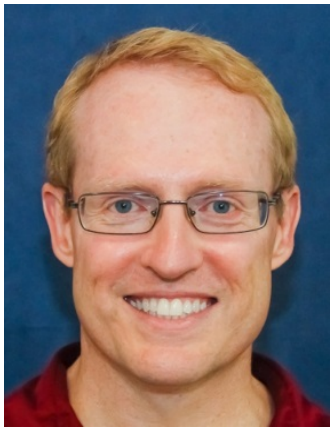
Overview of Java 8 Lamba Expressions & Method References

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

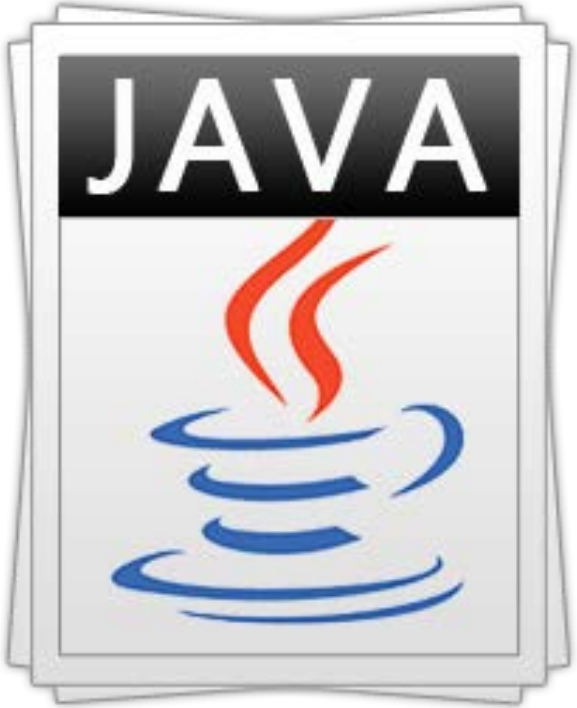
- Recognize foundational functional programming features in Java 8, e.g.,
  - Lambda expressions

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 & Method References

Overview of Lambda Expressions

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

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

See www.drdobbs.com/jvm/lambda-expressions-in-java-8/240166764

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.,

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

This lambda expression takes no parameters (i.e., "()") & defines a computation that will run in a separate thread

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.,

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

Lambda expressions are compact since they just focus on computation(s) to perform

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.,

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

vs

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

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

Overview of Method References

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

Kind	Example
1. Reference to a static method	ContainingClass::staticMethodName
2. Reference to an instance method of a particular object	containingObject::InstanceMethodName
3. Reference to an instance method of an arbitrary object of a given type	ContainingType::methodName
4. Reference to a constructor	ClassName::new

See 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

Kind	Example
1. Reference to a static method	ContainingClass::staticMethodName
2. Reference to an instance method of a particular object	containingObject::InstanceMethodName
3. Reference to an instance method of an arbitrary object of a given type	ContainingType::methodName
4. Reference to a constructor	ClassName::new

Overview of Method References

- Method references are more compact than anonymous inner classes & lambda expressions, e.g.,

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

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

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

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex1

Overview of Method References

- Method references are more compact than anonymous inner classes & lambda expressions, e.g.,

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

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

Lots of syntax for anonymous inner class

vs

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

vs

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

Overview of Method References

- Method references are more compact than anonymous inner classes & lambda expressions, e.g.,

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

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

vs

Lambda expression omits name & syntax

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

vs

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

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

Overview of Method References

- Method references are more compact than anonymous inner classes & lambda expressions, e.g.,

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

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 anonymous inner classes & lambda expressions, e.g.,

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

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

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

It's generally a good idea to use method references whenever you can

Overview of Method References

- The contents of a collection or array can be printed in various ways

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

Overview of Method References

- The contents of a collection or array can be printed in various ways

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

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

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

    prints

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

See hwww.tutorialspoint.com/java/util/arrays_aslist.htm

Overview of Method References

- The contents of a collection or array can be printed in various ways

```
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() loop can be used in conjunction with a stream & method reference

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

    prints

```
BarbaraJamesMaryJohnLindaMichaelLindajamesmary
```

See www.javaworld.com/article/2461744/java-language/java-language-iterating-over-collections-in-java-8.html

End of Overview of Java 8 Lambda Expressions & Method References