Lambda Expressions in Java 8: Part 3 – Lambda Building Blocks in java.util.function

Originals of slides and source code for examples: http://courses.coreservlets.com/Course-Materials/java.html
Also see Java 8 tutorial: http://www.coreservlets.com/java-8-tutorial/ and many other Java EE tutorials: http://www.coreservlets.com/
Customized Java training courses (onsite or at public venues): http://courses.coreservlets.com/java-training.html

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

λ

coreservlets.com – custom onsite training

For customized training related to Java or JavaScript, please email hall@coreservlets.com

Marty is also available for consulting and development support

The instructor is author of several popular Java EE books, two of the most popular Safari videos on Java and JavaScript, and this tutorial.

Courses available at public venues, or custom versions can be held on-site at your organization.

- **Courses developed and taught by Marty Hall**
  - JSF 2.3, PrimeFaces, Java programming (using Java 8, for those new to Java), Java 8 (for Java 7 programmers), JavaScript, jQuery, Angular 2, Ext JS, Spring Framework, Spring MVC, Android, GWT, custom mix of topics.
  - Java 9 training coming soon.
  - Courses available in any state or country.
  - Maryland/DC companies can also choose afternoon/evening courses.
- **Courses developed and taught by coreservlets.com experts (edited by Marty)**
  - Hadoop, Spark, Hibernate/JPA, HTML5, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Lambda building blocks in java.util.function**
  - Simply-typed versions
    - *Blah*UnaryOperator, *Blah*BinaryOperator, *Blah*Predicate, *Blah*Consumer
  - Generically-typed versions
    - Predicate
    - Function
    - BinaryOperator
    - Consumer
    - Supplier

Lambda Building Blocks in java.util.function

Main Points

- **java.util.function: many reusable interfaces**
  - Although they are technically interfaces with ordinary methods, they are treated as though they were functions
- **Simply typed interfaces**
  - IntPredicate, LongUnaryOperator, DoubleBinaryOperator, etc.
- **Generically typed interfaces**
  - Predicate<T> — T in, boolean out
  - Function<T,R> — T in, R out
  - Consumer<T> — T in, nothing (void) out
  - Supplier<T> — Nothing in, T out
  - BinaryOperator<T> — Two T's in, T out

Simply Typed Building Blocks

Main Points

- **Interfaces like Integrable widely used**
  - So, Java 8 should build in many common cases
- **Can be used in wide variety of contexts**
  - So need more general name than "Integrable"
- **java.util.function defines many simple functional (SAM) interfaces**
  - Named according to arguments and return values
    - E.g., replace my Integrable with builtin DoubleUnaryOperator
  - You need to look in API for the method names
    - Although the lambdas themselves don't refer to method names, your code that *uses* the lambdas will need to call the methods explicitly

Simply-Typed and Generic Interfaces

- **Types given**
  - Samples (*many* others!)
    - IntPredicate (int in, boolean out)
    - LongUnaryOperator (long in, long out)
    - DoubleBinaryOperator(two doubles in, double out)
  - Example

```
DoubleBinaryOperator f = (d1, d2) -> Math.cos(d1 + d2);
```

- **Genericized**
  - There are also generic interfaces (Function<T,R>, Predicate<T>, etc.) with widespread applicability
    - And concrete methods like "compose" and "negate"

Interface from Previous Lecture

```
@FunctionalInterface
public interface Integrable {
  double eval(double x);
}
```


Numerical Integration Method

```
public static double integrate(Integrable function,
                               double x1, double x2,
                               int numSlices){
  if (numSlices < 1) {
    numSlices = 1;
  }
  double delta = (x2 - x1)/numSlices;
  double start = x1 + delta/2;
  double sum = 0;
  for(int i=0; i<numSlices; i++) {
    sum += delta * function.eval(start + delta * i);
  }
  return(sum);
}
```

Method for Testing

```
public static void integrationTest(Integrable function,
                                   double x1, double x2) {
  for(int i=1; i<7; i++) {
    int numSlices = (int)Math.pow(10, i);
    double result =
      MathUtilities.integrate(function, x1, x2, numSlices);
    System.out.printf("  For numSlices =%,10d result = %,.8f%n",
                      numSlices, result);
  }
}
```


Using Numerical Integration

```
MathUtilities.integrationTest(x -> x*x, 10, 100);
MathUtilities.integrationTest(x -> Math.pow(x,3), 50, 500);
MathUtilities.integrationTest(Math::sin, 0, Math.PI);
MathUtilities.integrationTest(Math::exp, 2, 20);
```

Using Builtin Building Blocks

- **In integration example, replace this**

```
public static double integrate(Integrable function, …) {
  ... function.eval(...);  ...
}
```

- **With this**

```
public static double integrate(DoubleUnaryOperator function, …) {
  ... function.applyAsDouble(...); ...
}
```

- **Then, omit definition of Integrable entirely**
  – Because DoubleUnaryOperator is a functional (SAM) interface containing a method with the same signature as the method of the Integrable interface

General Case

- **If you are tempted to create an interface purely to be used as a target for a lambda**
  – Look through java.util.function and see if one of the functional (SAM) interfaces there can be used instead
    - DoubleUnaryOperator, IntUnaryOperator, LongUnaryOperator
      – double/int/long in, same type out
    - DoubleBinaryOperator, IntBinaryOperator, LongBinaryOperator
      – Two doubles/ints/longs in, same type out
    - DoublePredicate, IntPredicate, LongPredicate
      – double/int/long in, boolean out
    - DoubleConsumer, IntConsumer, LongConsumer
      – double/int/long in, void return type
    - Genericized interfaces: Function, Predicate, Consumer, etc.
      – Covered in next section

Generic Building Blocks: Predicate

Predicate: Main Points

- **Simplified definition**

```
public interface Predicate<T> {
  boolean test(T t);
}
```

Simplified because Predicate has some non-abstract methods (covered later), and it (of course!) uses the @FunctionalInterface annotation.

- **Idea**
  – Lets you make a "function" to test a condition
- **Benefit**
  – Lets you search collections for entry or entries that match a condition, with much less repeated code than without lambdas
- **Syntax example**

```
Predicate<Employee> matcher = e -> e.getSalary() > 50_000;
if(matcher.test(someEmployee)) {
  doSomethingWith(someEmployee);
}
```

Example: Finding Entries in List that Match Some Test

- **Idea**
  - Very common to have a list, then take a subset of the list by throwing away entries that fail a test
- **Java 7**
  - You tended to repeat the code for different types of tests
- **Java 8 first cut**
  - Use Predicate<TypeInOurList> to generalize the test
- **Java 8 second cut**
  - Use Predicate<T> to generalize to different types of lists
- **Java 8 third cut (later lecture)**
  - Use the builtin filter method of Stream to get the benefits of chaining, lazy evaluation, and parallelization

Without Predicate: Finding Employee by First Name

```
public static Employee findEmployeeByFirstName(List<Employee> employees,
                                               String firstName) {
  for(Employee e: employees) {
    if(e.getFirstName().equals(firstName)) {
      return(e);
    }
  }
  return(null);
}
```

Without Predicate: Finding Employee by Salary

```
public static Employee findEmployeeBySalary(List<Employee> employees,
                                            double salaryCutoff) {
  for(Employee e: employees) {
    if(e.getSalary() >= salaryCutoff) {
      return(e);
    }
  }
  return(null);
}
```

Most of the code from the previous example is repeated. If we searched by last name or employee ID, we would yet again repeat most of the code.

Refactor #1: Finding First Employee that Passes Test

```
public static Employee firstMatchingEmployee(List<Employee> candidates,
                                      Predicate<Employee> matchFunction) {
  for(Employee possibleMatch: candidates) {
    if(matchFunction.test(possibleMatch)) {
      return(possibleMatch);
    }
  }
  return(null);
}
```

Refactor #1: Benefits

- **Now**
  - We can now pass in different match functions to search on different criteria. Succinct and readable.
    - `firstMatchingEmployee(employees, e -> e.getSalary() > 500_000);`
    - `firstMatchingEmployee(employees, e -> e.getLastName().equals("…"));`
    - `firstMatchingEmployee(employees, e -> e.getId() < 10);`
- **Before**
  - Cumbersome interface.
    - Without lambdas, we could have defined an interface with a "test" method, then instantiated the interface and passed it in, to avoid some of the previously repeated code. But, this approach would be so verbose that it wouldn't seem worth it in most cases. The method calls above, in contrast, are succinct and readable.
- **Doing even better**
  - The code is still tied to the Employee class, so we can do even better (next slide).

Refactor #2: Finding First Entry that Passes Test

```
public static <T> T firstMatch(List<T> candidates,
                               Predicate<T> matchFunction) {
  for(T possibleMatch: candidates) {
    if(matchFunction.test(possibleMatch)) {
      return(possibleMatch);
    }
  }
  return(null);
}
```

We can now pass in different match functions to search on different criteria as before, but can do so for any type, not just for Employees.

Using firstMatch

- **firstMatchingEmployee examples still work**
  - `firstMatch(employees, e -> e.getSalary() > 500_000);`
  - `firstMatch(employees, e -> e.getLastName().equals("…"));`
  - `firstMatch(employees, e -> e.getId() < 10);`
- **But more general code now also works**
  - ```
    Country firstBigCountry =
      firstMatch(countries, c -> c.getPopulation() > 10_000_000);
    ```
  - ```
    Car firstCheapCar =
      firstMatch(cars, c -> c.getPrice() < 15_000);
    ```
  - ```
    Company firstSmallCompany =
      firstMatch(companies, c -> c.numEmployees() <= 50);
    ```
  - ```
    String firstShortString =
      firstMatch(strings, s -> s.length() < 4);
    ```


Testing Lookup by First Name

```
private static final List<Employee> EMPLOYEES = EmployeeSamples.getSampleEmployees();
private static final String[] FIRST_NAMES = { "Archie", "Amy", "Andy" };

@Test
public void testNames() {
  assertThat(findEmployeeByFirstName(EMPLOYEES, FIRST_NAMES[0]),
             is(notNullValue()));
  for(String firstName: FIRST_NAMES) {
    Employee match1 =
      findEmployeeByFirstName(EMPLOYEES, firstName);
    Employee match2 =
      firstMatchingEmployee(EMPLOYEES, e -> e.getFirstName().equals(firstName));
    Employee match3 =
      firstMatch(EMPLOYEES, e -> e.getFirstName().equals(firstName));
    assertThat(match1, allOf(equalTo(match2), equalTo(match3)));
  }
}
```

Testing goals:
- The hardcoded version gives same answer as the version with the Predicate<Employee>, but not merely by both always returning null.
- The version with generic types gives same answer and has identical syntax (except for method name) as the version with Predicate<Employee>.

Reminder: JUnit covered in earlier section.

Testing Lookup by Salary

```
private static final List<Employee> EMPLOYEES = EmployeeSamples.getSampleEmployees();
private static final int[] SALARY_CUTOFFS = { 200_000, 300_000, 400_000 };

@Test
public void testSalaries() {
  assertThat(findEmployeeBySalary(EMPLOYEES, SALARY_CUTOFFS[0]),
             is(notNullValue()));
  for(int cutoff: SALARY_CUTOFFS) {
    Employee match1 =
      findEmployeeBySalary(EMPLOYEES, cutoff);
    Employee match2 =
      firstMatchingEmployee(EMPLOYEES, e -> e.getSalary() >= cutoff);
    Employee match3 =
      firstMatch(EMPLOYEES, e -> e.getSalary() >= cutoff);
    assertThat(match1, allOf(equalTo(match2), equalTo(match3)));
  }
}
```


Definition of Predicate Revisited

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
}
```

Except for @FunctionalInterface, this is the same way you could have written Predicate in Java 7. But, it wouldn't have been very useful in Java 7 because the code that supplied the Predicate would have to use a clumsy and verbose inner class instead of a lambda.

And, I am oversimplifying this definition, because Predicate has some default and static methods. But, they wouldn't be needed for the use of Predicate on previous slides.

General Lambda Principles Revisited

- **Interfaces in Java 8 are same as in Java 7**
  - Predicate is same in Java 8 as it would have been in Java 7, except you can (and should!) optionally use @FunctionalInterface
    - To catch errors (multiple methods) at compile time
    - To express design intent (developers should use lambdas)
- **Code that uses interfaces is the same in Java 8 as in Java 7**
  - I.e., the definition of firstMatch is exactly the same as you would have written it in Java 7. The author of firstMatch must know that the real method name is test.
- **Code that calls methods that expect 1-method interfaces can now use lambdas**
  - `firstMatch(employees, e -> e.getSalary() > 500_000);`

Generic Building Blocks: Function

Function: Main Points

- **Simplified definition**

```
public interface Function<T,R> {
  R apply(T t);
}
```

- **Idea**
  - Lets you make a “function” that takes in a T and returns an R
    - BiFunction is similar, but “apply” takes two arguments
- **Benefit**
  - Lets you transform a value or collection of values, with much less repeated code than without lambdas
- **Syntax example**

```
Function<Employee, Double> raise = e -> e.getSalary() * 1.1;
for(Employee employee: employees) {
  employee.setSalary(raise.apply(employee));
}
```


Example 1: Refactoring our String-Transformation Code

- **Previous lecture**
  - We made StringFunction interface and transform method to demonstrate different types of method references.
- **Refactor 1**
  - Replace StringFunction with Function<String,String>
  - But we also have to change the transform method. General lambda principle: code that uses the interfaces is the same as in Java 7, and must know the real method name.
- **Refactor 2**
  - Use Function<T,R> instead of Function<String,String>
  - Generalize transform to take in a T and return an R

Previous Section: Transforming with StringFunction

- **Our interface**

```
@FunctionalInterface
public interface StringFunction {
  String applyFunction(String s);
}
```

- **Our method**

```
public static String transform(String s, StringFunction f) {
  return(f.applyFunction(s));
}
```

- **Sample usage**

```
String result = Utils.transform(someString, String::toUpperCase);
```


Refactor 1: Use Function

- **Our interface**
  – None!
- **Our method**

```
public static String transform(String s, Function<String,String> f) {
  return(f.apply(s));
}
```

- **Sample use (unchanged)**

```
String result = Utils.transform(someString, String::toUpperCase);
```

Refactor 2: Generalize the Types

- **Our interface**
  – None
- **Our method**

```
public static <T,R> R transform(T value, Function<T,R> f) {
  return(f.apply(value));
}
```

- **Sample usage (more general)**

```
String result = Utils.transform(someString, String::toUpperCase);
List<String> words = Arrays.asList("hi", "bye");
int size = Utils.transform(words, List::size);
```


Example 2: Finding Sum of Arbitrary Property

- **Idea**
  – Very common to take a list of employees and add up their salaries
  – Also common to take a list of countries and add up their populations
  – Also common to take a list of cars and add up their prices
- **Java 7**
  – You tended to repeat the code for each of those cases
- **Java 8**
  – Use Function to generalize the transformation operation (salary, population, price)

Without Function: Finding Sum of Employee Salaries

```
public static int salarySum(List<Employee> employees) {
  int sum = 0;
  for(Employee employee: employees) {
    sum += employee.getSalary();
  }
  return(sum);
}
```


Without Function: Finding Sum of Country Populations

```
public static int populationSum(List<Country> countries) {
  int sum = 0;
  for(Country country: countries) {
    sum += country.getPopulation();
  }
  return(sum);
}
```

With Function: Finding Sum of Arbitrary Property

```
public static <T> int mapSum(List<T> entries,
                             Function<T, Integer> mapper) {
  int sum = 0;
  for(T entry: entries) {
    sum += mapper.apply(entry);
  }
  return(sum);
}
```


Results

- You can reproduce the results of salarySum
  - `int numEmployees = mapSum(employees, Employee::getSalary);`
- You can also do many other types of sums:
  - `int totalWeight = mapSum(packages, Package::getWeight);`
  - `int totalFleetPrice =  mapSum(cars, Car::getStickerPrice);`
  - `int regionPopulation = mapSum(countries, Country::getPopulation);`
  - ```
    int regionElderlyPopulation =
      mapSum(listOfCountries,
             c -> c.getPopulation() – c.getPopulationUnderSixty());
    ```
  - `int sumOfNumbers =  mapSum(listOfIntegers, Function.identity());`
    ```

Other Generic Building Blocks

BinaryOperator: Main Points

- **Simplified definition**

```
public interface BinaryOperator<T> {
  T apply(T t1, T t2);
}
```

- **Idea**
  - Lets you make a "function" that takes in two T's and returns a T
    - This is a specialization of BiFunction<T,U,R> where T, U, and R are all the same type.
- **Benefit**
  - See Function. Having all the values be same type makes it particularly useful for "reduce" operations that combine values from a collection.
- **Syntax example**

```
BinaryOperator<Integer> adder = (n1, n2) -> n1 + n2;
      // The lambda above could be replaced by Integer::sum
int sum = adder.apply(num1, num2);
```

BinaryOperator: Applications

- **Make mapSum more flexible**
  - Instead of
    - `mapSum(List<T> entries, Function<T, Integer> mapper)`
  - you could generalize further and pass in combining operator (which was hardcoded to "+" in mapSum)
    - ```
      mapReduce(List<T> entries, Function<T, R> mapper,
                BinaryOperator<R> combiner)
      ```
- **Hypothetical examples**
  - ```
    int payroll =
      mapReduce(employees, Employee::getSalary, Integer::sum);
    ```
  - `double lowestPrice = mapReduce(cars, Car::getPrice, Math::min);`
- **Problem:**
  - What do you do if there are no entries? mapSum would return 0, but what would mapReduce return? We will deal with this exact issue when we cover the reduce method of Stream, which uses BinaryOperator in just this manner.

Consumer: Main Points

- **Simplified definition**

```
public interface Consumer<T> {
  void accept(T t);
}
```

- **Idea**
  - Lets you make a "function" that takes in a T and does some side effect to it (with no return value)
- **Benefit**
  - Lets you do an operation (print each value, set a raise, etc.) on a collection of values, with much less repeated code than without lambdas
- **Syntax example**

```
Consumer<Employee> raise = e -> e.setSalary(e.getSalary() * 1.1);
for(Employee employee: employees) {
  raise.accept(employee);
}
```

Consumer: Application

- **The builtin forEach method of Stream uses Consumer**
  - `employees.forEach(e -> e.setSalary(e.getSalary()*1.1));`
  - `values.forEach(System.out::println);`
  - `textFields.forEach(field -> field.setText(""));`
- **More details**
  - See later lecture on Streams

Supplier: Main Points

- **Simplified definition**

```
public interface Supplier<T> {
  T get();
}
```

- **Idea**
  - Lets you make a no-arg "function" that returns a T. It can do so by calling "new", using an existing object, or anything else it wants.
- **Benefit**
  - Lets you swap object-creation functions in and out. Especially useful for switching among testing, production, etc.
- **Syntax example**

```
Supplier<Employee> maker1 = Employee::new;
Supplier<Employee> maker2 = () -> randomEmployee();
Employee e1 = maker1.get();
Employee e2 = maker2.get();
```

Using Supplier to Randomly Make Different Types of Person

```
private final static Supplier[] peopleGenerators =
  { Person::new, Writer::new, Artist::new, Consultant::new,
    EmployeeSamples::randomEmployee,
    () -> { Writer w = new Writer();
            w.setFirstName("Ernest");
            w.setLastName("Hemingway");
            w.setBookType(Writer.BookType.FICTION);
            return(w); }
  };

public static Person randomPerson() {
  Supplier<Person> generator =
    RandomUtils.randomElement(peopleGenerators);
  return(generator.get());
}
```

When randomPerson is called, it first randomly chooses one of the people generators, then uses that Supplier to build an instance of a Person or subclass of Person.

Helper Method: randomElement

```
public class RandomUtils {
  private static Random r = new Random();

  public static int randomInt(int range) {
    return(r.nextInt(range));
  }

  public static int randomIndex(Object[] array) {
    return(randomInt(array.length));
  }

  public static <T> T randomElement(T[] array) {
    return(array[randomIndex(array)]);
  }
}
```

Using randomPerson

- **Test code**

```
System.out.printf("%nSupplier Examples%n");
for(int i=0; i<10; i++) {
  System.out.printf("Random person: %s.%n", EmployeeUtils.randomPerson());
}
```

- **Results (one of many possible outcomes)**

```
Supplier Examples
Random person: Andrea Carson (Consultant).
Random person: Desiree Designer [Employee#14 $212,000].
Random person: Andrea Evans (Artist).
Random person: Devon Developer [Employee#11 $175,000].
Random person: Tammy Tester [Employee#19 $166,777].
Random person: David Carson (Writer).
Random person: Andrea Anderson (Person).
Random person: Andrea Bradley (Writer).
Random person: Frank Evans (Artist).
Random person: Erin Anderson (Writer).
```


Wrap-Up

Summary

- **Type-specific building blocks**
  - *Blah*UnaryOperator, *Blah*BinaryOperator, *Blah*Predicate, *Blah*Consumer
- **Generic building blocks**
  - Predicate

```
Predicate<Employee> matcher = e -> e.getSalary() > 50000;
if(matchFunction.test(someEmployee)) { doSomethingWith(someEmployee); }
```

  - Function

```
Function<Employee, Double> raise = e -> e.getSalary() + 1000;
for(Employee employee: employees) { employee.setSalary(raise.apply(employee)); }
```

  - BinaryOperator

```
BinaryOperator<Integer> adder = (n1, n2) -> n1 + n2;
int sum = adder.apply(num1, num2);
```

  - Consumer

```
Consumer<Employee> raise = e -> e.setSalary(e.getSalary() * 1.1);
for(Employee employee: employees) { raise.accept(employee); }
```


Questions?