Lambda Expressions in Java

Group 9
Labiba Labanya, Tahmid Muttaki, Sean Copenhaver, Matt Withrow Capone, Danny Chhour

A lambda expression:

- A short block of code which takes in parameters and returns a value
- Similar to methods; but they do not need method name and can be implemented in the body of a method; usually passed as parameter to a function
- Expression cannot contain variables, assignments or statements such as if or for
- Using code blocks, complex operations can be done using lambda expressions
- Replaces anonymous inner class syntax with simpler and shorter version of code

```
public static void main(String... args) {
    Runnable r2 = () -> System.out.println("Howdy, world!");
    r2.run();
  }
```

Syntax

Lambda expression has 3 elements:

- Parameter
- Token/symbol
- Logic

General Expression:

(parameter, anotherParameter) -> {return anotherParameter + parameter};

2 cases:

Single parameter: *parameter -> {return “Hello”;};*

No parameter: *() -> {return 10;};*

Logic execution

- Single line logic: don’t need return statement and braces

  (a, b) -> a + b;

- Void return:

  With logic: can’t use keyword return

  Expression: *() -> {System.out.println(“ ”)}*

  Without logic: *() -> {};*

Functional Interfaces

- A functional interface is an interface with just one abstract method
- Examples of functional interfaces include: Comparable, Runnable, Comparator
- Used to pass code to a function

```
Collections.sort(listDevs, new Comparator<Developer>() {
    @Override
    public int compare(Developer o1, Developer o2) {
        return o1.getName().compareTo(o2.getName());
    }
});
```

Functional Interfaces & Lambda Expressions

- Lambda expressions can be written in place of functional interfaces

Functional Interface

```
Collections.sort(listDevs, new Comparator<Developer>() {
    @Override
    public int compare(Developer o1, Developer o2) {
        return o1.getName().compareTo(o2.getName());
    }
});
```

Functional Interface with Lambda Expression

```
listDevs.sort((o1, o2)->o1.getName().compareTo(o2.getName()));
```

Variable Capture

Java lambda expressions are capable of accessing variables declared outside the lambda function body under certain circumstances.

Java lambdas can capture the following types of variables:

- Local variables
- Instance variables
- Static variables

Variable Capture

Local Variable Capture

```
public interface MyFactory {
    public String create(char[] chars);
}
```

```
MyFactory myFactory = (chars) -> {
    return new String(chars);
};
```

```
String myString = "Test";

MyFactory myFactory = (chars) -> {
    return myString + ":" + new String(chars);
};
```

Instance Variable Capture

```
public class EventConsumerImpl {

    private String name = "MyConsumer";

    public void attach(MyEventProducer eventProducer){
        eventProducer.listen(e -> {
            System.out.println(this.name);
        });
    }
}
```

Static Variable Capture

```
public class EventConsumerImpl {
    private static String someStaticVar = "Some text";

    public void attach(MyEventProducer eventProducer){
        eventProducer.listen(e -> {
            System.out.println(someStaticVar);
        });
    }
}
```

Advantages of lambda expressions

- Readability:
  - Need fewer lines of code
  - Readable without interpretation

```
List<String> colors = Arrays.asList("red", "yellow", "green");

colors.forEach(color -> System.out.println(color));
```

- Higher efficiency:
  - Sequential and parallel execution support by passing behavior as an argument in methods using Stream API
  - Higher efficiency (parallel) can be achieved in case of bulk operations on collection

```
Stream<Dog> dogStream = Stream.of(dogArray);

Stream<Dog> sortedDogStream = dogStream.sorted((Dog m, Dog n) -> Integer.compare(m.getHeight(), n.getHeight()));
```

Advantages of lambda expressions

- Compact:
  - No need to create inner class
  - Every inner class creates a .class file, lambda expression eliminates that and reduces deployment artifacts.
  - Reduces the size of jar file

- Essence of functional programming:
  - Passing a lambda expression to another function allows us to pass not only values but also behaviors
  - Raises the level of abstraction and allows to build more generic, flexible and reusable API.

Common Use - Android Application

- Android applications are now written in Kotlin
  - Similar to Java

- Application needs to know when an event has occurred and how to react to that change
  - What to do when they swipe left?
  - What to do when the touch an icon?

setOnClickListener

Added in API level 1

```
public void setOnClickListener (View.OnClickListener l)
```

Register a callback to be invoked when this view is clicked. If this view is not clickable, it becomes clickable.

Parameters	
`l`	`View.OnClickListener`: The callback that will run This value may be `null`.

View.OnClickListener

Added in API level 1

Kotlin | **Java**

```
public static interface View.OnClickListener
```

android.view.View.OnClickListener

Known indirect subclasses
CharacterPickerDialog, KeyboardView, QuickContactBadge

Interface definition for a callback to be invoked when a view is clicked.

Summary

Public methods	
`abstract void`	`onClick(View v)` Called when a view has been clicked.

Without Lambda Expressions

```
mButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // do something here
    }
});
```

With Lambda Expressions

```
mButton.setOnClickListener((View v) -> {
    // do something here
});
```

Discussion Questions

1. What other uses of lambda expressions can you think of besides click events in mobile development?
2. When are inner classes more appropriate than lambda expressions?