PIC 20A
Anonymous classes, Lambda Expressions, and Functional Programming

David Hyde
UCLA Mathematics

Last edited: June 1, 2020

Introductory example

When you write an `ActionListener` for a GUI, you write a `class` and only use it once.

```
class MyGUI {
  private JTextArea text;
  private final JButton button1, button2;

  private class B1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("button1 clicked");
    }
  }
  private class B2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println(text.getText());
    }
  }
  ...
```

Example: anonymous classes as EventListeners

```
button1.addActionListener(new B1L());
button2.addActionListener(new B2L());
```

This is wasteful. Why do you need to name something you only use once?

This is poor style. The code above doesn't tell you what `B1L` and `B2L` do. You need to find their implementations somewhere else to figure that out.

Outline

Anonymous classes

When you use a `class` only once, don't name it. Make it anonymous.

Writing an anonymous `class` is like writing the `class` definition and then immediately calling the constructor (only once).

Let's see how to do this through examples.

Example: anonymous classes as EventListeners

This code uses an inner `class`.

```
button1.addActionListener(new B1L());
```

Example: anonymous classes as EventListeners

This code still uses an inner class.

```
ActionListener AL1 = new B1L();
button1.addActionListener(AL1);
```

Example: anonymous classes as EventListeners

Instead use an anonymous class.

```
ActionListener AL1 = new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    System.out.println("button1 clicked");
  }
};
button1.addActionListener(AL1);
```

The anonymous `class` has one instance, which is referred to by the reference `AL1`.

Example: anonymous classes as EventListeners

We can also get rid of the reference.

```
button1.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    System.out.println("button1 clicked");
  }
});
```

Example: anonymous classes as EventListeners

For multiple `ActionListeners`, write multiple anonymous classes.

```
button1.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println("button1 clicked");
  }
});
button2.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println(text.getText());
  }
});
```

(I'll omit `@Override` for brevity.)

Example: anonymous classes as MouseListener

Anonymous classes can have multiple methods.

```
text.addMouseListener( new MouseListener() {
  public void mouseEntered(MouseEvent e) {
    System.out.println("Mouse entered");
  }
  public void mouseClicked(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
});
```

Outline

Lambda expression

Anonymous `classes` are useful, but the syntax of anonymous `classes` can be unwieldy and unclear.

In Java 8, lamabda expressions were introduced. A lambda expression lets you concisely write an anonymous `class` that implements a functional `interface`.

Let's see how to do this through examples.

Functional interface

- Legally speaking, an `interface` is a functional `interface` if it has exactly one `abstract` method.
- Conceptually speaking, an `interface` is a functional `interface` if its purpose is to provide a function.
- `Comparator<E>` is a functional `interface`, legally and conceptually.
- `Comparable<E>`, `Iterable<E>`, or `Iterator<E>` are not functional `interfaces`, conceptually.
- Functional `interfaces` often have the annotation `@FunctionalInterface`.
- `java.util.function` has may useful functional `interfaces`.

Example: lambda expressions for reduce

Java uses functional `interfaces` instead of function pointers.

```
@FunctionalInterface
public interface BinOper {
  int op(int a, int b);
}
```

```
public static int reduce(int[] arr, BinOper b) {
  int ret = b.op(arr[0], arr[1]);
  for (int i=2; i<arr.length; i++)
    ret = b.op(ret, arr[i]);
  return ret;
}
```

Example: lambda expressions for reduce

We can implement a functional `interface` with an anonymous `class`.

```
int[] i_arr = ...;

int max = reduce(i_arr, new BinOper() {
  int op(int a, int b) {
    if (a>b) return a;
    else return b;
  }
});
```

It's nice that all of the logic is here, and not somewhere else in a separate `BinOper class` definition.

However, even this is a bit verbose and hard to read.

Example: lambda expressions for reduce

Instead, use a lambda expression.

```
int[] i_arr = ...;

int max = reduce(i_arr,
  (a,b) -> {
    if (a>b) return a;
    else return b;
  }
);
```

A lamba expression is an anonymous `class`.

`(a,b)` is the method inputs. What comes after `->` is the method body.

Example: lambda expressions for reduce

We can simplify the method with the conditional operator.

```
int[] i_arr = ...;

int max = reduce(i_arr,
  (a,b) -> { return (a>b) ? a : b; }
);
```

Example: lambda expressions for reduce

When a lambda expression's method body consists of a single `return` statement, we can omit the curly braces and `return`.

```
int[] i_arr = ...;

int max = reduce(i_arr, (a,b) -> (a>b) ? a : b);
int sum = reduce(i_arr, (a,b) -> a+b);
```

Example: lambda expressions for Predicate

We can write a named class that implements Predicate<T>.

```
class Pred implements Predicate<Number> {
  @Override
  public boolean test(Number n) {
    return n.doubleValue()>=0;
  }
}
```

```
Set<Integer> s1 = new HashSet<>();
...
List<Integer> pos_entries
                  = ListUtil.select(s1,new Pred());
```

Example: lambda expressions for Predicate

However, it's much cleaner to use a lambda expression.

```
List<Integer> pos_entries
 = ListUtil.select(s1, (n)->n.doubleValue()>=0);
```

When a lambda expression's parameter list has one parameter, we can omit the parentheses.

```
List<Integer> pos_entries
 = ListUtil.select(s1, n->n.doubleValue()>=0);
```

Example: lambda expressions for ActionListener

You can use lambda expressions when the `return` type is `void`.

```
button2.addActionListener(
  (e) -> { System.out.println(text.getText()); }
);
```

(`ActionListener`'s `abstract` method `returns` `void`.)

When the method `returns` `void` and the method body has only one statement, you can omit the curly braces.

```
button2.addActionListener(
        e -> System.out.println(text.getText()));
```

Lambda expressions with references

References can refer to an instance of an anonymous `class` written with a lambda expression.

```
ActionListener AL2 =
        e -> System.out.println(text.getText());
button2.addActionListener(AL2);

BinOper sum_op = (a,b) -> a+b;
int sum = reduce(i_arr, sum_op);
```

How do lambda expressions work?

Lambda expressions only work with functional `interfaces`.

The single method you write is the implementation of the single `abstract` method.

The method inputs are the method inputs of the single `abstract` method, which types are known. So the types of the inputs are inferred from the method's signature.

The method's `return` type are the `abstract` method's `return` type, which are known. So the `return` type is inferred from the method's signature.

Method reference

Imagine you wrote class Person and you have

```
public class PersonComp {
  public static int cmpByName(Person a, Person b)
  {
    return a.getName().compareTo(b.getName());
  }

  public static int cmpByAge(Person a, Person b)
  {
    return a.getBirthday().
                        compareTo(b.getBirthday());
  }
}
```

Method reference

You can use other methods to write a lamba expression.

```
Person[] p_arr = ...
//use static method
Arrays.sort(p_arr,
                (a,b)->PersonComp.cmpByAge(a,b) );

String[] s_arr = ...
//use instance method
Arrays.sort(stringArray,
                (a,b)->a.compareToIgnoreCase(b) );
```

However, even this is more complicated than necessary, because the lambda expression simply calls another method.

Method reference

To simplify these lambda expressions, you can use method references.

```
Person[] p_arr = ...
Arrays.sort(p_arr, PersonComp::cmpByName);

String[] s_arr = ...
Arrays.sort(s_arr, String::compareToIgnoreCase);
```

When a lambda expression simply calls another method, you can simplify it into a method reference.

Outline

Procedural programming vs. object oriented programming

Procedural and object oriented programming are two contrasting programming styles.

Procedural programming revolves around using procedures (functions that are called not for their return value, perhaps because they return `void`) to operate on chunks of data (e.g. `structs` of C).

Object oriented programming revolves around objects, collections of data members accompanied by useful member functions.

Procedural programming vs. object oriented programming

C is called a procedural programming language because it provides the tools for procedural programming.

Java and C++ are called are object oriented languages because they also provide the tools for object oriented programming.

Imperative programming vs. functional programming

Imperative and functional programming are two contrasting styles.

Imperative programming describes *how* a program should operate (to accomplish what you want) via variables and loops.

Functional or declarative programming describes *what* a program should accomplish via functions that do not cause side effects. Functions in functional programming can be understood via inputs and outputs, as they have no side effects.

Imperative programming vs. functional programming

Haskell and Lisp are called functional programming languages because the languages provide the tools for functional programming.

Java and C++ were called are imperative languages because the languages provide the tools for imperative programming.

Java 8 and C++11 introduced major functional programming features into the languages. Now functional programming on these languages is much more effective.

Example: summing even numbers between 1 and 1000

The imperative way of summing all integers between 1 and 1000 is

```
int sum = 0;
for (int i=0; i\le 1000; i++)
  if (i%2==0)
    sum += i;
```

You tell (as if you're speaking in an "imperative mood") how the program should operate to achieve what you want.

Example: summing even numbers between 1 and 1000

The functional way is

```
int sum = IntStream.rangeClosed(1,1000)
   .filter(a->a%2==0)
   .sum();
```

You're saying "create a stream of integers from 1 to 1000, filter to keep only even numbers, and sum them".

java.util.Stream

In Java 8, `Stream<T>`s were introduced. These are different from I/O streams.

`IntStream` and `DoubleStream` provide the functionality of `Stream<T>`s for the corresponding primitive types.

We won't cover `Stream<T>`s in detail. Just think of them as vectors that you can functionally operate on.

Example: functional Monte Carlo

Remember the Monte Carlo example? Here's how to do it functionally.

```
Random rand = new Random(seed);
double apx_pi = IntStream.generate( ()->{
    double x = rand.nextDouble();
    double y = rand.nextDouble();
    return (x*x+y*y<=1) ? 1 : 0;} )
  .limit(1_000_000)
  .average()
  .getAsDouble()
  * 4;
```

Example: more functional examples

```
List<Person> peopleList;
...

double averageMaleAge = peopleList.stream()
  .filter(p -> p.getGender() == Person.MALE)
  .mapToInt(Person::getAge)
  .average()
  .getAsDouble();
```

This code is very readable. The step-by-step logic of computing the average male age is very easy to follow.

Example: more functional examples

Imagine `class Person` has a complicated and sophisticated method that can roughly predict whether a person will like beer.

```
List<Person> peopleList;
...

peopleList.stream
   .filter(p -> p.getAge() >= 21)
   .filter(Person::probablyLikesBeer)
   .forEach( p -> System.out.println(
                  p.firstName() + " wants a beer") );
```

You could easily replace the last lambda expression to `p -> sendBeerSpamMailTo(p)`.

Why functional programming?

Advantages of functional programming:

- Functional programming is well-suited for certain problems, such as problems of data science.
- Functional programming often results in clean code.
- Functions without side effects (those mostly used in functional programming) tend to be easier to test (debug).
- Parallelization is easier.

Disadvantages of functional programming:

- Functional code is slower. Imperative code is closer to how CPUs actually work and is therefore faster.
- The functional style of thinking takes some getting used to.

So with what style should I program?

Programming style is not binary. A practical coding style will be on a spectrum.

Most code is partly procedural and partly object oriented.

Most code is partly imperative and partly functional.

A programming language can encourage but cannot bind you to a certain programming style. You can write object oriented and functional programs in C, although doing so will be a bit awkward.

In my opinion, it's best to let the task pick the tool.