CENG 443
Introduction to Object-Oriented Programming Languages and Systems

Lambda Expressions – I

Moving to Java 8

- In Java 8, functional interfaces, lambdas, and method references were added to make it easier to create function objects. The streams API was added in tandem with these language changes to provide library support for processing sequences of data elements.
- **Functional approach proven to be concise, flexible, and parallelizable**
- Java could not resist Lambdas
  - Concise syntax
    - More succinct and clear than anonymous inner classes
  - Deficiencies with anonymous inner classes
    - Bulky, confusion re “this” and naming in general, no access to non-final local vars, hard to optimize
  - Convenient for new streams library
    - shapes.forEach(s -> s.setColor(Color.RED));
- Programmers are used to the approach
    - Callbacks, closures, map/reduce idiom

Advantage of Lambdas: Concise and Expressive

- Old

```
button.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    doSomethingWith(e);
  }
});
```

- New

```
button.addActionListener(e -> doSomethingWith(e));
```

Support New Way of Thinking

- When functional programming approach is used, many classes of problems are easier to solve and result in code that is clearer to read and simpler to maintain
- Functional programming does not replace object-oriented programming in Java 8. OOP is still the main approach for representing types. But functional programming augments and improves many methods and algorithms.
- Streams are wrappers around data sources (arrays, collections, etc.) that use lambdas, support map/filter/reduce, use lazy evaluation, and can be made parallel automatically
  - **employees.stream().parallel().forEach(e -> e.giveRaise(1.15));**

Lambdas: Syntax

- You write what looks like a function
  - `Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());`
  - `taskList.execute(() -> downloadSomeFile());`
  - `someButton.addActionListener(event -> handleButtonClick());`
  - `double d = MathUtils.integrate(x -> x*x, 0, 100, 1000);`
- You get an instance of a class that implements the interface that was expected in that place
  - The expected type must be an interface that has *exactly one* (abstract) method
  - Called “Functional Interface” or “Single Abstract Method (SAM) Interface”
  - The designation of a single ABSTRACT method is not redundant, because in Java 8 interfaces can have concrete methods, called “default methods”.
  - Java 8 interfaces can also have static methods.

Example: Sorting Strings by Length

- Arrays.sort(T[] a, Comparator<? super T> c)

  Sorts the specified array of objects according to the order induced by the specified comparator.

- Java 7

```
Arrays.sort(testStrings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return(s1.length() - s2.length());
  }
});
```

- Java 8

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

How do we get there?

Step 1 – Drop Interface and Method names

- From the API, Java already knows the second argument to Arrays.sort is a Comparator, so you do not need to say it is a Comparator.
- Comparator has only one method, so you do not need to say that method name is compare.
- Add "->" (i.e., dash then greater-than sign) between method params and method body

```
Arrays.sort(testStrings, ~~new Comparator<String>() {~~
  ~~@Override~~
  ~~public int compare~~(String s1, String s2) {
    return(s1.length() - s2.length());
  ~~}~~
~~}~~);
```

```
Arrays.sort(testStrings,
  (String s1, String s2) -> { return(s1.length() – s2.length()); });
```

Legal but not ideal!

Step 2 – Drop Parameter Type Declarations

- By looking at the first argument (testStrings), Java can infer that the type of the second argument is Comparator<String>. Thus, parameters for compare are both Strings. Since Java knows this, you do not need to say so.
- Java is still doing strong, compile-time type checking. The compiler is just inferring types. Somewhat similar to how Java infers types for the diamond operator.
  - List<String> words = new ArrayList<>();
- In a few cases, types are ambiguous, and compiler will warn you that it cannot infer the types. In that case, you can*not* drop the types declarations.
- But mostly, you can do:

```
Arrays.sort(testStrings,
        (~~String~~ s1, ~~String~~ s2) -> { return(s1.length() - s2.length()); });


Arrays.sort(testStrings,
        (s1, s2) -> { return(s1.length() - s2.length()); });
```

Legal but not ideal!

Step 3 – Use Expression Instead of Block

- If method body can be written as a single return statement, you can drop the curly braces and “return”, and just put the return value as an expression.
- This cannot always be done, especially if you use loops or if statements. However, lambdas are most commonly used when the method body is short, so this can usually be done. If not, leaving curly braces and “return” is legal, but if body of lambda is long, you might want to reconsider and use normal inner class instead.

```
Arrays.sort(testStrings,
  (s1, s2) -> ~~{ return (~~s1.length() - s2.length()~~); }~~);
```

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

IDEAL!

Optional Step 4 – Omit Parens When there is Exactly One Param

- If the method of the interface has *exactly* one parameter, the parens are optional
- Java 7

```
button.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    doSomethingWith(e);
  }
});
```

- Java 8

```
button.addActionListener((e) -> doSomethingWith(e));

button.addActionListener(e -> doSomethingWith(e));
```

Summary: Lambda Syntax

Omit interface and method names

```
Arrays.sort(testStrings, new Comparator<String>() {
   @Override public int compare(String s1, String s2) { return(s1.length() - s2.length()); }
});
```

replaced by

```
Arrays.sort(testStrings, (String s1, String s2) -> { return(s1.length() – s2.length()); });
```

Omit parameter types

```
Arrays.sort(testStrings, (String s1, String s2) -> { return(s1.length() – s2.length()); });
```

replaced by

```
Arrays.sort(testStrings, (s1, s2) -> { return(s1.length() – s2.length()); });
```

Use expressions instead of blocks

```
Arrays.sort(testStrings, (s1, s2) -> { return(s1.length() – s2.length()); });
```

replaced by

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() – s2.length());
```

Drop parens if single param to method

```
button1.addActionListener((event) -> popUpSomeWindow(...));
```

replaced by

```
button1.addActionListener(event -> popUpSomeWindow(...));
```

Example

- Java 7

```
    taskList.execute(new Runnable() {
        @Override
        public void run() {
            processSomeImage(imageName);
        }
    });
    button.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent event) {
            doSomething(event);
        }
    });
```

- Java 8

```
    taskList.execute(() -> processSomeImage(imageName));
    button.addActionListener(event -> doSomething(event));
```

Under the hood

```
Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());
```

- **What really is happening**
  - You used a shortcut way of representing an instance of a class that implements Comparator<T>
  - You provided the body of the compare method after the "->",
- **How you usually think about it**
  - You passed in the comparison function
- **Function types**
  - Java 8 does *not* technically have function types, since under the hood, lambdas become instances of classes that implement whatever interface was expected.
  - Nevertheless, you normally think of lambdas as functions!

Where Can Lambdas Be Used?

- **Find any variable or parameter that expects an interface that has one method**
  - Technically 1 abstract method, but in Java 7 there was no distinction between a 1-method interface and a 1-abstract-method interface.
  - These 1-method interfaces are called “functional interfaces” or “SAM (Single Abstract Method) interfaces”.
  - public interface Blah { String foo(String someString); }
- **Code that uses interface is the same**
  - public void someMethod(**Blah** b) { … b.**foo**(…) …}
  - Code that uses the interface must still know the real method name in the interface
- **Code that calls the method that expects the interface can supply lambda**
  - someMethod(**s -> s.toUpperCase() + "!"**);

Example: Numerical Integration

- Simple numerical integration using rectangle (mid-point) rule

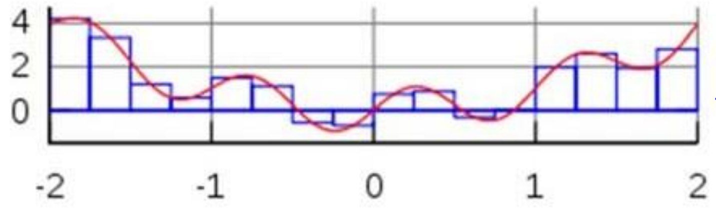

Diagram from
http://en.wikipedia.org/wiki/Numerical_integration

- Want to use lambdas to make it convenient and succinct to supply the function that will be integrated
  - Need to define a functional (SAM) interface with a "double eval(double x)" method to specify function to be integrated

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

Numerical Integration Method

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


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);
  }
}
```

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

Lambda Principles

- **Interfaces in Java 8 are the same as in Java7**
  - Integrable was the same as it would be in Java 7, except that you can (should) optionally use @FunctionalInterface
- **Code that *uses* interfaces is the same in Java 8 as in Java 7**
  - E.g., the definition of integrate is exactly the same as you would have written it in Java 7. The author of integrate must know that the real method name is eval.
- **Code that *calls* methods that expect 1-method interfaces can now use lambdas**

```
MathUtilities.integrate(x -> Math.sin(x), 0, Math.PI, …);
```

Instead of new Integrable() { public void eval(double x) { return(Math.sin(x)); } }

Example: Timing Utility

- Goal
  - Pass in a "function"
  - Run the function
  - Print elapsed time
- Problem: Java evaluates args on the call
  - TimingUtils.timeOp(someSlowCalculation());
  - The calculation is computed before timeOp is called!
- Solution: use lambdas
  - TimingUtils.timeOp(() -> someSlowCalculation());
  - timeOp can run the calculation internally
- Could be done with inner classes
  - But, code that calls timeOp is long and cumbersome

The Op Interface

```
@FunctionalInterface
public interface Op {  void
  runOp();
}
```

- **Catches errors at compile time**
  - If developer later adds a second abstract method, interface will not compile
- **Expresses design intent**
  - Tells fellow developers that this is interface that you expect lambdas to be used for
- **But, like @Override not technically required**
  - You can use lambdas *anywhere* 1-abstract-method interfaces (aka functional interfaces, SAM interfaces) are expected, whether or not that interface used @FunctionalInterface

TimingUtils Class

```
public class TimingUtils {
  private static final double ONE_BILLION = 1_000_000_000;

  public static void timeOp(Op operation) {
    long startTime = System.nanoTime();
    operation.runOp();
    long endTime = System.nanoTime();
    double elapsedSeconds = (endTime - startTime)/ONE_BILLION;
    System.out.printf("  Elapsed time: %.3f seconds.%n",
    elapsedSeconds);
  }
}
```

Testing Code

```
public class TimingTests {
  public static void main(String[] args) {
    for(int i=3; i<8; i++) {
      int size = (int)Math.pow(10, i);
      System.out.printf("Sorting array of length %,d.%n", size);
      TimingUtils.timeOp(() -> sortArray(size));
    }
  }

  public static double[] randomNums(int length) {
    double[] nums = new double[length];
    for(int i=0; i<length; i++) {
      nums[i] = Math.random();
    }
    return(nums);
  }

  public static void sortArray(int length) {
    double[] nums = randomNums(length);
    Arrays.sort(nums);
  }
```

Summary

- Yay! We have lambdas
  - Concise and brief
  - Retrofits into existing APIs
  - Familiar to developers that know functional programming
  - Fits well with new streams API
- Boo! We do not have full functional programming (?)
  - Type of a lambda is class that implements interface, not a "real" function
  - Must create or find interface first, must know method name
  - Cannot use mutable local variables