CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Lambda Expressions – III

Lambda Building Blocks in java.util.function

- 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

- Interfaces like Integrable widely used
  - So, Java 8 should have built in interfaces for common cases
- Can be used in wide variety of contexts
  - So need more general name than "Integrable"
- java.util.function defines many simple functional interfaces
  - Named according to arguments and return values
    - E.g., replace 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

- Sample of Types
  - IntPredicate (int in, boolean out)
  - LongUnaryOperator (long in, long out)
  - DoubleBinaryOperator(two doubles in, double out)
  - Many others..
- 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”

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

- Before creating 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.

Generic Building Blocks: Predicate

- 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

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

Predicate has some non-abstract methods

Syntax example

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

Finding Entries in List that Match Some Test

- 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

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

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

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);
}
```

- Benefit : 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);
```

- Can we do better?

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 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)));
  }
}
```

Generic Building Blocks: Function

- **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

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

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

Remember: 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

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

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: 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

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

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());`

BinaryOperator

- **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.

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

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

Applications of BinaryOperator

- **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

- **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

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

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

Consumer Applications

- 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(""));
```

- See later lecture on Streams

Supplier

- **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.

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

```
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).
```

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); }
```