Overview of Java 8 Lambda Expressions & Method References

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
Learning Objectives in this Lesson

• Recognize foundational functional programming features in Java 8, e.g.,
  • Lambda expressions
  • Method (& constructor) references
Learning Objectives in this Lesson

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

Several concise examples are used to showcase foundational Java 8 features
Overview of Lambda Expressions
Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later

```java
new Thread(() ->
    System.out.println("hello world"))
    .start();
```

Overview of Lambda Expressions

A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread(() ->
    System.out.println("hello world");
).start();
```

This lambda expression takes no parameters, i.e., "()"
Overview of Lambda Expressions

• A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread(() ->
    System.out.println("hello world"))
    .start();
```

*It defines a computation that will run in a separate Java thread*

See [docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html](http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html)
Overview of Lambda Expressions

- A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread() ->
    System.out.println("hello world")
    .start();
```

*It defines a computation that will run in a separate Java thread*

See [docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html](http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html)
Overview of Lambda Expressions

• A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread(() ->
    System.out.println("hello world"))
    .start();
```

Runnable `r = () -> System.out.println("hello world");`
new Thread(r).start();

*You can also store a lambda expression into a variable & pass that variable to a method*

See docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html
A lambda expression is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread(() ->
          System.out.println("hello world"))
     .start();
```

Lambda expressions are compact since they just focus on computation(s) to perform.
A *lambda expression* is an unnamed block of code (with optional parameters) that can be stored, passed around, & executed later, e.g.,

```java
new Thread(() ->
    System.out.println("hello world"))
    .start();
```

VS

Conversely, this anonymous inner class requires more code to write each time

```java
new Thread(new Runnable() {
    public void run() {
        System.out.println("hello world");
    }
}).start();
```
Overview of Lambda Expressions

- Lambda expressions can work with multiple parameters in a more compact manner than anonymous inner classes

```java

Arrays.sort(nameArray, new Comparator<String>(){
    public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```java
Arrays.sort(nameArray,
    (s, t) -> s.compareToIgnoreCase(t));
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex1](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex1)
Overview of Lambda Expressions

- Lambda expressions can work with multiple parameters in a more compact manner than anonymous inner classes, e.g.,

```java

Arrays.sort(nameArray,
    new Comparator<String>(){
        public int compare(String s,String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }
    });

VS

Arrays.sort(nameArray, (s, t) -> s.compareToIgnoreCase(t));
```
Lambda expressions can work with multiple parameters in a more compact manner than anonymous inner classes, e.g.,

```java
Arrays.sort(nameArray, new Comparator<String>(){
    public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }});
```

Extraneous syntax for anonymous inner class
Overview of Lambda Expressions

- Lambda expressions can work with multiple parameters in a more compact manner than anonymous inner classes, e.g.,

```java

Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }
});
```

VS

```java
Arrays.sort(nameArray, (s, t) -> s.compareToIgnoreCase(t));
```

*(s, t) is short for (String s, String t), which leverages Java 8’s type inference capabilities*

See docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html
Overview of Lambda Expressions

• Lambda expressions can work with multiple parameters in a more compact manner than anonymous inner classes, e.g.,

```java

Arrays.sort(nameArray, new Comparator<String>(){
    public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }});
```

VS

```java
Arrays.sort(nameArray, (s, t) -> s.compareToIgnoreCase(t));
```

*This lambda expression omits the method name & extraneous syntax*
Overview of Lambda Expressions

• A lambda expression can access (effectively) final variables from the enclosing scope

```java
int answer = 42;
new Thread(() ->
    System.out.println("The answer is " + answer))
    .start();
```

This lambda expression can access the value of "answer," which is an effectively final variable whose value never changes after it’s initialized.

See [www.linkedin.com/pulse/java-8-effective-final-gaurhari-dass](www.linkedin.com/pulse/java-8-effective-final-gaurhari-dass)
Overview of Method References
## Overview of Method References

A method reference is a compact, easy-to-read handle for a method that already has a name.

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Reference to a static method</td>
<td>ContainingClass::staticMethodName</td>
</tr>
<tr>
<td>2. Reference to an instance method of a particular object</td>
<td>containingObject::instanceMethodName</td>
</tr>
<tr>
<td>3. Reference to an instance method of an arbitrary object of a given type</td>
<td>ContainingType::methodName</td>
</tr>
<tr>
<td>4. Reference to a constructor</td>
<td>ClassName::new</td>
</tr>
</tbody>
</table>

See [docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html](docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html)
Overview of Method References

- A method reference is a compact, easy-to-read handle for a method that already has a name

<table>
<thead>
<tr>
<th>Kind</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Reference to a static method</td>
<td>ContainingClass::staticMethodName</td>
</tr>
<tr>
<td>2. Reference to an instance method of a particular object</td>
<td>containingObject::instanceMethodName</td>
</tr>
<tr>
<td>3. Reference to an instance method of an arbitrary object of a given type</td>
<td>ContainingType::methodName</td>
</tr>
<tr>
<td>4. Reference to a constructor</td>
<td>ClassName::new</td>
</tr>
</tbody>
</table>
Overview of Method References

- Method references are even more compact than anonymous inner classes & lambda expressions
Overview of Method References

- Method references are more compact than alternative mechanisms, e.g.,

```java

Arrays.sort(nameArray, new Comparator<String>() {
    public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }
});

VS

Arrays.sort(nameArray,
    (s, t) -> s.compareToIgnoreCase(t));

VS

Method reference is even more compact

Arrays.sort(nameArray, String::compareToIgnoreCase);
```
Overview of Method References

- Method references are more compact than alternative mechanisms, e.g.,

  ```java
  Arrays.sort(nameArray, new Comparator<String>() {
      public int compare(String s, String t) { return s.toLowerCase().compareTo(t.toLowerCase()); }
  });
  ```

  VS

  ```java
  Arrays.sort(nameArray, (s, t) -> s.compareToIgnoreCase(t));
  ```

  VS

  ```java
  Arrays.sort(nameArray, String::compareToIgnoreCase);
  ```

It’s good practice to use method references whenever you can!
Applying Method References in Practice
Applying Method References in Practice

- Method references can be used to print a collection or array in various ways

```java
String[] nameArray = {
    "Barbara", "James", "Mary", "John", 
    "Robert", "Michael", "Linda", "james", "mary"};
```

List of names represented as strings
Method references can be used to print a collection or array in various ways.

```java
```

System.out.println() can be used to print out an array.

```java
System.out.println(Arrays.asList(nameArray));
```

prints

```
[Barbara, James, Mary, John, Linda, Michael, Linda, james, mary]
```

See docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html#println
Applying Method References in Practice

- Method references can be used to print a collection or array in various ways
  ```java
  ```

- System.out.println() can be used to print out an array
  ```java
  System.out.println(Arrays.asList(nameArray));
  ```
  prints 
  
  Returns a fixed-size list backed by the specified array
  
  [Barbara, James, Mary, John, Linda, Michael, Linda, james, mary]

See [www.tutorialspoint.com/java/util/arrays_aslist.htm](http://www.tutorialspoint.com/java/util/arrays_aslist.htm)
Method references can be used to print a collection or array in various ways:

```java
```

- System.out.println() can be used to print out an array.
- Java 8’s forEach() method can be used to print out values of an array.

Method references can be used to print a collection or array in various ways. For example:

```java

• System.out.println() can be used to print out an array.

• Java 8’s forEach() method can be used to print out values of an array, e.g.

  In conjunction with a stream & method reference:

  ```java
  Stream.of(nameArray).forEach(System.out::print);
  ```

  prints

  ```java
  Factory method that creates a stream from an array
  ```

  BarbaraJamesMaryJohnLindaMichaelLindaJamesMary

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#of](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#of)
Method references can be used to print a collection or array in various ways:

```java

System.out.println();
Java 8’s forEach() method can be used to print out values of an array, e.g.

In conjunction with a stream & method reference:

```java
Stream.of(nameArray).forEach(System.out::print);
``` 

prints

BarbaraJamesMaryJohnLindaMichaelLindajamesmary

Perform method reference action on each stream element

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach](http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach)
Method references can be used to print a collection or array in various ways.

```java
String[] nameArray = {"Barbara", "James", "Mary", "John", 
                      "Robert", "Michael", "Linda", "james", "mary"};
```

- `System.out.println()` can be used to print out an array.
- Java 8’s `forEach()` method can be used to print out values of an array, e.g.
  - In conjunction with a stream & method reference
  - In conjunction with a collection (e.g., List)

```java
Arrays.asList(nameArray).forEach(System.out::print);
```

prints

```
Barbara James Mary John Linda Michael Linda James mary
```

See [docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList](http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList)
Applying Method References in Practice

- Method references can be used to print a collection or array in various ways
  ```java
  ```
- System.out.println() can be used to print out an array
- Java 8’s forEach() method can be used to print out values of an array, e.g.
  - In conjunction with a stream & method reference
    ```java
    Arrays.asList(nameArray).forEach(System.out::print);
    ```
  - In conjunction with a collection (e.g., List)
    ```java
    System.out.println(Arrays.asList(nameArray));
    ```
    prints
    ```
    BarbaraJamesMaryJohnLindaMichaelLindajamesmary
    ```

See [docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html#forEach](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html#forEach)
Method references can be used to print a collection or array in various ways. Here’s an example:

```java
String[] nameArray = {
    "Barbara", "James", "Mary", "John", 
    "Robert", "Michael", "Linda", "james", "mary"};
```

- `System.out.println()` can be used to print out an array.
- Java 8’s `forEach()` method can be used to print out values of an array, e.g.:
  - In conjunction with a stream & method reference
  - In conjunction with a collection (e.g., List)
  - `forEach()` on a stream differs slightly from `forEach()` on a collection

See [stackoverflow.com/a/23232560](http://stackoverflow.com/a/23232560)
**Applying Method References in Practice**

- Method references can be used to print a collection or array in various ways
  ```java
  ```
  - `System.out.println()` can be used to print out an array
  - Java 8’s `forEach()` method can be used to print out values of an array, e.g.
    - In conjunction with a stream & method reference
    - In conjunction with a collection (e.g., List)
  - `forEach()` on a stream differs slight from `forEach()` on a collection
    - e.g., `forEach()` ordering is undefined on a stream, whereas it’s defined for a collection

See [stackoverflow.com/a/23232560](http://stackoverflow.com/a/23232560)
Implementing Closures with Lambda Expressions
Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”

```java
class ClosureExample {
    private int mRes;

    Thread makeThreadClosure(String s, int n) {
        return new Thread(() ->
            System.out.println(s + (mRes += n)));
    }

    ClosureExample() throws InterruptedException {
        Thread t = makeThreadClosure("result = ", 10);
        t.start(); t.join();
    }
}
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex1](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex1)
Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”
- A closure is a method that has an environment w/at least 1 bound variable

```java
class ClosureExample {
    private int mRes;

    Thread makeThreadClosure(String s, int n) {
        return new Thread(() -> {
            System.out.println(s + (mRes += n));
        });
    }

    ClosureExample() throws InterruptedException {
        Thread t = makeThreadClosure("result = ", 10);
        t.start(); t.join();
    }
}
```

See [en.wikipedia.org/wiki/Closure_(computer_programming)](en.wikipedia.org/wiki/Closure_(computer_programming))
Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”
- A closure is a method that has an environment w/ at least 1 bound variable

```java
class ClosureExample {
    private int mRes;

    Thread makeThreadClosure(String s, int n) {
        return new Thread(() ->
            System.out.println(s + (mRes += n)));
    }

    ClosureExample() throw InterruptedException {
        Thread t = makeThreadClosure("result = ", 10);
        t.start(); t.join();
    }
}
```

A bound variable is name that has a value, such as a number
Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”
- A closure is a method that has an environment w/ at least 1 bound variable

```java
class ClosureExample {
    private int mRes;

    Thread makeThreadClosure(String s, int n) {
        return new Thread(() -> System.out.println(s + (mRes += n)));
    }

    ClosureExample() throws InterruptedException {
        Thread t = makeThreadClosure("result = ", 10);
        t.start(); t.join();
    }
}
```

This lambda implements a closure that captures a private field & method params

See bruceeckel.github.io/2015/10/17/are-java-8-lambdas-closures
Implementing Closures with Lambda Expressions

- Lambda expressions can implement (simplified) variants of “closures”
- A closure is a method that has an environment w/at least 1 bound variable

```java
class ClosureExample {
    private int mRes;

    Thread makeThreadClosure(String s, int n) {
        return new Thread(()
                -> System.out.println(s + (mRes += n)));
    }

    ClosureExample() throw InterruptedException {
        Thread t = makeThreadClosure("result = ", 10);
        t.start(); t.join();
    }
}
```

See [dzone.com/articles/java-8-lambdas-limitations-closures](dzone.com/articles/java-8-lambdas-limitations-closures)

Values of private fields can be updated in a lambda, but not params or local vars (which are read-only)
Lambda expressions can implement (simplified) variants of “closures”

A closure is a method that has an environment w/at least 1 bound variable

```java
class ClosureExample {
    private int mRes;

    Thread makeThreadClosure(String s, int n) {
        return new Thread(() -> {
            System.out.println(s + (mRes += n));
        });
    }

    ClosureExample() throw InterruptedException {
        Thread t = makeThreadClosure("result = ", 10);
        t.start(); t.join();
    }
}
```

Constructor creates a closure & runs it in a background thread
End of Overview of Java 8 Lambda Expressions & Method References