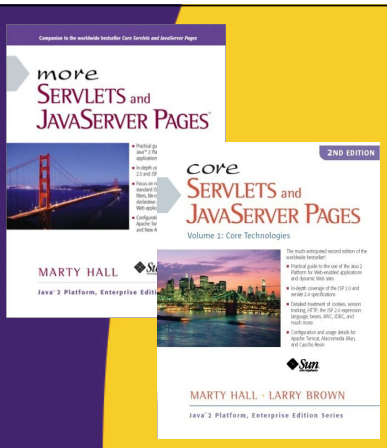
Lambda Expressions in Java 8: Part 2 – Lambda Building Blocks in java.util.function

Originals of slides and source code for examples: http://www.coreservlets.com/java-8-tutorial/
Also see the general Java programming tutorial – http://courses.coreservlets.com/Course-Materials/java.html
and customized Java training courses (onsite or at public venues) – http://courses.coreservlets.com/java-training.html

© 2015 Marty Hall



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 lambdas don't refer to method names, your code that *uses* the lambdas will need to call the methods

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>) 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);
}
```

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 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

- **boolean test(T t)**
  - 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() > 50000;
if(matcher.test(someEmployee)) {
  doSomethingWith(someEmployee);
}
```

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() > 500000);
    - 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() > 500000);
  - 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() > 1000000);
  - Car firstCheapCar = firstMatch(cars, c -> c.getPrice() < 15000);
  - Company firstSmallCompany = firstMatch(companies, c -> c.numEmployees() <= 50);
  - String firstShortString = firstMatch(strings, s -> s.length() < 4);

Definition of Predicate

```
@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 passed in the Predicate would have to use a clumsy and verbose inner class instead of a lambda.

And, I am over simplifying 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 Principle 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 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 was 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() > 50000);

Generic Building Blocks: Function

Function: Main Points

- **R apply(T t)**
  - 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() + 1000;
for(Employee employee: employees) {
  employee.setSalary(raise.apply(employee));
}
```

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);
}
```


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);
}
```

You can reproduce salarySum via mapSum(employees, Employee::getSalary).
But, now you can also do many other types of sums:

- int sumOfIds = mapSum(employees, Employee::getEmployeeId);
- int sumOfSalariesAndIds = mapSum(employees, e -> e.getSalary() + e.getEmployeeId());
- int totalFleetPrice = mapSum(listOfCars, Car::getStickerPrice);
- int regionPopulation = mapSum(listOfCountries, Country::getPopulation);
- int regionElderlyPopulation = mapSum(listOfCountries, c -> c.getPopulation() – c.getPopulationUnderSixty());
- int sumOfNumbers = mapSum(listOfIntegers, Function.identity());

Other Generic Building Blocks

BinaryOperator: Main Points

- **T apply(T t1, T t2)**
  - 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 righthand side above could also be 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 addition in mapSum)
    - mapCombined(List<T> entries,
      Function<T, R> mapper,
      BinaryOperator<R> combiner)
- **This is more or less what the builtin "reduce" method does**
  - See second lecture on Streams

Consumer: Main Points

- **void accept(T t)**
  - 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()*11/10))
  - values.forEach(System.out::println)
  - textFields.forEach(field -> field.setText(""))
- **More details**
  - See second lecture on Streams

Supplier: Main Points

- **T get()**
  - 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 Subclasses 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?

More info:

http://courses.coreservlets.com/Course-Materials/java.html – General Java programming tutorial

http://www.coreservlets.com/java-8-tutorial/ – Java 8 tutorial

http://courses.coreservlets.com/java-training.html – Customized Java training courses, at public venues or onsite at *your* organization

http://coreservlets.com/ – JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training

Many additional free tutorials at coreservlets.com (JSF, Android, Ajax, Hadoop, and lots more)

Customized Java EE Training: http://courses.coreservlets.com/

Java 7, Java 8, JSF 2, PrimeFaces, Android, JSP, Ajax, jQuery, Spring MVC, RESTful Web Services, GWT, Hadoop

Developed and taught by well-known author and developer. At public venues or onsite at *your* location.