Lambda Expressions

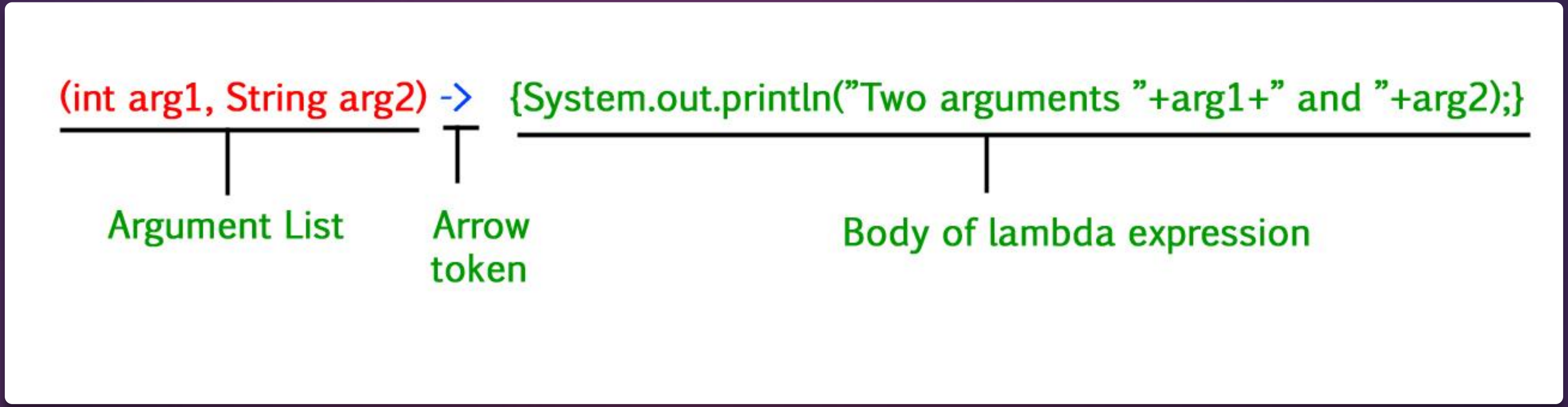
ANJU MUNOTH

Lambda expressions

- Introduced in Java SE 8
- Represent instances of functional interfaces (interfaces with a single abstract method).
- Provide a concise way to express instances of single-method interfaces using a block of code.
- Implement the only abstract function and therefore implement functional interfaces
- A lambda expression is essentially an anonymous function (a function without a name) that can be used to implement a method defined by a functional interface

Functionalities of lambda expressions

- Functional Interfaces: A functional interface is an interface that contains only one abstract method. Used to implement the abstract method
- Code as Data: Treat functionality as a method argument.
- Class Independence: Create functions without defining a class.
- Pass and Execute: Pass lambda expressions as objects and execute on demand.



```
(int arg1, String arg2) -> {System.out.println("Two arguments "+arg1+" and "+arg2);}
```

Structure of Lambda Expression

Syntax

- Java Lambda Expression has the following syntax:

(argument list) -> { body of the expression }

- Components:
- Argument List: Parameters for the lambda expression
- Arrow Token (->): Separates the parameter list and the body
- Body: Logic to be executed.

Lambda Expression Parameters

Zero Parameter

Single Parameter

Multiple Parameters

Lambda Expression with Zero parameter

- () -> System.out.println(“Zero parameter lambda”);

Lambda Expression with Single parameter

- (p) -> System.out.println(“One parameter: ” + p);

- Not mandatory to use parentheses if the type of that variable can be inferred from the context

```
// A Java program to demonstrate simple lambda expressions

import java.util.ArrayList;
class Test {
    public static void main(String args[])
    {
        // Creating an ArrayList with elements {1, 2, 3, 4}
        ArrayList<Integer> arrL = new ArrayList<Integer>();
        arrL.add(1);
        arrL.add(2);
        arrL.add(3);
        arrL.add(4);
        // Using lambda expression to print all elements of arrL
        System.out.println("Elements of the ArrayList : ");
        arrL.forEach(n -> System.out.println(n));
        // Using lambda expression to print even elements  of arrL
        System.out.println("Even elements of the ArrayList : ");
        arrL.forEach(n -> {
            if (n % 2 == 0)
                System.out.println(n);
        });
    }
}
```

Lambda Expression with Multiple parameters

- (p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

```
// Example to demonstrate lambda expressions with multiple parameters
@FunctionalInterface
interface Functional {
    int operation(int a, int b);
}

public class Test {

    public static void main(String[] args) {
        // Using lambda expressions to define the operations
        Functional add = (a, b) -> a + b;
        Functional multiply = (a, b) -> a * b;

        // Using the operations
        System.out.println(add.operation(6, 3));  // Output: 9
        System.out.println(multiply.operation(4, 5));  // Output: 20
    }
}
```

Lambdas and collections

```
import java.util.Arrays;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Using lambda to iterate through the list
        names.forEach(name -> System.out.println(name));

        // Using lambda to filter and print names starting with 'A'
        names.stream()
            .filter(name -> name.startsWith("A"))
            .forEach(name -> System.out.println("Filtered name: " + name));
    }
}
```

```
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String
name, float price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
```

```
public class LambdaExpressionExample{
    public static void main(String[] args) {
        List<Product> list=new ArrayList<Product>();

        //Adding Products
        list.add(new Product(1,"HP Laptop",25000f));
        list.add(new Product(3,"Keyboard",300f));
        list.add(new Product(2,"Dell Mouse",150f));

        System.out.println("Sorting on the basis of
name...");

        // implementing lambda expression
        Collections.sort(list,(p1,p2)->
p1.name.compareTo(p2.name);
        );
        for(Product p:list){
            System.out.println(p.id+" "+p.name+"
"+p.price);
        }
    }
}
```

Java Lambda Expression Example: Comparator

```
class Product{
   int id;
   String name;
   float price;
   public Product(int id, String
name, float price) {
      super();
      this.id = id;
      this.name = name;
      this.price = price;
   }
}
```

```
public class LambdaExpressionExample11{
   public static void main(String[] args) {
      List<Product> list=new ArrayList<Product>();
      list.add(new Product(1,"Samsung A5",17000f));
      list.add(new Product(3,"Iphone 6S",65000f));
      list.add(new Product(2,"Sony Xperia",25000f));
      list.add(new Product(4,"Nokia Lumia",15000f));
      list.add(new Product(5,"Redmi4 ",26000f));
      list.add(new Product(6,"Lenevo Vibe",19000f));

      // using lambda to filter data
      Stream<Product> filtered_data =
list.stream().filter(p -> p.price > 20000);

      // using lambda to iterate through collection
      filtered_data.forEach(
            product ->
System.out.println(product.name+": "+product.price)
      );
   }
}
```

Java Lambda Expression Example: Filtering

Why use Lambda Expression?

Help to implement functional interfaces more concisely and expressively,

Eliminate lengthy boilerplate code and strengthening readability.

Incorporates a functional programming style to Java and treats the functions as first-class citizens.

Lambda expressions uniquely allow to operate on stream collections in a functional style with a simplified syntax.

Flexibility is also apparent in cases where the behaviour should be containerized and as an argument so that it can be modularized and simple to maintain.

Lambda expressions help to provide a more compact alternative to anonymous classes and thus to simplify the code and reduce verbosity.

Foster the implementation of parallel programming constructs and, in general, give room for the elimination of bugs through coding practices that are neat and error-free.

Disadvantages of lambda expressions

Restricted to Functional Interfaces:

- Only functional interfaces, which contain a single abstract method, are compatible with lambda expressions.
- Limitation makes them only applicable in specific situations, which might be problematic if you have to interact with interfaces that include multiple abstract methods.

Disadvantages of lambda expressions

Readability Issues with Complex Reasoning:

- Lambda expressions are useful for short and straightforward processes, but when applied to more complicated reasoning, they may become less readable.
- In some situations, a different class or a more conventional approach might make sense.

Disadvantages of lambda expressions

Debugging:

- Lambda expressions can be difficult to debug, particularly if they are intricate or include several levels of abstraction.
- Stack traces might not always be able to pinpoint the exact location of a problem in a lambda expression.

Disadvantages of lambda expressions

State Management Difficulties:

- Lambda expressions lack a state and are stateless.
- Because of this, handling mutable states and capturing variables from enclosing scopes can be difficult, particularly when working with concurrent or parallel streams.

Disadvantages of lambda expressions

Performance Overhead in Specific Situations:

- Although lambda expressions are optimized for modern Java runtime systems, there may be instances in which using lambda expressions results in a minor performance overhead.
- While this usually doesn't matter for the majority of applications, it could be taken into account in situations where performance is crucial.

Things to avoid with lambdas

Avoid Specifying Parameter Types

- A compiler, in most cases, is able to resolve the type of lambda parameters with the help of type inference.
- Consequently, adding a type to the parameters is optional and can be omitted.
- can do this:

```
(a, b) -> a.toLowerCase() +
b.toLowerCase();
```

- Instead of this:

```
(String a, String b) -> a.toLowerCase() +
b.toLowerCase();
```

Things to avoid with lambdas

Avoid Parentheses Around a Single Parameter

- Lambda syntax only requires parentheses around more than one parameter, or when there is no parameter at all.
- That's why it's safe to make our code a little bit shorter, and to exclude parentheses when there is only one parameter.

- can do this:

```
a -> a.toLowerCase();
```

- Instead of this:

```
(a) -> a.toLowerCase();
```

Things to avoid with lambdas

Avoid Return Statement and Braces

- **Braces** and ***return*** statements are optional in one-line lambda bodies.
- Can be omitted for clarity and conciseness.
- can do this:

```
a -> a.toLowerCase();
```

- Instead of this:

```
a -> {return a.toLowerCase()};
```

Things to avoid with lambdas

Use Method References

- lambda expressions just call methods which are already implemented elsewhere.
- In this situation, it is very useful to use another Java 8 feature, **method references**.
- not always shorter, but it makes the code more readable.
- can do this:

```
a -> a.toLowerCase();
```

- Instead of this:

```
String::toLowerCase;
```

Things to avoid with lambdas

Use “Effectively Final” Variables

- Accessing a non-final variable inside lambda expressions will cause a compile-time error, **but that doesn’t mean that we should mark every target variable as *final*.**
- According to the “**effectively final**” concept, a compiler treats every variable as *final* as long as it is assigned only once.
- It’s safe to use such variables inside lambdas because the compiler will control their state and trigger a compile-time error immediately after any attempt to change them.
- **following code will not compile:**
- public void method() {

```
String localVariable = "Local";
Foo foo = parameter -> {
String localVariable = parameter;
return localVariable;
};
}
```

Kinds of Method References

Kind	Syntax	Examples
Reference to a static method	*ContainingClass::staticMethodName*	Person::compareByAge MethodReferencesExamples::appendStrings
Reference to an instance method of a particular object	*containingObject::instanceMethodName*	myComparisonProvider::compareByName myApp::appendStrings2
Reference to an instance method of an arbitrary object of a particular type	*ContainingType::methodName*	String::compareToIgnoreCase String::concat
Reference to a constructor	*ClassName*::new	HashSet::new

```
import java.util.function.BiFunction;
public class
MethodReferencesExamples {

    public static <T> T mergeThings(T a, T
b, BiFunction<T, T, T> merger) {
        return merger.apply(a, b);
    }

    public static String
appendStrings(String a, String b) {
return a + b;    }

    public String appendStrings2(String a,
String b) {
        return a + b;
    }
```

```
 public static void main(String[] args) {
        MethodReferencesExamples myApp = new
MethodReferencesExamples();
        // Calling the method mergeThings with a lambda
expression
        Sysem.out.println(MethodReferencesExamples.
            mergeThings("Hello ", "World!", (a, b) -> a + b));

        // Reference to a static method
        System.out.println(MethodReferencesExamples.
            mergeThings("Hello ", "World!",
MethodReferencesExamples::appendStrings));

        // Reference to an instance method of a particular
object
        System.out.println(MethodReferencesExamples.
            mergeThings("Hello ", "World!",
myApp::appendStrings2));

        // Reference to an instance method of an arbitrary
object of a
        // particular type
        System.out.println(MethodReferencesExamples.
            mergeThings("Hello ", "World!", String::concat)); }}
```

Reference to an Instance Method of a Particular Object

```
class ComparisonProvider {
   public int compareByName(Person a, Person b) {
      return a.getName().compareTo(b.getName());
   }

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

ComparisonProvider myComparisonProvider = new ComparisonProvider();
Arrays.sort(rosterAsArray, myComparisonProvider::compareByName);

- method reference myComparisonProvider::compareByName invokes the method compareByName that is part of the object myComparisonProvider.
- The JRE infers the method type arguments, which in this case are (Person, Person).

Reference to an Instance Method of an Arbitrary Object of a Particular Type

```
String[] stringArray = { "Barbara", "James", "Mary", "John", "Patricia", "Robert", "Michael", "Linda" };
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

- Equivalent lambda expression for the method reference String::compareToIgnoreCase would have the formal parameter list (String a, String b), where a and b are arbitrary names used to better describe this example.
- Method reference would invoke the method a.compareToIgnoreCase(b).
- Similarly, the method reference String::concat would invoke the method a.concat(b).