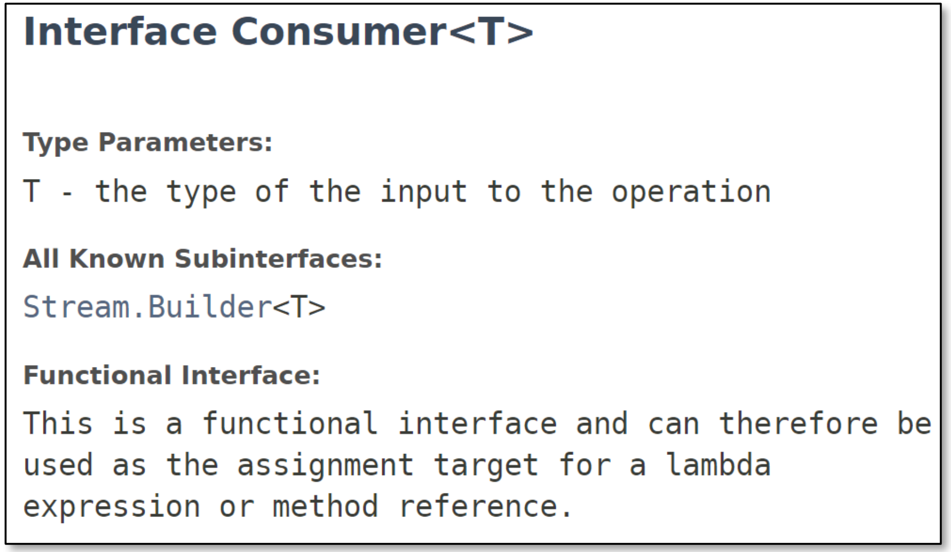
# Java 8 Functional Interfaces

## Consumer

Douglas C. Schmidt

## Learning Objectives in This Lesson

- Recognize foundational functional programming features in Java 8, e.g.,
  - Lambda expressions
  - Method & constructor references
  - Key functional interfaces
    - Predicate
    - Function
    - BiFunction
    - Supplier
    - Consumer



**Interface Consumer<T>**

**Type Parameters:**

T - the type of the input to the operation

**All Known Subinterfaces:**

Stream.Builder<T>

**Functional Interface:**

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

Douglas C. Schmidt

# Overview of Functional Interfaces: 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); }`

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); }`**

*Consumer is a generic interface that is parameterized by one reference type*

See [docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html](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); }`

*Its single abstract method is passed one parameter & returns nothing*

See [docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html](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"));
```

*Create a list of threads with the names of 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](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"));
```

*Print out threads using forEach()*

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

See docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html#forEach

# 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);
    }
  }
```

`System.out::println`

The consumer parameter is bound to the System.out::println method reference.

# 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().

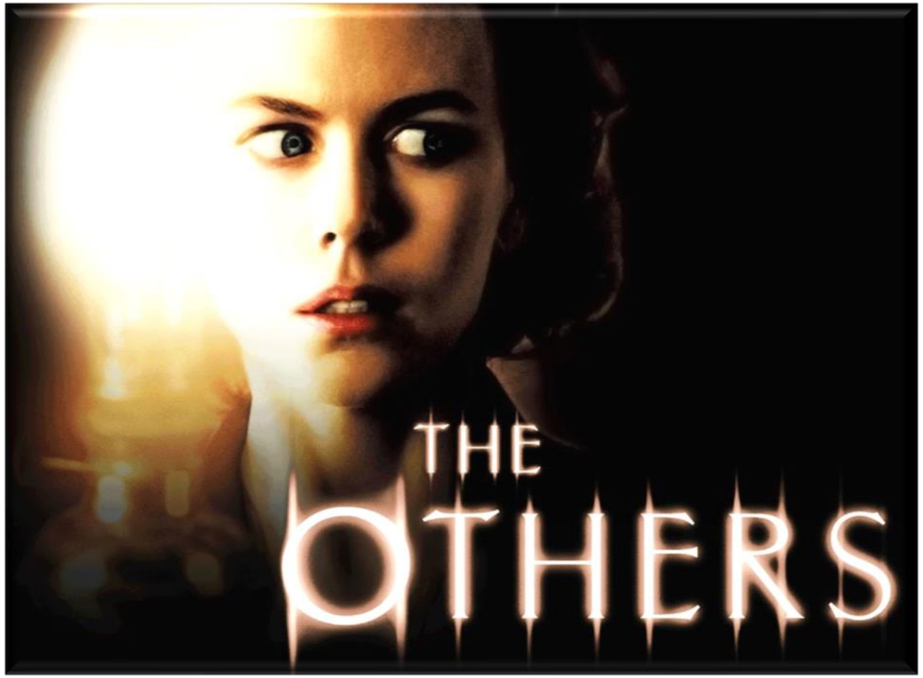
# Java 8 Functional Interfaces

## Other Properties

Douglas C. Schmidt

## Learning Objectives in This Lesson

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



Douglas C. Schmidt

# Other Properties of Functional Interfaces

## Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods

```
@FunctionalInterface
public interface Comparator<T> {
  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

- Functional interfaces may also have default methods and/or static methods, e.g.,

```
@FunctionalInterface
public interface Comparator<T> {
  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(); }
  ...
```

*A comparison function that imposes a total ordering on some collection of objects.*

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

## Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.,

```
@FunctionalInterface
public interface Comparator<T> {
  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(); }
  ...
```

*This annotation type indicates that this interface type declaration is intended as a functional interface.*

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

# Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.,

```
@FunctionalInterface
public interface Comparator<T> {
  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(); }
  ...
```

*The primary abstract method in this functional interface.*

# Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.,

```
@FunctionalInterface
public interface Comparator<T> {
  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(); }
  ...
```

*A default method provides the default implementation, which can be overridden.*

See earlier lesson on "*Overview of Functional Interfaces: Function*"

# Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.,

```
@FunctionalInterface
public interface Comparator<T> {
  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(); }
  ...
```

*This static method provides the one-and-only implementation.*

## Other Properties of Functional Interfaces

- Functional interfaces may also have default methods and/or static methods, e.g.,

```
@FunctionalInterface
public interface Comparator<T> {
  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](docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)