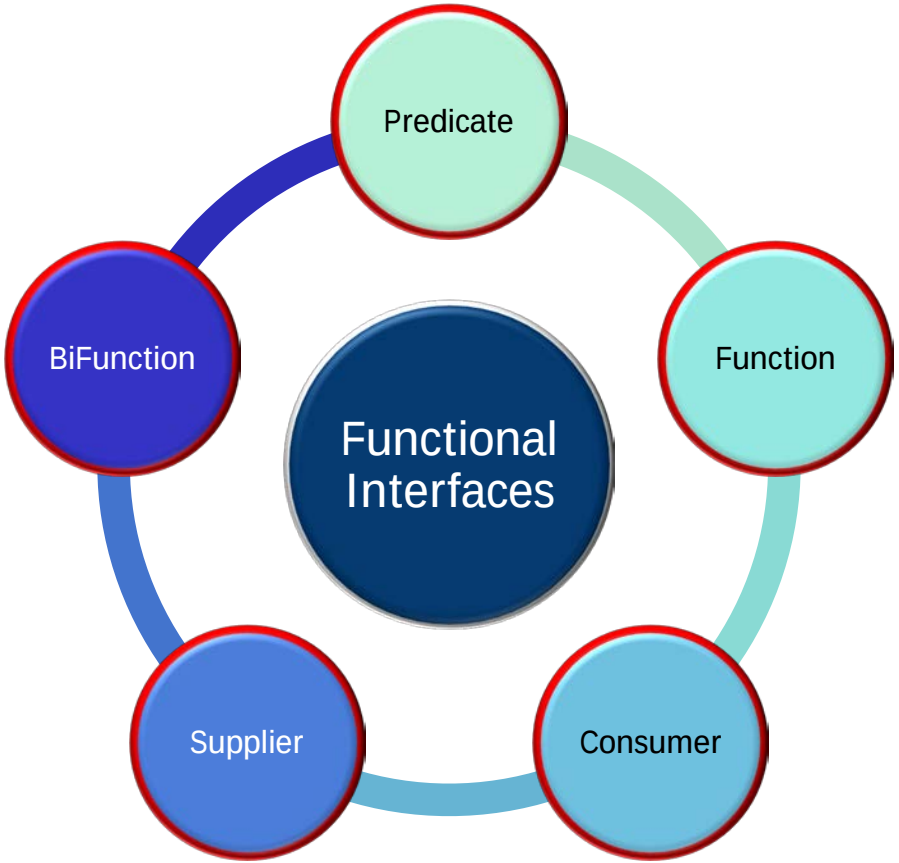
- Java 8 defines many types of functional interfaces
  - This list is large due to the need to support reference types & primitive types..

We focus on the most common types of functional interfaces



Overview of Common Functional Interfaces

- Java 8 defines many types of functional interfaces
  - This list is large due to the need to support reference types & primitive types..



All the functional interfaces in the upcoming examples are “stateless”!

Overview of Functional Interfaces: Predicate, Function, & BiFunction

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

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

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

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

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

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

System.out.println(iqMap);

iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);

System.out.println(iqMap);
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex10

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

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

System.out.println(iqMap);

iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);

System.out.println(iqMap);
```

This predicate lambda deletes entries with iq <= 100

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

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

System.out.println(iqMap);

iqMap.entrySet().removeIf(entry -> entry.getValue() <= 100);

System.out.println(iqMap);
```

entry is short for (EntrySet entry), which leverages the type inference capabilities of Java 8's compiler

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

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

```
interface Collection<T> {
  ...
  default boolean removeIf(Predicate<? super E> filter) {
    ...
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
      if (filter.test(each.next())) {
        each.remove();
      ...
```

Here's how the removeIf() method uses the Predicate passed to it

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

```
interface Collection<T> {
  ...
  default boolean removeIf(Predicate<? super E> filter) {
    ...
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
      if (filter.test(each.next())) {
        each.remove();
      ...
```

```
entry ->
entry.getValue()
<= 100
```

This predicate lambda returns true if an element is <= too the value 100

Overview of Common Functional Interfaces: Predicate

- A *Predicate* performs a test that returns true or false, e.g.,
  - `public interface Predicate<T> { boolean test(T t); }`

```
interface Collection<T> {
  ...
  default boolean removeIf(Predicate<? super E> filter) {
    ...
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
      if (filter.test(each.next())) {
        each.remove();
      ...
```

```
entry ->
entry.getValue()
<= 100
```

```
if (each.next().getValue() <= 100)
```

The ‘entry’ in the lambda predicate is replaced by the parameter to test()

Overview of Common Functional Interfaces: Function

- 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 Common Functional Interfaces: Function

- 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 Common Functional Interfaces: Function

- 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) {
  ... // Determines if a number if prime
}
```

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

Overview of Common Functional Interfaces: Function

- 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) {
  ... // Determines if a number if prime
}
```

This method provides atomic "check then act" semantics

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

Overview of Common Functional Interfaces: Function

- 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) {
  ... // Determines if a number if prime
}
```

A lambda expression that calls a function

Overview of Common Functional Interfaces: Function

- 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) {
  ... // Determines if a number if prime
}
```

Could also be a passed as a method reference

Overview of Common Functional Interfaces: Function

- 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 the computeIfAbsent() method uses the function passed to it

Overview of Common Functional Interfaces: Function

- 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);
      ...
```

this::primeChecker

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

Overview of Common Functional Interfaces: Function

- 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);
      ...
```

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

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

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); }`

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 parameters & returns a result, e.g.,
  - `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 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.

```
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);
```

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

vs.

```
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.

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

BiFunctional lambda subtracts 50 IQ points from each person in map

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

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)
        ...
```

Here's how the replaceAll() method uses the BiFunction passed to it

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)
        ...
```

`(k, v) -> v - 50`

`V newValue = oldValue - 50`

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

Overview of Functional Interfaces: Consumer & Supplier

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`

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

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`

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

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`

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

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`

```
List<Thread> threads =
  Arrays.asList(new Thread("Larry"),
                new Thread("Curly"),
                new Thread("Moe"));
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

Print out threads using forEach()

See docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html#forEach-java.util.function.Consumer-

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`

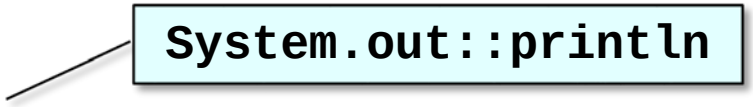
```
public interface Iterable<T> {
  ...
  default void forEach(Consumer<? super T> action) {
    for (T t : this) {
      action.accept(t);
    }
  }
```

Here's how the forEach() method uses the Consumer passed to it

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`

```
public interface Iterable<T> {
  ...
  default void forEach(Consumer<? super T> action) {
    for (T t : this) {
      action.accept(t);
    }
  }
```



The consumer lambda parameter is bound to println() on the System.out field

Overview of Common Functional Interfaces: Consumer

- A *Consumer* accepts a parameter & returns no results, e.g.,
  - `public interface Consumer<T> { void accept(T t); }`

```
public interface Iterable<T> {
  ...
  default void forEach(Consumer<? super T> action) {
    for (T t : this) {
      action.accept(t);
    }
  }
```

`System.out.println(t)`

The accept() method is replaced by the call to System.out.println()

Overview of Common Functional Interfaces: Consumer

- This example also shows a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
List<Thread> threads =
   new ArrayList<>(Arrays.asList(new Thread("Larry"),
                                 new Thread("Curly"),
                                 new Thread("Moe")));
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

Overview of Common Functional Interfaces: Consumer

- This example also shows a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
List<Thread> threads =
   new ArrayList<>(Arrays.asList(new Thread("Larry"),
                                 new Thread("Curly"),
                                 new Thread("Moe")));
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

A method reference used to sort threads by name

Overview of Common Functional Interfaces: Consumer

- This example also shows a *Function*, e.g.,
  - `public interface Function<T, R> { R apply(T t); }`

```
List<Thread> threads =
   new ArrayList<>(Arrays.asList(new Thread("Larry"),
                                 new Thread("Curly"),
                                 new Thread("Moe")));
threads.forEach(System.out::println);
threads.sort(Comparator.comparing(Thread::getName));
threads.forEach(System.out::println);
```

A method reference used to sort threads by name

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

Overview of Common Functional Interfaces: Consumer

- This example also shows 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)); }
```

Here's how the comparing() method uses the Function passed to it

Overview of Common Functional Interfaces: Consumer

- This example also shows 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

Overview of Common Functional Interfaces: Consumer

- This example also shows 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 getName() methods are then called to compare the two thread names

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

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

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

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

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };

String being = ...;

Optional<String> disposition =
  Optional.ofNullable(beingMap.get(being));

System.out.println("disposition of "
                   + being + " = "
                   + disposition.orElseGet(() -> "unknown"));
```

Returns default value if being not found

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex6

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

```
Map<String, String> beingMap = new HashMap<String, String>()
{ { put("Demon", "Naughty"); put("Angel", "Nice"); } };

String being = ...;

Optional<String> disposition =
  Optional.ofNullable(beingMap.get(being));

System.out.println("disposition of "
                   + being + " = "
                   + disposition.orElseGet(() -> "unknown"));
```

Returns default value if being is not found

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

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

```
class Optional<T> {
  ...
  public T orElseGet(Supplier<? extends T> other) {
    return value != null
      ? value
      : other.get();
  }
```

Here's how the orElseGet() method uses the Supplier passed to it

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

```
class Optional<T> {
  ...
  public T orElseGet(Supplier<? extends T> other) {
    return value != null
      ? value
      : other.get();
  }
```

```
() -> "unknown"
```

The string literal "unknown" is bound to the supplier lambda parameter

Overview of Common Functional Interfaces: Supplier

- A *Supplier* returns a value & takes no parameters, e.g.,
  - `public interface Supplier<T> { T get(); }`

```
class Optional<T> {
  ...
  public T orElseGet(Supplier<? extends T> other) {
    return value != null
      ? value
      : other.get();
  }
```

`() -> "unknown"`

`: "unknown"`

The string "unknown" returns by orElseGet() if the value is null

Overview of Common Functional Interfaces: Supplier

- A constructor reference is also a *Supplier*, e.g.,
  - `public interface Supplier<T> { T get(); }`

```
class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }

    private String hello() {
        return "hello";
    }
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex7

Overview of Common Functional Interfaces: Supplier

- A constructor reference is also a *Supplier*, e.g.,
  - `public interface Supplier<T> { T get(); }`

```
class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }

    private String hello() {
        return "hello";
    }
}
```

Create a supplier object that's initialized with a constructor reference for class CrDemo

See www.speakingcs.com/2014/08/constructor-references-in-java-8.html

Overview of Common Functional Interfaces: Supplier

- A constructor reference is also a *Supplier*, e.g.,
  - `public interface Supplier<T> { T get(); }`

```
class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }

    private String hello() {
        return "hello";
    }
}
```

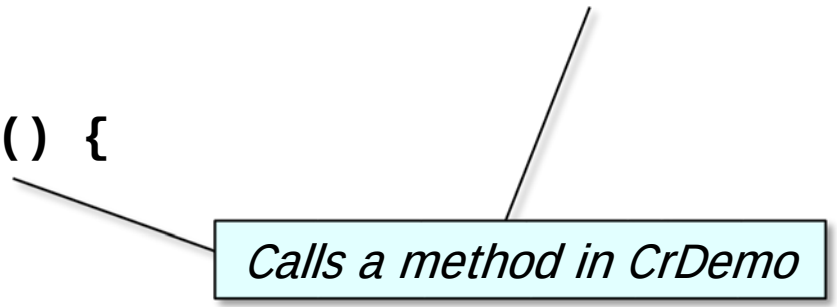
get() creates a CrDemo object using the constructor reference

Overview of Common Functional Interfaces: Supplier

- A constructor reference is also a *Supplier*, e.g.,
  - `public interface Supplier<T> { T get(); }`

```
class CrDemo {
    public static void main(String[] argv) {
        Supplier<CrDemo> supplier = CrDemo::new;
        System.out.println(supplier.get().hello());
    }

    private String hello() {
        return "hello";
    }
}
```



Other Properties of Functional Interfaces

Other Properties of Functional Interfaces

- A functional interface may also have default methods or static methods

```
interface Comparator {
  int compare(T o1, T o2);

  boolean equals(Object obj);

  default Comparator<T> reversed()
  { return Collections.reverseOrder(this); }

  static <T extends Comparable<? super T>>
  Comparator<T> reverseOrder()
  { return Collections.reverseOrder(); }
  ...
```

See docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

Other Properties of Functional Interfaces

- A functional interface may also have default methods or static methods, e.g.

```
interface Comparator {
  int compare(T o1, T o2);

  boolean equals(Object obj);

  default Comparator<T> reversed()
  { return Collections.reverseOrder(this); }

  static <T extends Comparable<? super T>>
  Comparator<T> reverseOrder()
  { return Collections.reverseOrder(); }
  ...
```

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

Other Properties of Functional Interfaces

- A functional interface may also have default methods or static methods, e.g.,

```
interface Comparator {
  int compare(T o1, T o2);

  boolean equals(Object obj);

  default Comparator<T> reversed()
  { return Collections.reverseOrder(this); }

  static <T extends Comparable<? super T>>
  Comparator<T> reverseOrder()
  { return Collections.reverseOrder(); }
  ...
```

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

Other Properties of Functional Interfaces

- A functional interface may also have default methods or static methods, e.g.,

```
interface Comparator {
  int compare(T o1, T o2);

  boolean equals(Object obj);

  default Comparator<T> reversed()
  { return Collections.reverseOrder(this); }

  static <T extends Comparable<? super T>>
  Comparator<T> reverseOrder()
  { return Collections.reverseOrder(); }
  ...
```

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

Other Properties of Functional Interfaces

- A functional interface may also have default methods or static methods, e.g.,

```
interface Comparator {
  int compare(T o1, T o2);

  boolean equals(Object obj);

  default Comparator<T> reversed()
  { return Collections.reverseOrder(this); }

  static <T extends Comparable<? super T>>
  Comparator<T> reverseOrder()
  { return Collections.reverseOrder(); }
  ...
```

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

Other Properties of Functional Interfaces

- A functional interface may also have default methods or static methods, e.g.,

```
interface Comparator {
  int compare(T o1, T o2);

  boolean equals(Object obj);

  default Comparator<T> reversed()
  { return Collections.reverseOrder(this); }

  static <T extends Comparable<? super T>>
  Comparator<T> reverseOrder()
  { return Collections.reverseOrder(); }
  ...
```

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

Other Properties of Functional Interfaces

- A functional interface may also have default methods or static methods, e.g.,

```
interface Comparator {
  int compare(T o1, T o2);

  boolean equals(Object obj);

  default Comparator<T> reversed()
  { return Collections.reverseOrder(this); }

  static <T extends Comparable<? super T>>
  Comparator<T> reverseOrder()
  { return Collections.reverseOrder(); }
  ...
```

An abstract method that overrides a public java.lang.Object method does not count as part of the interface's abstract method count

See docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html

End of Overview of Java 8 Functional Interfaces