Table of Contents

1 Understanding Lambdas

1.1 Why Lambdas

1. Enable functional programing
2. Readable and concise code
3. Easier to use API's and libraries
4. Enables support for parallel processing.

1.2 Functional vs Object Oriented Programming

Functional programs let us to write better code, more readable and more maintainable code.

Functional program is another tool for programmers.

1.3 Passing behavior in OOP

In OOP Everything is an object, All code blocks are "associated" with classes and objects.

Till Java 7, we can pass behavior by using method in class/Interface (by implementation), So behavior is belongs to a class and will switched by using an implementation of an interface.

- **We can't have a function or method on its own, it has to be in a class.**
- **We can't define a function/method on isolation**

For example –

```
public class Greeter {
        public void greet(Greeting greeting) {
                greeting.perform();
        }
        public static void main(String[] args) {
                Greeter greeter = new Greeter();
                HelloWorldGreeting helloWorldGreeting = new HelloWorldGreeting()
                greeter.greet(helloWorldGreeting)
        }
}
public interface Greeting {
        public void perform();
}
public class HelloWorldGreeting implements Greeting {
        public void perform() {
                System.out.print("Hello world!");
        }
}
```

1.4 Introducing Lambda Expressions

- In the above example, we are **not just passing the behavior itself**; we are passing a thing (class instance) which has a behavior.

  Like below -

```
public void greet(Greeting greeting) {
        greeting.perform();
}
```

- What if we pass action in isolation (which is not belongs to a class) pass like below and just execute the action.

```
public void greet(action) {
        action();
}
```

Lambdas are introduced just to achieve this, and Lambda's lets you create these entities, which are just functions, which don't belong to a class. They are just functions exists in isolation. They are called lambda expressions.

1.5 Functions as values and Lambda expressions

Inline values in java are like below –

```
String name="foo"

Int num= 10;

Student a = new Student()

Student b = a;
```

Can we assign a block of code it to a variable? Can we do like this?

```
ablockOfcode  =  {
        --------
        --------
        --------
}
```

In java, what is the way of writing a block of code?

It's just defining a method, to have a block of code.

How can we assign method to a variable?

Imagine assign the method to a variable, (not execution, just assignment), it's just assigning a block of code (method) to a variable.

Like below –

```
aBlockOfCode    =   public void greet() {
                      System.out.println("hello")
                    }
```

This can be done in java by using Lambda expressions, In the above code snippet we can get rid of unnecessary information like below –

1. As function existing in isolation, doesn't make sense to call it as "public/privat", so we can get rid of "public" in the above declaration like below –

```
aBlockOfCode    =   void greet() {
                      System.out.println("hello")
                    }
```

2. In the above code we don't need to have 2 names for a variable (**aBlockOfCode**, **greet**) , So aBlockOfCode is a variable we going to use to refer the above block of code, so we can get rid **"greet"** like below –

```
aBlockOfCode    =   void  () {
                      System.out.println("hello")
                    }
```

3. And, step further, by using type inference, java compiler can find the return type by using the expression in the block of code, so we don't need to specify the return type, so remove "void" like below –

```
aBlockOfCode    =   () {
                      System.out.println("hello")
                    }
```

4. With small addition in the above statement we can define the lambda expression like below.

```
aBlockOfCode    =   () -> {
                      System.out.println("hello")
                    }
```

5. If body of the lambda is just one line then Lambda expression will be like below –

```
aBlockOfCode    =   () -> System.out.println("hello")
```

1.6 Lambda expressions Examples

Consider below example –

```
greetingFucntion    =    ()  -> System.out.println("hello")
```

In the above statement , **"() -> System.out.println("hello")"** is a function assigned to a variable **"greetingFucntion "**

What is the type of the variable "greetingFucntion" ? Hold that thought a bit for now.

This variable can be passed around, and can be sent to other method/ functions. Something Like below (but not exactly same in Java 8) –

```
Public void greet(greetingFucntion  ){
  greetingFucntion  ()
}
greet(()  -> System.out.println("hello"));
```

More Functions -

Functions can be sometimes with parameters.

```
doubleNumberFucniton =    public int douleTheNum (int a) {
          return a*2
}
```

Lambda expression for above function is –

```
doubleNumberFucniton =    (int a)  - >  {a *2 }
doubleNumberFucniton =    (int a)  - >  a *2
```

Few more examples-

```
addFucniton  =    (int a, int b)  - >  a + b;
safeDivFucniton  =   (int a, int b)  - >  {
          if( b == 0) return 0;
          return a / b;
};
stringLengthFucniton  =    (String a)  - >  s.length;
```

Questions:

What is the type of variable?

How to execute the function?

1.7 Lambda as interface type

Java implementers could have done like this, to define a lambda expressions, assign it to a "FunctionType" type variable.

FunctionType<Void, void> lambdaGreeting = () -> System.out.print("Hello world!");

But we don't have a type called FunctionType in java, instead of that java implementer introduced a concept **called Functional Interface.**

Below are the steps, to define function Type.

1. Define an interface.

   ```
   Interface MyLambda {
   }
   ```

2. Define **only one abstract method** in interface with "same exact signature as Lambda"**, If we have multiple methods compiler doesn't know which method to use for lambda expression and gives an error.**

   ```
   interface MyLambda {
       void foo();
   }
   ```

3. Now use this "Interface name" as a type for lambda expression, it will act as a "function" Type and compiler will be happy with that. **This is how we will create Function Types.**

4. So we used to call it as Functional interface.

   ```
   MyLambda lambdaGreeting = () -> System.out.print("Hello world!");
   ```

One more example –

To create a Type of below lambda, we need to create a one more interface like this-

```
MyAdd addFucniton  =   (int a, int b)  - >  a + b;

interface MyAdd {
    int  addXYZ(int a, int b);
}
```

In the above steps, Interface and method names are doesn't matter.

only matter's is lambda expression signature should match the signature of method.

If you have interface which is already matching the signature of the lambda we can use it, If you observe Greeting interface has the same signature with one method, So we can use that Interface like below

```
Greeting lambdaGreeting = () -> System.out.print("Hello world!");
```

1.8 Lambda vs Interface implementation

What is the difference between below two statements?

a) **HelloWorldGreeting helloWorldGreeting = new HelloWorldGreeting()**
b) **Greeting lambdaGreeting = () -> System.out.print("Hello world!");**

In the first line (a) – we are creating an object for direct implementation class to Interface

In the second line (b) – Lambda expression is assigned to Interface type, and lambda expression will be executed like below, just as if it was an instance of a class –

lambdaGreeting.perform()

In a way we are implementing an interface by using the Lambda in line "b"

There are differences; certain things are different between two implementations, though differences are hard to find.

Like line (a), Interfaces can be implemented with anonymous inner class, like below –

```
Greeting lambdaGreeting =  new Greeting() {
        Public void Perform() {
                System.out.print("Hello world!");
        }
};
```

- **Mostly think that lambda's are short cuts of anonymous class, but it's little different.**
- **Lambda is its own thing. Don't think like it's just short cut to Anonymous class.**

```
public class Greeter {
        public void greet(Greeting greeting) {
                greeting.perform();
        }
        public static void main(String[] args) {
                Greeter greeter = new Greeter();
                Greeting lambdaGreeting = () -> System.out.print("Hello world!");
                Greeting innerClassGreeting = new Greeting() {
                        public void perform() {
                                System.out.print("Hello world!");
                        }
                };
                greeter.greet(() -> System.out.print("Hello world!"));
                greeter.greet(innerClassGreeting);
        }}
```

Both above lines print “Hello world”

1.9 Type inference

In java 8, Type inference is the concept used to infer the type definition, samething applies to Lambda’s

```
public class TypeInferenceExample {
	public static void main(String[] args) {
		StringLengthLambda stringLengthLambda = (String s) -> s.length();
		StringLengthLambda stringLengthLambda = (s) -> s.length(); // Type inference
		StringLengthLambda stringLengthLambda = s -> s.length(); // remove brakets
		printLambda(s -> s.length()); // Pass lambda expression to a method

	}
	public static void printLambda(StringLengthLambda l) {
		System.out.print(l.getLength("Hello Lambda"));
	}
	interface StringLengthLambda {
		int getLength(String s);
	}
}
```

1.10 Runnable using Lambda

Though there are several other reasons, most important reason to use “**Interface”** as type for lambda expression instead of creating a new ‘Function” Type is backward compatibility.

Java API’s interfaces which are having only one method are good candidates to use lambda expressions.

Example – Runnable interface.

```
public class RunnableExample {
	public static void main(String[] args) {
		Thread myThread = new Thread(new Runnable() {
			@Override
			public void run() {
				System.out.println("Printed inside Runnable");
			}
		});
		myThread.run();
		Thread myLambdaThread = new Thread(() -> System.out.println("Lambda
		Runnable"));
		myLambdaThread.run();
		}
	}
```

2 Functional Interface

- If an interface has only one abstract method then we call it as functional Interface.
- In java 8 we can define method with implementations.

Ex – If we have an Interface with 3 methods –

If two methods are implemented and one is abstract, then this kind of interface can be considered for Lambda expressions, and these are called functional Interfaces.

- At later point of time if somebody add a new abstract method then it's no longer a functional Interface.
- In java, we can mark an Interface as functional Interface by using an annotation called @FunctionalInterface, which will force the implementers to have only one abstract method in interface.
- If an interface has used to be an interface for Lambda then it should not be changed by adding a new abstract method in future for backward compatibility.

```
@FucntionalInterface // though its optional, but best practice to use it
public interface Greeting {
        public void perform();
}
```

- Lambda is simply an implementation of the method in the Interface.

3 Lambdas Exercise

Implement an example in Java 7 and by using lambda in java 8.

```
public class Person {
	private String firstName;
	private String lastName;
	private int age;
	public Person(String firstName, String lastName, int age) {
		super();
		this.firstName = firstName;
		this.lastName = lastName;
		this.age = age;
	}
	@Override
	public String toString() {
		return "Person [firstName=" + firstName + ", lastName=" + lastName + ", age="
		+ age + "]";
	}
	// getter and setters
}


public class Unit1Exercise {
	public static void main(String[] args) {
		List<Person> people = Arrays.asList(
				new Person("Charles", "Dickens", 60),
				new Person("Lewis", "Carroll", 42),
				new Person("Thomas", "Carlyle", 51),
				new Person("Charlotte", "Bronte", 45),
				new Person("Matthew", "Arnold", 39)
				);
		// Step 1: Sort list by last name
		// Step 2: Create a method that prints all elements in the list
		// Step 3: Create a method that prints all people that have last name beginning
		with C
	}
}
```

Java 7 way –

```
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import io.javabrains.common.Person;
public class Unit1ExerciseSolutionJava7 {

	public static void main(String[] args) {
		List<Person> people = Arrays.asList(
				new Person("Charles", "Dickens", 60),
				new Person("Lewis", "Carroll", 42),
				new Person("Thomas", "Carlyle", 51),
				new Person("Charlotte", "Bronte", 45),
				new Person("Matthew", "Arnold", 39)
				);

		// Step 1: Sort list by last name
		Collections.sort(people, new Comparator<Person>() {
			@Override
			public int compare(Person o1, Person o2) {
				return o1.getLastName().compareTo(o2.getLastName());
			}
		});

		// Step 2: Create a method that prints all elements in the list
		System.out.println("Printing all persons");
		printAll(people);

		// Step 3: method that prints all people having last name beginning with C
		System.out.println("Printing all persons with last name beginning with C");
		printConditionally(people, new Condition() {
			@Override
			public boolean test(Person p) {
				return p.getLastName().startsWith("C");
			}
		});
		System.out.println("Printing all persons with first name beginning with C");
		printConditionally(people, new Condition() {
			@Override
			public boolean test(Person p) {
				return p.getFirstName().startsWith("C");
```

```
                        }
                });

        }
        private static void printConditionally(List<Person> people, Condition condition) {
                for (Person p : people) {
                        if (condition.test(p)) {
                                System.out.println(p);
                        }
                }
        }
        private static void printAll(List<Person> people) {
                for (Person p : people) {
                        System.out.println(p);
                }
        }
}
interface Condition {
        boolean test(Person p);
}
```

Java 8 way

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import io.javabrains.common.Person;

public class Unit1ExerciseSolutionJava8 {

public static void main(String[] args) {
	List<Person> people = Arrays.asList(
				new Person("Charles", "Dickens", 60),
				new Person("Lewis", "Carroll", 42),
				new Person("Thomas", "Carlyle", 51),
				new Person("Charlotte", "Bronte", 45),
				new Person("Matthew", "Arnold", 39)
				);

	// Step 1: Sort list by last name
	Collections.sort(people, (p1, p2) -> p1.getLastName().compareTo(p2.getLastName()));

	// Step 2: Create a method that prints all elements in the list
	System.out.println("Printing all persons");
	printConditionally(people, p -> true);

	// Step 3: Create a method that prints all people that have last name beginning with C
	System.out.println("Printing all persons with last name beginning with C");
	printConditionally(people, p -> p.getLastName().startsWith("C"));

	System.out.println("Printing all persons with first name beginning with C");

	printConditionally(people, p -> p.getFirstName().startsWith("C"));
}

private static void printConditionally(List<Person> people, Condition condition) {
	for (Person p : people) {
		if (condition.test(p)) {
			System.out.println(p);
		}
	}
}
}
```

4 Using Functional Interfaces

In the above example, Condition is an overhead to create it. In Java 8, designers created some predefined interfaces for common scenarios. So we can use those out of the box interfaces.

Example – Predicate functional Interface, above code snipper can be changes like below –

```
Import java.util.function.*
private static void printConditionally(List<Person> people, Predicate predicate) {
        for (Person p : people) {
                if (predicate.test(p)) {
                        System.out.println(p);
                }
        }
}
```

There are plenty of functional interfaces in the above package.

More java functional Interfaces:

```
Main() {

        // Step 1: Sort list by last name
        Collections.sort(people, (p1, p2) -> p1.getLastName().compareTo(p2.getLastName()));

        // Step 2: Create a method that prints all elements in the list
        performConditionally(people, p -> true, p -> System.out.println(p));

        // Step 3: Create a method that prints all people that have last name beginning with C
        performConditionally(people, p -> p.getLastName().startsWith("C"), p ->
        System.out.println(p));

        performConditionally(people, p -> p.getFirstName().startsWith("C"), p ->
        System.out.println(p.getFirstName()));
}

private static void performConditionally(List<Person> people,
                        Predicate<Person> predicate, Consumer<Person> consumer) {
        for (Person p : people) {
                if (predicate.test(p)) {
                        consumer.accept(p);
                }
        }
}
```

5 Exception handling in Lambdas

How to handle exceptions in lambda?

We can handle exceptions in wherever lambda's are executed, however better approach would be wrap the lambda in another lambda and try to handle the exceptions in wrapped lambda to make actual business logic separate from exception logic.

```
public class ExceptionHandlingExample {

    public static void main(String[] args) {
        int [] someNumbers = { 1, 2, 3, 4 };
        int key = 2;
        process(someNumbers, key, wrapperLambda((v, k) -> System.out.println(v / k)));
    }
    private static void process(int[] someNumbers, int key,
                                BiConsumer<Integer, Integer> consumer) {
        for (int i : someNumbers) {
            consumer.accept(i, key);
        }
    }
    private static BiConsumer<Integer, Integer>
            wrapperLambda(BiConsumer<Integer, Integer> consumer) {
        // return  consumer;
        // return  ((v, k) -> System.out.println(v+ k);
        return (v, k) ->  {
            try {
                consumer.accept(v, k);
            } catch (ArithmeticException e) {
                System.out.println("Exception caught in wrapper lambda");
            }
        };
    }
}
```

6 Closures in Lambda Expressions

- Effectively final variable “b” will be used in Anonymous class at line number “a”.
- Frozen variable “b” value “20” is used in lambda expression like below statement , the value will be passed through lambda expression.

```
public class ClosuresExample {
	public static void main(String[] args) {
		int a = 10;
		int b = 20; // is having a frozen
		doProcess(a,  new Process() {
				public void process(int i){
					System.out.println(i + b); // line --- (a)
				}
			}
		);

		doProcess(a, i -> System.out.println(i + b));  //  line --- (b)
	}
	public static void doProcess(int i, Process p) {
		p.process(i);
	}
}

interface Process {
	void process(int i);
}
```

7 “this” reference in Lambdas

- In static method can’t access “this reference in static context.
- In static main method, anonymous class can access “this” reference which refers to anonymous class instance.
- In Anonymous class “this” reference will be overridden by anonymous class instance.
- When we use lambda in static method, we can’t use “this” reference as lambda just infers the “this” reference from enclosing instance.
- “this” reference is not modified (untouched) inside a lambda. Lambda expression doesn’t override “this” reference from enclosing context.
- If lambda expression is in an instance method then “this” refers to the object that the instance method invoked.

```
public class ThisReferenceExample {
    public void doProcess(int i, Process p) {
            p.process(i);
    }

    public void execute() {
            doProcess(10, i -> {
                    System.out.println("Value of i is " + i);
                    System.out.println(this);
            });
    }

    public static void main(String[] args) {
            thisReferenceExample.doProcess(10, new Process() {
                    @Override
                    public void process(int i) {
                            System.out.println("Value of i is " + i);
                            System.out.println(this);  // Prints anonymous inner class ref.
                    }
                    public String toString() {
                            return "This is anonymous class instance";
                    }

            });
            ThisReferenceExample thisReferenceExample = new ThisReferenceExample();
            /* thisReferenceExample.doProcess(10, i -> {
                    System.out.println("Value of i is " + i);
                    // System.out.println(this);  //with lambdas it won’t work
            });
```

```
		*/
		thisReferenceExample.execute();
	}

	public String toString() {
		return "This is the main ThisReferenceExample class instance";
	}
}
```

8 Method references

- Mehtod reference is the new concept in Java 8.
- In lambda expression, when we execute a method in lambda expression by using the same input parameter, we can use method reference which is a short hand like below
- Below lambda is not taking any parameter and method printMessage() is also not taking any parameter then, lambda expression can be replaced with a method reference.

```
// MethodReferenceExample1::printMessage === () -> printMessage()

public class MethodReferenceExample1 {
	public static void main(String[] args) {
	Thread t = new Thread(MethodReferenceExample1::printMessage);  // () -> method()
		// MethodReferenceExample1::printMessage === () -> printMessage()
		t.start();
	}
	public static void printMessage() {
		System.out.println("Hello");
	}
}
```

One more example –

```
public class MethodReferenceExample2 {
	public static void main(String[] args) {
		List<Person> people = Arrays.asList(
					new Person("Charles", "Dickens", 60),
					new Person("Lewis", "Carroll", 42),
					new Person("Matthew", "Arnold", 39)
					);
		System.out.println("Printing all persons");
		performConditionally(people, p -> true, System.out::println); // p -> method(p)
	}
	private static void performConditionally(List<Person> people,
			Predicate<Person> predicate,  Consumer<Person> consumer) {
		for (Person p : people) {
			if (predicate.test(p)) {
				consumer.accept(p);
			}
		}
	}
}
```

9 Collections improvements

9.1 ForEach Loop

- In java 8, new forEach() loop got introduced which is an **internal for loop** as opposed **external for loops** available in till Java 7.
- forEach() is an internal for loop which are executed by compiler, and program doesn't have control on it.
- forEach() can enable compiler to run the loop in parallel.

```
public class CollectionIterationExample {
        public static void main(String[] args) {
                List<Person> people = Arrays.asList(
                                new Person("Charles", "Dickens", 60),
                                new Person("Lewis", "Carroll", 42),
                                new Person("Matthew", "Arnold", 39)
                                );
                System.out.println("Using for loop");
                for (int i = 0; i < people.size(); i++) {
                        System.out.println(people.get(i));
                }

                System.out.println("Using for in loop");
                for (Person p : people) {
                        System.out.println(p);
                }

                System.out.println("Using lambda for each loop");
                people.forEach(System.out::println);
        }
}
```

9.2 Streams

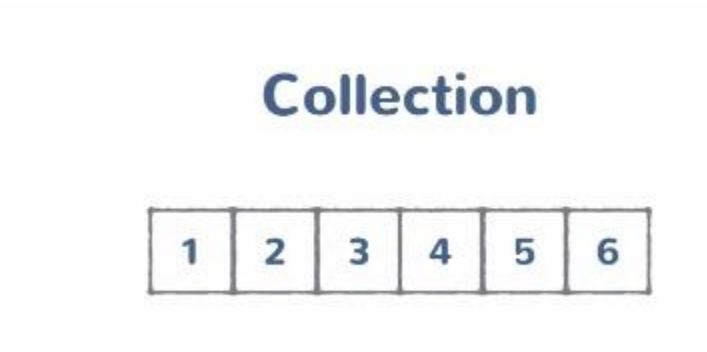
A sequence of elements supporting sequential and parallel aggregate operations.

In the below collection, if we have multiple operation to be done, iterating through all the collection for each operation is not an effective way.

Example –

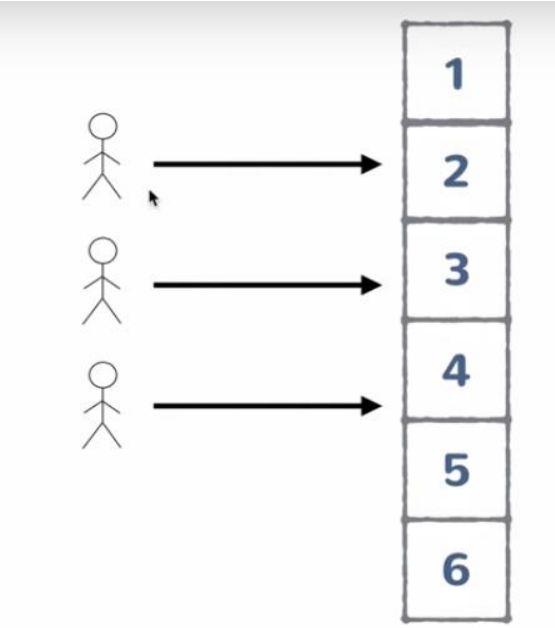
If 3 persons are working in manufacturing company to fix the engine, color the car and fix the tiers respectively.

If each of them iterate through 6 cars in a below pattern one by one then it's an ineffective way of iteration.



Solution with Streams

Assume Cars in a conveyor belt or assembly line, which are passing through, and workers stayed at same place to operate on each element while they come in.



```
public class StreamsExample1 {
	public static void main(String[] args) {
		List<Person> people = Arrays.asList(
				new Person("Charles", "Dickens", 60),
				new Person("Lewis", "Carroll", 42),
				new Person("Thomas", "Carlyle", 51),
				new Person("Charlotte", "Bronte", 45),
				new Person("Matthew", "Arnold", 39)
				);

		people.stream()
		.filter(p -> p.getLastName().startsWith("C"))
		.forEach(p -> System.out.println(p.getFirstName()));

		long count = people.parallelStream()
		.filter(p -> p.getLastName().startsWith("D"))
		.count();

		System.out.println(count);
	}
}
```

Streams are having intermediate and terminated operations.

- Intermediate operation doesn't perform any operation instead it creates intermediate steam for the final terminated operation.
- Terminated operation is final operation which will be performed on Streams once after the terminated operation no further operation will be performed on stream.

Intermediate operation:

- filter
- map
- flatmap
- peek
- distinct
- sorted
- limit

Terminated operation

- forEach
- toArray

- reduce
- collect
- min
- max
- count
- anymatch
- allMatch
- noneMatch
- findFirst
- findAny