Introduction to Java 8

- Dates and Times
- Default Interface Methods
- Functional Programming
- Parallel Programming

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java 8 Feature Overview

- Java 8 is a significant update to earlier versions of Java and provides several major additions, including:
  - **New Date and Time API** – Classes to make working with dates and times easier and safer
  - **Default interfaces methods** – Syntax to add concrete methods to interfaces
  - **Functional programming** – Syntax to make methods **first class citizens**
  - **Parallel processing** – Classes to make writing parallel algorithms easier
  - **JavaScript integration** – A new, more powerful JavaScript engine in the JVM so you can execute JavaScripts from a Java program

---

For a complete listing of new Java 8 features, see:
http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Java 8 Date/Time API

- The Java date/time/calendar API has long been tedious and difficult to use
- Java 8 introduces the **java.time** package that includes classes such as **LocalDate** and **LocalTime**

```
1    LocalDate dob = LocalDate.of(1994, 8, 27);
2    System.out.println(dob);
3
4    LocalTime appointment = LocalTime.of(13, 30);
5    System.out.println(appointment);
```

The Java 8 Date/Time API was inspired by the popular Joda Time library.

This new API was introduced as JSR-310 before being incorporated into Java 8.

Unlike the old java.util.Date and Calendar, objects of these classes are immutable, and thus are thread-safe.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Default Methods

- During the life of a project, sometimes it's useful to add new functionality to base interfaces
- Java 8 provides **default methods** also known as **extension methods** that let you add concrete methods to existing interfaces without breaking existing classes that implement the interfaces

MyInterface.java

MyClass.java

TestDefaultMethods.java

Java 8 also lets you define static methods in interfaces. As usual, static methods in interfaces are considered part of the interface itself in memory, not part of the instances of classes that implement the interface.

The Java 8 class library uses default and static methods to add new functionality to interfaces such as java.util.Comparator to support other new Java 8 functionality such as streams.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package defaultmethods;
2
3    public interface MyInterface
4    {
5        void method1();
6
7        default void method2()
8        {
9            System.out.println("Method 2");
10       }
11
12       static void method3()
13       {
14           System.out.println("Method 3");
15       }
16   }
```

```
1    package defaultmethods;
2
3    public class MyClass implements MyInterface
4    {
5        @Override
6        public void method1()
7        {
8            System.out.println("Method 1");
9        }
10
11   }
```

```
1    package defaultmethods;
2
3    public class TestDefaultMethods
4    {
5        public static void main(String[] args)
6        {
7            MyClass m = new MyClass();
8
9            m.method1();
10           m.method2();
11           MyInterface.method3();
12       }
13   }
```

Introduction to Functional Programming

- **Functional programming** is a discipline in which you write programs that consist of functions that don't modify any external state, and thus have no **side effects**
- Languages that provide functional programming features include **Lisp**, **Scheme**, **Haskell**, **Scala** and **Clojure**
- A complete discussion of functional programming is beyond the scope of this course, but we will cover **lambda expressions** and other Java 8 functional-programming support

---

Many other "imperative languages" like Java include support for functional techniques. For example, JavaScript provides several features that enable functional programming including anonymous functions and closures.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is a Lambda Expression?

- A **lambda expression** is an anonymous function (method) that you can define, then pass as a parameter to other methods
- Prior to Java 8, developers often used **anonymous inner classes** with a single method for this purpose, but that syntax is verbose and perhaps confusing

```
1    JButton jb = new JButton("press me");
2
3    // pass anonymous inner-class object as parameter
4    jb.addActionListener(new ActionListener()
5    {
6      public void actionPerformed(ActionEvent e)
7      {
8        System.out.println("I've been clicked!");
9      }
10   });
```

The term "lambda expression" originally came from mathematics, where the Greek letter lambda is used to define a mathematical function arguments.

The equivalent code using Java 8 lambda expressions:

```
jb.addActionListener((e) ->
{
  System.out.println("I've been clicked lambda!");
});
```

Alternatively:

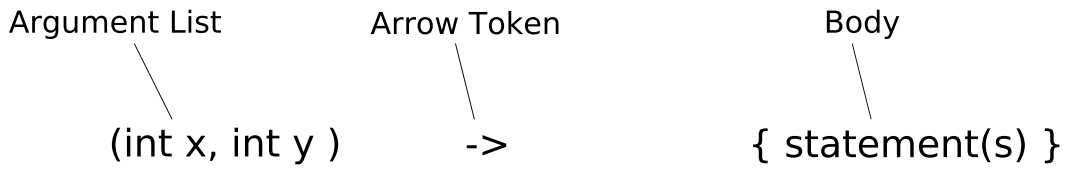
```
ActionListener al = (ae) -> System.out
  .println("I've been clicked lambda again");

jb.addActionListener(al);
```

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Lambda Expression Syntax

- Lambda expressions consist of an **argument list**, an **arrow token** and a **body**




```
1    n -> n % 2 != 0
2    (char c) -> c == 'y'
3    (x, y) -> x + y
4    (int x, int y) -> x * x + y * y
5    () -> 42;
6    () -> { return 3.14; }
7    (String s) -> { System.out.println(s); }
8    s -> System.out.println(s);
9    () -> { System.out.println("Hello World!"); }
```

2 - 7

In the argument list, you can omit the parameter types, and if there's exactly one parameter, you can omit the parenethis.

If there's only one statement in the body, you can omit the curly braces.

You can also omit the return statement.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Functional Interfaces

- A **functional interface** is an interface with a single method
- You can use functional interfaces as types of variables and parameters and then provide a lambda expression

LambdaSyntax.java

The examples in the following listing show the lambda expressions from the previous page, assigning them to variables whose type are functional interfaces.

Note that the java.awt.event.ActionListener we used a couple of pages ago is a functional interface with a single method named actionPerformed().

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package lambdas;
2
3    public class LambdaSyntax
4    {
5        interface Functional1
6        {
7            boolean method1(int x);
8        }
9
10       interface Functional2
11       {
12           boolean method1(char c);
13       }
14
15       interface Functional3<T>
16       {
17           T method1(T a, T b);
18       }
19
20       interface Functional4
21       {
22           int method1(int a, int b);
23       }
24
25       interface Functional5
26       {
27           int method1();
28       }
29
30       interface Functional6
31       {
32           double method1();
33       }
34
35       interface Functional7
36       {
37           void method1(String s);
38       }
39
40       interface Functional8
41       {
42           void method1();
43       }
44
45       public static void main(String[] args)
```

```
46      {
47          Functional1 a = n -> n % 2 != 0;
48          Functional2 b = (char c) -> c == 'y';
49          Functional3<Integer> c = (x, y) -> x + y;
50          Functional4 d = (int x, int y) -> x * x + y * y;
51          Functional5 e = () -> 42;
52          Functional6 f = () -> { return 3.14; };
53          Functional7 g = (String s) -> { System.out.println(s); };
54          Functional7 h = s -> System.out.println(s);
55          Functional8 i = () -> { System.out.println("Hello World!"); };
56      }
57  }
```

```
1    package lambdas;
2
3    public class LambdaSyntax
4    {
5        interface Functional1
6        {
7            boolean method1(int x);
8        }
9
10       interface Functional2
11       {
12           boolean method1(char c);
13       }
14
15       interface Functional3<T>
16       {
17           T method1(T a, T b);
18       }
19
20       interface Functional4
21       {
22           int method1(int a, int b);
23       }
24
25       interface Functional5
26       {
27           int method1();
28       }
29
30       interface Functional6
31       {
32           double method1();
33       }
34
35       interface Functional7
36       {
37           void method1(String s);
38       }
39
40       interface Functional8
41       {
42           void method1();
43       }
44
45       public static void main(String[] args)
```

```
46      {
47          Functional1 a = n -> n % 2 != 0;
48          Functional2 b = (char c) -> c == 'y';
49          Functional3<Integer> c = (x, y) -> x + y;
50          Functional4 d = (int x, int y) -> x * x + y * y;
51          Functional5 e = () -> 42;
52          Functional6 f = () -> { return 3.14; };
53          Functional7 g = (String s) -> { System.out.println(s); };
54          Functional7 h = s -> System.out.println(s);
55          Functional8 i = () -> { System.out.println("Hello World!"); };
56      }
57  }
```

Using Lambda Expressions Example

- In Java 8, **java.util.Comparator** can act as a functional interface so you can use lambdas to sort collections

Student.java

TestComparator.java

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package university;
2
3    import java.time.LocalDate;
4
5    public class Student implements Comparable<Student>
6    {
7        private int studentID;
8        private String name;
9        private double gpa;
10       private LocalDate dateOfBirth;
11
12       public Student(int studentID, String name, double gpa, LocalDate dateOfBirth)
13       {
14           this.studentID = studentID;
15           this.name = name;
16           this.gpa = gpa;
17           this.dateOfBirth = dateOfBirth;
18       }
19
20       @Override
21       public int compareTo(Student o)
22       {
23           return this.name.compareTo(o.name);
24       }
25
26       public int getStudentID()
27       {
28           return studentID;
29       }
30
31       public void setStudentID(int studentID)
32       {
33           this.studentID = studentID;
34       }
35
36       public String getName()
37       {
38           return name;
39       }
40
41       public void setName(String name)
42       {
43           this.name = name;
44       }
45
```

```
46      public double getGpa()
47      {
48          return gpa;
49      }
50
51      public void setGpa(double gpa)
52      {
53          this.gpa = gpa;
54      }
55
56      public LocalDate getDateOfBirth()
57      {
58          return dateOfBirth;
59      }
60
61      public void setDateOfBirth(LocalDate dateOfBirth)
62      {
63          this.dateOfBirth = dateOfBirth;
64      }
65
66      @Override
67      public String toString()
68      {
69          return "Student [studentID=" + studentID + ", name=" + name + ", gpa="
70                  + gpa + ", dateOfBirth=" + dateOfBirth + "]";
71      }
72
73      @Override
74      public int hashCode()
75      {
76          final int prime = 31;
77          int result = 1;
78          result = prime * result + studentID;
79          return result;
80      }
81
82      @Override
83      public boolean equals(Object obj)
84      {
85          if (this == obj)
86              return true;
87          if (obj == null)
88              return false;
89          if (getClass() != obj.getClass())
90              return false;
```

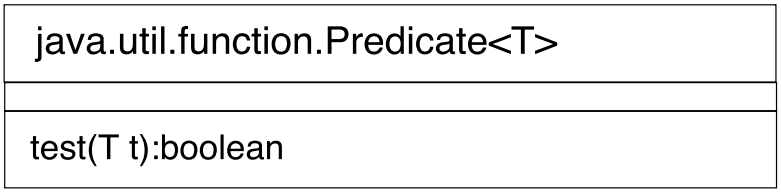
```
91            Student other = (Student) obj;
92            if (studentID != other.studentID)
93                return false;
94            return true;
95        }
96    }
```

```
1    package lambdas;
2
3    import java.time.LocalDate;
4    import java.util.Comparator;
5    import java.util.SortedSet;
6    import java.util.TreeSet;
7
8    import university.Student;
9
10   public class TestComparator
11   {
12       private static SortedSet<Student> students = new TreeSet<Student>();
13
14       private static void init()
15       {
16           students.add(new Student(12, "Sam Smith", 3.45, LocalDate.of(1994, 4,
17                   12)));
18           students.add(new Student(4, "Sue Jones", 4.00, LocalDate
19                   .of(1992, 6, 22)));
20           students.add(new Student(74, "Harry Wolfe", 3.33, LocalDate.of(1993, 1,
21                   18)));
22           students.add(new Student(22, "Paul Westerberg", 2.56, LocalDate.of(
23                   1994, 8, 27)));
24       }
25
26       public static void main(String[] args)
27       {
28           init();
29
30           // print using natural sort order
31           System.out.println("Natural order");
32           System.out.println("-------------");
33           for (Student s : students)
34               System.out.println(s);
35           System.out.println();
36
37           // print in descending GPA using a lambda
38           Comparator<Student> sorter = (Student s1, Student s2) -> {
39               int retVal = Double.compare(s1.getGpa(), s2.getGpa());
40               return retVal;
41           };
42           SortedSet<Student> sorted = new TreeSet<Student>(sorter);
43           sorted.addAll(students);
44           System.out.println("Sorted by GPA");
45           System.out.println("-------------");
```

```
46            for (Student s : sorted)
47                System.out.println(s);
48            System.out.println();
49        }
50    }
```

The java.util.function Package

- Java 8 introduces the **java.util.function** package that contains functional interfaces that are useful with lambda expressions
- One such interface is **Predicate**, which contains a single method, **test** that accepts an object and returns a boolean
- Predicates typically act as a filter for objects in a collection



The java.util.function.Predicate interface actually has a few more methods (e.g and(), or() and negate()), but they are default or static methods and thus don't violate the requirement that functional interfaces have a single method.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Predicate Example

- This example uses Predicates to process various subsets of a collection

TestPredicate.java

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package lambdas;
2
3    import java.time.LocalDate;
4    import java.util.SortedSet;
5    import java.util.TreeSet;
6    import java.util.function.Predicate;
7
8    import university.Student;
9
10   public class TestPredicate
11   {
12       private static SortedSet<Student> students = new TreeSet<Student>();
13
14       private static void init()
15       {
16           students.add(new Student(12, "Sam Smith", 3.45, LocalDate.of(1994, 4,
17                   12)));
18           students.add(new Student(4, "Sue Jones", 4.00, LocalDate
19                   .of(1992, 6, 22)));
20           students.add(new Student(74, "Harry Wolfe", 3.33, LocalDate.of(1993, 1,
21                   18)));
22           students.add(new Student(22, "Paul Westerberg", 2.56, LocalDate.of(
23                   1994, 8, 27)));
24       }
25
26       public static void print(Predicate<Student> p)
27       {
28           for (Student s : students)
29           {
30               if (p.test(s))
31                   System.out.println(s);
32           }
33       }
34
35       public static void main(String[] args)
36       {
37           init();
38
39           Predicate<Student> highGpa = aStudent -> aStudent.getGpa() > 3.4;
40           Predicate<Student> youngster = aStudent -> aStudent.getDateOfBirth()
41                   .getYear() >= 1994;
42
43           // print all high GPA students
44           System.out.println("High GPA students");
45           System.out.println("-------------");
```

```
46            print(highGpa);
47            System.out.println();
48
49            // print all young students
50            System.out.println("Young students");
51            System.out.println("-------------");
52            print(youngster);
53
54        }
55
56    }
```

Introduction to Java 8 Streams

- **Streams** are sequences of elements that support operations such as filtering and sorting in a declarative fashion akin to SQL
- Streams support both sequential and parallel operations
- You can retrieve a stream from any collection that implements java.util.Collection

---

Parallel operations make efficient use of multiple processors, transparently running operations in multiple threads.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Stream Example

- Many Stream methods return a stream so you can chain method calls

TestStream.java

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package streams;
2
3    import static java.util.Comparator.comparing;
4
5    import java.time.LocalDate;
6    import java.util.List;
7    import java.util.SortedSet;
8    import java.util.TreeSet;
9    import java.util.stream.Collectors;
10
11   import university.Student;
12
13   public class TestStream
14   {
15       private static SortedSet<Student> students = new TreeSet<Student>();
16
17       private static void init()
18       {
19           students.add(new Student(12, "Sam Smith", 3.45, LocalDate.of(1994, 4,
20                   12)));
21           students.add(new Student(4, "Sue Jones", 4.00, LocalDate
22                   .of(1992, 6, 22)));
23           students.add(new Student(74, "Harry Wolfe", 3.33, LocalDate.of(1993, 1,
24                   18)));
25           students.add(new Student(22, "Paul Westerberg", 2.56, LocalDate.of(
26                   1994, 8, 27)));
27       }
28
29       public static void main(String[] args)
30       {
31           init();
32
33           // Comparator<Student> sorter = (Student s1, Student s2) -> {
34           // int retVal = Double.compare(s1.getGpa(), s2.getGpa());
35           // return retVal;
36           // };
37
38           // Stream<Student> stream = students.stream();
39           // Stream<Student> honorRoll = stream.filter(s -> s.getGpa() >= 3.0);
40           // Stream<Student> honorRollSorted = honorRoll.sorted(sorter);
41           // Stream<Student> honorRollSorted =
42           // honorRoll.sorted(java.util.Comparator.comparing(Student::getGpa));
43           // List<Student> honorRollSortedList =
44           // honorRollSorted.collect(Collectors.toList());
45
```

```
46          List<Student> honorRollSortedList = students.stream()
47                  .filter(s -> s.getGpa() >= 3.0)
48                  .sorted(comparing(Student::getGpa))
49                  .collect(Collectors.toList());
50
51          for (Student s : honorRollSortedList)
52              System.out.println(s);
53      }
54
55  }
```

Java 8 and JavaScript

- Java 8 provides a fast, ECMAScript v5.1-compliant JavaScript engine known as **Nashorn**
- Java programs can thus evaluate JavaScripts in the JVM

test.js

TestNashorn.java

Java 7 provided a JavaScript engine based on Mozilla Rhino - Rhino hasn't been updated recently and had some performance issues.

The example on the next pages shows a Java program calling a JavaScript function - it's also possible for JavaScript code to call Java methods.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    function calc()
2    {
3        var array2 = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18];
4
5        var total = 0;
6
7        for(var i = 0; i < array2.length; i++)
8        {
9          total += array2[i];
10       }
11
12       return total;
13   }
```

```
1    package javascript_Nashorn;
2
3    import java.io.FileReader;
4
5    import javax.script.Invocable;
6    import javax.script.ScriptEngine;
7    import javax.script.ScriptEngineManager;
8
9    public class TestNashorn
10   {
11       public static void main(String[] args) throws Exception
12       {
13           ScriptEngine engine = new ScriptEngineManager()
14                   .getEngineByName("nashorn");
15
16           engine.eval(new FileReader("test.js"));
17
18           Invocable invocable = (Invocable) engine;
19
20           Object result = invocable.invokeFunction("calc");
21           System.out.println(result);
22       }
23
24   }
```

Chapter Summary

In this chapter, you learned about Java 8 features:

- Date/Time Classes
- Functional Programming
- Streams
- JavaScript Integration

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum