Java Functional Programming

Lambda Expression

Chang Xu

Agenda

1. Who am I?
2. Introduction to lambda expression
3. Motivation of lambda
4. Anonymous class
5. Simple lambda expression syntax
6. Lambda examples

Who Am I ?

Name: Chang Xu (https://web.njit.edu/~cx62/)

Oracle Certified Associate, Java SE 8 Programmer

An Oracle Certified Associate, Java SE 8 Programmer has demonstrated knowledge of object-oriented concepts, the Java programming language and general knowledge of Java platforms and technologies.

ISSUED BY
Oracle

ISSUED TO
Chang Xu

EARNER EMAIL

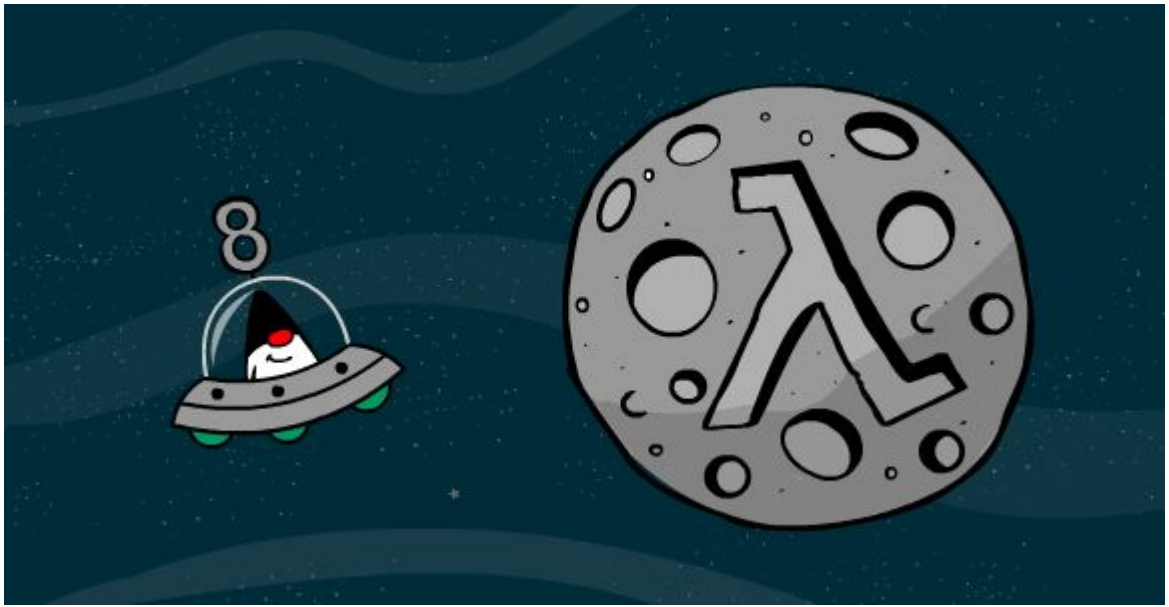
ISSUED ON
20 Feb 2016

SKILLS

Application Development | Java | Java SE 8 | Oracle Associate | Programming

Java 8 - Introduce Lambda Expression

Mark Reinhold, Oracle's Chief Architect, describes Lambda expressions as the single largest upgrade to the programming model ever — larger even than generics.



Why Java needs Lambda

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

Anonymous Class

1. Enable you to declare and instantiate a class at the same time
2. They are like local classes except that they do not have a name.
3. Use them if you need to use a local class only once.

Anonymous Class

```
HelloWorld frenchGreeting = new HelloWorld() {
        String name = "tout le monde";
        public void greet() {
            greetSomeone("tout le monde");
        }
        public void greetSomeone(String someone) {
            name = someone;
            System.out.println("Salut " + name);
        }
    };
```

Anonymous Class

1. The new operator
2. The name of an interface to implement or a class to extend. In this example the anonymous class is implementing the interface HelloWorld.
3. Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression. Note: When you implement an interface, there is no constructor, so you use an empty pair of parentheses, as in this example.
4. A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

Anonymous Inner Class

In Java, anonymous inner classes provide a way to implement classes that may occur only once in an application. For example, in a standard Swing or JavaFX application a number of event handlers are required for keyboard and mouse events. Rather than writing a separate event-handling class for each event, you can write something like this.

```
JButton testButton = new JButton("Test Button");
  testButton.addActionListener(new ActionListener(){
  @Override public void actionPerformed(ActionEvent ae){
     System.out.println("Click Detected by Anon Class");
   } });
```

Class Example

Original code: https://web.njit.edu/~theo/courses/cs602/week6/lecture/6-7.html

AIC Implementation:

```
b.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
            if (flag){
                flag=false;
            }else{
                flag =true;
            }
                repaint();
            }
});
```

Class Example

Lambda Implementation:

```
b.addActionListener(      (e)    ->
        {
                if (flag){
                        flag=false;
                }else{
                        flag =true;
                }
                repaint();
        });
```

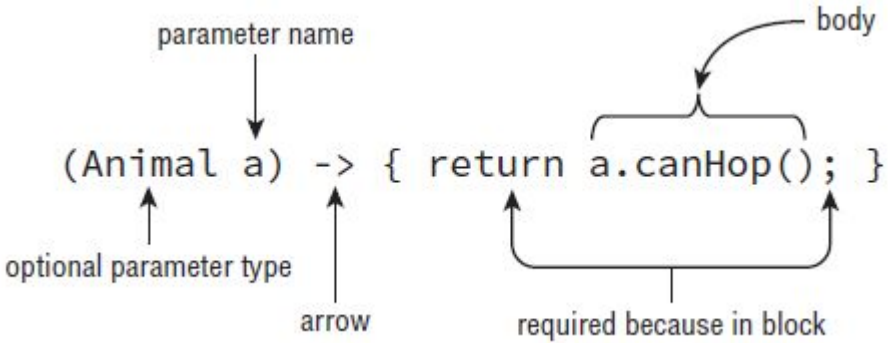
Lambda Quick View - Thread

```
//Old way:
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello from thread");
    }
}).start();

//New way:
new Thread(
    () -> System.out.println("Hello from thread")
).start();
```

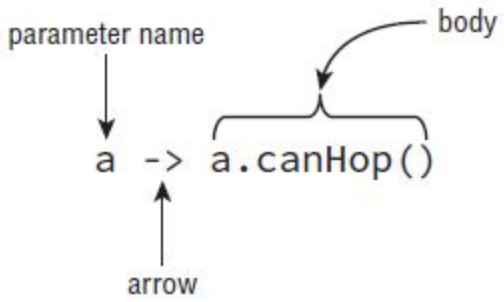
Lambda Quick View - Print List

```
//Old way:
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
for(Integer n: list) {
    System.out.println(n);
}

//New way:
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
list.forEach(n -> System.out.println(n));


//or we can use :: double colon operator in Java 8
list.forEach(System.out::println);
```

Simple Lambda Expression Syntax - Complete



Simple Lambda Expression Syntax - Simplified



Simplifying Rules

1. The parenthesis can only be omitted if there is a single parameter and its type is not explicitly stated.

2. Can omit braces when there is only one statement in the body.

3. Does not require to type return or semicolon when braces are omitted.

Examples on Lambda Syntax

1. print(() -> ture);
2. print(a -> a.startsWith(“Java”));
3. print((a, b) -> a.startsWith(“Java”));
4. print((String a, String b) -> a.startsWith(“Java”);
5. print(a, b -> a.startsWith(“Java”));
6. print(a -> { a.startsWith(“Java”); });
7. print(a -> { return a.startsWith(“Java”) });

Examples on Lambda Syntax

```
(a, b) -> { int a = 0;  return 5;}
```

```
(a, b) -> { int c = 0;  return 5;}
```

Coding Example 1

Programming Task:

Print out all the animals in a list according to some criteria.

Animals can hop or can swim.

Animals {fish, kangaroo, rabbit, turtle}

Initial Setup - Create Animal Class

```
public class Animal {
	private String species;
	private boolean canHop;
	private boolean canSwim;
	public Animal (String speciesName, boolean hopper, boolean swimmer) {
		species = speciesName;
		canHop = hopper;
		canSwim = swimmer;
	}
	public boolean canHop() { return canHop; }
	public boolean canSwim() { return canSwim; }
	public String toString() { return species; }
}
```

Initial Setup - Create CheckTrait Interface

```
public interface CheckTrait {
        boolean test (Animal a);
}
```

Initial Setup - Create CheckIfHopper Class

```
public class CheckIfHopper implements CheckTrait {
	public boolean test (Animal a) {
		return a.canHop();
	}
}
```

Code without Lambda

```
1: import java.util.*;
2: public class TraditionalSearch {
3:     public static void main(String[] args) {
4:         List<Animal> animals = new ArrayList<Animal>();
5:         animals.add( new Animal (“fish”, false, true));
6:         animals.add( new Animal (“kangaroo”, true, false));
7:         animals.add( new Animal (“rabbit”, true, false));
8:         animals.add( new Animal (“turtle”, false, true));
9:
10:        print (animals, new CheckIfHopper() );
11:    }
```

Code without Lambda

```
12:   private static void print (List<Animal> animals, CheckTrait checker) {
13:           for (Animal animal : animals) {
14:                   if (checker.test (animal) )
15:                           System.out.print (animal + " ");
16:                   }
17:           System.out.println();
18:   }
19:}
```

Code without Lambda

What if we want to print animals who can swim?

1. Traditional way is to write another class CheckIfSwims.
2. Instantiates the class under line 10.

OR

Improve The Code with Lambda

Simple Lambda Expression Implementation:

replace line 10 with:

```
print ( animals, a -> a.canHop() );
```

if we want to print swimmers:

```
print ( animals, a -> a.canSwim() );
```

How about animals cannot swim?

```
print ( animals, a -> ! a.canSwim() );
```

Lambda Works with Interface

Lambdas work with interfaces that have only one method.

These are called functional interfaces - interfaces that can be used with functional programming. (simplified definition)

We want to test Animals, Strings, Plants, and so on. Imagine we have to create lots of interfaces to use lambdas.

Predicate Interface

```
package java.util.function

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

T for generics syntax.

Improve Code - Implementing Predicate Interface

```
1: import java.util.*;
2: import java.util.function.*;
3: public class PredicateSearch {
4:     public static void main(String[] args) {
5:             List<Animal> animals = new ArrayList<Animal>();
6:             animals.add(new Animal("fish", false, true));
7:
8:             print(animals, a -> a.canHop());
9:      }
```

Improve Code - Implementing Predicate Interface

```
10:   private static void print(List<Animal> animals, Predicate<Animal> checker) {
11:         for (Animal animal : animals) {
12:               if (checker.test(animal))
13:               System.out.print(animal + " ");
14:         }
15:         System.out.println();
16:   }
17: }
```

More about Predicate Interface

Java 8 integrates Predicate interface into some existing classes.

For example:

ArrayList declares removeIf( ) method that takes Predicate

Programming Task:

We want to remove all of the bunnies' names that don't begin with the letter h.

More about Predicate Interface

```
List<String> bunnies = new ArrayList<>();
bunnies.add("long ear");
bunnies.add("floppy");
bunnies.add("hoppy");

System.out.println(bunnies);                    // [long ear, floppy, hoppy]

bunnies.removeIf(s -> s.charAt(0) != 'h');
System.out.println(bunnies);                     // [hoppy]
```

Coding Example 2

Name	Perform action on selected members
Primary Actor	Administrator
Preconditions	Administrator is logged in to the system.
Postconditions	Action is performed only on members that fit the specified criteria.
Main Success Scenario	1. Administrator specifies criteria of members on which to perform a certain action. 2. Administrator specifies an action to perform on those selected members. 3. Administrator selects the **Submit** button. 4. The system finds all members that match the specified criteria. 5. The system performs the specified action on all matching members.
Extensions	1a. Administrator has an option to preview those members who match the specified criteria before he or she specifies the action to be performed or before selecting the **Submit** button.

Initial Setup

```
public class Person {
    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public void printPerson() {
        // ...
    }
}
```

Approach 1

Create Methods That Search for Members That Match One Characteristic

```
public static void printPersonsOlderThan(List<Person>
roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

Approach 2

Create More Generalized Search Methods

```
public static void printPersonsWithinAgeRange(
    List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

Approach 3

Specify Search Criteria Code in a Local Class

```
public static void printPersons(
    List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Approach 3

```
interface CheckPerson {
    boolean test(Person p);
}

class CheckPersonEligibleForSelectiveService implements
CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

Approach 4

Specify Search Criteria Code in an Anonymous Class

```
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

Approach 5

Specify Search Criteria Code with a Lambda Expression

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Approach 6

Use Standard Functional Interfaces with Lambda Expressions

```
interface Predicate<Person> {
    boolean test(Person t);
}
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Approach 7

Use Lambda Expressions Throughout Your Application

```
public static void printPersonsWithPredicate(
    List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Approach 7

```
public static void processPersons(
    List<Person> roster,
    Predicate<Person> tester,
    Consumer<Person> block) {
        for (Person p : roster) {
            if (tester.test(p)) {
                block.accept(p);
            }
        }
}
```

Approach 7

```
processPersons(
      roster,
      p -> p.getGender() == Person.Sex.MALE
          && p.getAge() >= 18
          && p.getAge() <= 25,
      p -> p.printPerson()
);
```

Approach 7

```
public static void processPersonsWithFunction(
    List<Person> roster,
    Predicate<Person> tester,
    Function<Person, String> mapper,
    Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

Approach 7

```
processPersonsWithFunction(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions

Local class: Use it if you need to create more than one instance of a class, access its constructor, or introduce a new, named type (because, for example, you need to invoke additional methods later).

Anonymous class: Use it if you need to declare fields or additional methods.

Lambda expression:

- Use it if you are encapsulating a single unit of behavior that you want to pass to other code. For example, you would use a lambda expression if you want a certain action performed on each element of a collection, when a process is completed, or when a process encounters an error.
- Use it if you need a simple instance of a functional interface and none of the preceding criteria apply (for example, you do not need a constructor, a named type, fields, or additional methods).

Nested class: Use it if your requirements are similar to those of a local class, you want to make the type more widely available, and you don't require access to local variables or method parameters.

- Use a non-static nested class (or inner class) if you require access to an enclosing instance's non-public fields and methods. Use a static nested class if you don't require this access.

References

Code examples are from:

Jeanne Boyarsky & Scott Selikoff. OCA Oracle Certified Associate Java SE 8 Programmer I Study Guide. 2015.

Java 8 Lambda Expressions Tutorial with Examples. http://viralpatel.net/blogs/lambda-expressions-java-tutorial/

Oracle Java Documentation. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html