Java Lambda Expression

Anurak Theanpurmpul

Functional Programming

- **Function is a block of code that accepts values, known as parameters, and the block of code is executed to compute a result.**
- Functional programming is **declarative**.
- The core idea is that we can create and manipulate functions, including creating functions at runtime. Thus, f**unctions become another thing (instead of just data) that your programs can manipulate.**
- Functional programming has become popular because of its **suitability in concurrent, parallel, and event-driven programming.**
- Java 8 introduced **lambda expressions** to support functional programming.

Core Concepts of FP

- **First Class Function** - a function can be **passed as an argument** to other functions, can be **returned by another function** and can be **assigned as a value to a variable.**
- **Pure Function**
- **Function Composition -** the process of combining two or more functions to produce a new function (a series of pipes for our data to flow through).
- **Closure**
- **Avoid shared state**
- **Avoid mutating state**
- **Avoid side effects**

Pure Function

- A pure function is a function which:
  - **Given the same input, will always return the same output.**
  - **Produces no side effects.**
- Extremely **self-contained and independent of state** - easy to move around, refactor, and reorganize in your code.
- Good candidates for parallel processing across many CPU.
- Demo **function1.js**

Side Effect

- A **side effect** is any application state change that is observable outside the called function other than its return value.
- Examples:
  - Modifying any external variable or object property (e.g., a global variable, or a variable in the parent function scope chain)
  - Logging to the console
  - Writing to the screen
  - Writing to a file
  - Writing to the network
  - Triggering any external process
  - Calling any other functions with side-effects
- Side-effect actions need to be isolated from the rest of program logic

Side Effect - Examples

```
public Program getCurrentProgram(TVGuide guide, int channel) {
  Schedule schedule = guide.getSchedule(channel);
  Program current = schedule.programAt(new Date());
  return current;
}
```

```
public Program getProgramAt(TVGuide guide, int channel, Date when) {
  Schedule schedule = guide.getSchedule(channel);
  Program program = schedule.programAt(when);
  return program;
}
```

Immutable vs Mutable

- **Immutable** object is an object that **can’t be modified** after it’s created.
- **Mutable** object is any object which **can be modified** after it’s created.
- **Immutable** types are safer from bugs, easier to understand, and more ready for change.
- Memory usage and performance concerns.

Higher-Order Functions

- **Reusability**
- **First class functions** - treat functions as data, assign them to variables, pass them to other functions, return them from functions.
- **Higher order functions** is any function which takes a function as an argument, returns a function, **or both**.
- Utilities function like map(), filter(), reduce() and etc.
- Demo **function2.js**
- A lambda expression in Java is like a **higher-order function**
- Build-in java higher-order functions.
- Demo **HighOrderFunctionExample.java**

What is Closure?

- A closure is the **combination of a function** bundled together (enclosed) with references to its surrounding state (the lexical environment).
- A closure gives you access to an outer function's scope from an inner function.
- Closures are useful because they let you associate data (the lexical environment) with a function that operates on that data.
- Domo **function3.js**
- In Java, lambdas only close over values, but not variables.
- Java has "closures with restrictions," which might not be "perfect" closures, but are nonetheless still quite useful.
- Demo **AreLambdasClosures.java**, **AreLambdasClosures2.java**

Imperative Programming vs Declarative Programming Example

- Filtering employees with high salaries, **previous to Java 8**

```
public List<Employee> employeeWithHighSalaries(double salary) {
   List<Employee> res = new ArrayList<>();
   for(Employee e: employees) {
      if(e.getSalary() > salary) res.add(e);
   }
   return res;
}
```

How it should be performed

- Filtering employees with high salaries, **using lambda expressions and the Java 8 stream API.**

```
public List<Employee> employeeWithHighSalaries(double salary) {
   return employees.stream()
                   .filter(e →e.getSalary() > salary)
                   .collect(Collectors.toList());
}
```

what it should be performed

Why Does Java Need Function?

- Data volume
- Hardware Specification
- Concurrent and Parallel processing
- Mutual object side effects

What is Lambda Expression?

- “Lambda” has its origin in Lambda calculus that uses the Greek letter lambda (**λ**) to denote a function abstraction.
- A lambda expression is an **unnamed** block of code (or an unnamed function) with a list of formal **parameters** and a **body**.
- **Java introduced lambda expressions to support functional programming**, which can be mixed with its already popular object-oriented features to develop robust, concurrent, parallel programs.
- Java is a **strongly-typed language**, but lambda expression **by itself** does not have a type.
- A lambda expression is **an expression that represents an instance of a functional interface.**
- **A lambda expressions can be used only with a functional interface.**

Syntax of Lambda Expression

- Syntax:
  - **(<LambdaParametersList>) -> { <LambdaBody> }**
  - The list of **parameters** is enclosed in parentheses, as is done for methods.
  - The **body** of a lambda expression can be **a block statement or an expression.**
  - An **arrow (->)** is used to separate the list of parameters and the body.
- What is the type of this lambda expression?
  - (x, y) -> x + y;
  - **Demo LambdaTypeExample.java**
- The type of a lambda expression is always inferred by the compiler by the **context** in which it is used.
- We can consider **lambda expressions as anonymous methods**—methods without a name.
- Demo
  - **NotAnonymousExample.java**
  - **AnonymousExample.java**
  - **AnonymousToLambdaExample.java**

Functional Interfaces

- Functional interfaces provide target types for lambda expressions and method references.
- A functional interface has exactly **one abstract method**.
- **Default methods**, **static methods**, **public methods inherited from the Object class** (ex. equals(Object obj) in Comparator) **are exceptions.**
- Java calls the method a "functional method" but the name "single abstract method" or SAM is often used.
- A new annotation **@FunctionalInterface,** this annotation gives you an additional assurance from the compiler.
- Since jdk 8, all the existing single method interfaces like Runnable and Callable in the JDK are now functional interfaces.
- Any place a functional interface is used, we can now use a lambda.
- Java treats lambdas as an instance of an interface type.

Functional Interfaces Examples

- Runnable interface class in JDK 8.

```
1   package java.lang;
2
3
4   @FunctionalInterface
5   public interface Runnable {
6
7       public abstract void run();
8   }
9
```

- https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html
- https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html
- Demo **FirstLambdaExample.java**
- Another example are Predicate, Comparator class and etc.

Target typing

- The compiler infers the type of a lambda expression.
- **Target type** - The context in which a lambda expression is used expects a type.
- **Target typing** - The process of inferring the type of a lambda expression from the context.
- **T t = <LambdaExpression>;**
  - The target type of the lambda expression in this context is T.
  - T must be a functional interface type.
  - The lambda expression has the same number and type of parameters as the abstract method of T.
  - The type of the returned value is assignment compatible to the return type of the abstract method of `T`.
  - The exceptions must be compatible with the declared `throws` clause of the abstract method of `T`.

Target typing Example

```java
@FunctionalInterface
interface Calculation {
    Integer apply(Integer x, Integer y);
}
```

```java
static Integer calculate(Calculation operation, Integer x, Integer y) {
    return operation.apply(x, y);
}
```

```java
static Calculation add = (x, y) -> x + y;
static Calculation subtraction = (x, y) -> x - y;
```

- Functional interface (*Calculation class*) describes the types, it gives the compiler all the information it needs.
- Use the method signature to infer the types of the lambda's parameters. There's only one method on the interface, so there's no ambiguity
- Demo **Calculator.java and ThrowsExceptionExample.java**

Context Lambda

- The idea of context is absolutely essential to creating a lambda expression object.
- It have three primary places where context typically comes from.
  - **Assigning the lambda to a variable** (ex. Calculator.java).
  - **Assigning the lambda as a method parameter** (ex. HighOrderFunctionExample.java and Calculator.java).
  - **Assigning the lambda to the return value of a method** (ex. HighOrderFunctionExample.java)
- And the final form, the one that is significantly less common but completely legitimate, is to **use a cast to specify what type of lambda we're trying to build.**

Explicit Lambda Expression

- In some contexts, there is no way the compiler can infer the type of a lambda expression, one such case is **passing lambda expressions to overloaded methods**.
- We can help the compiler by providing some more information.
  - make it explicit by specifying the type of the parameters.
  - Use a cast.
  - Assign lambda expression to a variable of the desired type, and then, pass the variable to the method.
- Demo **ExplicitLambdaExample.java**

Recap - Lambda Expression Contexts

- Lambda expressions can be used only in the following contexts:
  - **Assignment Context**: A lambda expression may appear to the right-hand side of the assignment operator in an assignment statement.
  - **Method Invocation Context**: A lambda expression may appear as an argument to a method or constructor call.
  - **Return Context**: A lambda expression may appear in a return statement inside a method, as its target type is the declared return type of the method.
  - **Cast expressions:** A lambda expression may be used if it is preceded by a cast. The type specified in the cast is its target type.

Variable Capture

- A lambda expression can access effectively final local variables.
- A local variable is effectively final in the following two cases:
  - It is **declared final**.
  - It is **not declared final**, but initialized only once.
- A lambda expression can modify a class and instance variable if they are not final.
- Demo **VariableExample.java**
- A lambda expression **can access instance and class variables of a class** whether they are effectively final or not(can modify them).
- Demo **VariableClassInstanceExample.java**

java.util.function

- Java SE 8 has added a lot of built-in functional interfaces in the **package java.util.function**.
- Can be used by the developer in lambda expressions instead of creating their own.
- https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Commonly Used Functional Interfaces

Interface	Function Descriptor	Description
Consumer<T>	T -> void	Represents an operation that accepts a single input argument and returns no result.
Function<T, R>	T -> R	Represents a function that accepts one argument and produces a result.
Predicate<T>	T -> boolean	Represents a predicate (boolean-valued function) of one argument.
Supplier<T>	() -> T	Represents a supplier of results.
UnaryOperator<T>	T -> T	Represents an operation on a single operand that produces a result of the same type as its operand.

Interface	Function Descriptor	Description
BiConsumer<T, U>	(T, U) -> void	Represents an operation that accepts two input arguments and returns no result.
BiFunction<T, U, R>	(T, U) -> R	Represents a function that accepts two arguments and produces a result.
BiPredicate<T, U>	(T, U) -> boolean	Represents a predicate (boolean-valued function) of two arguments.
BinaryOperator<T>	(T, T) -> T	Represents an operation upon two operands of the same type, producing a result of the same type as the operands.

Primitive Specializations

1. Predicate<T> Primitive Specializations
2. Consumer<T> Primitive Specializations
3. Function<T, R> Primitive Specializations
    a. Consuming Primitive Specializations
    b. Producing Primitive Specializations
    c. Consuming and Producing Primitive Specializations
4. Supplier<T> Primitive Specializations
5. UnaryOperator<T> Primitive Specializations
6. BinaryOperator<T> Primitive Specializations
7. BiConsumer<T, U> Primitive Specializations
8. BiFunction<T, U, R> Primitive Specializations

https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Function interface

- Represents a function that accepts one argument and produces a result.

Function Name	Function Method
Function<T,R>	R **apply**(T t) default <V> Function<V,R> **compose**(Function<? super V,? extends T> before) default <V> Function<T,V> **andThen**(Function<? super R,? extends V> after) static <T> Function<T,T> **identity**()
DoubleFunction<R>	R apply(double value)
DoubleToIntFunction	int applyAsInt(double value)
DoubleToLongFunction	long applyAsLong(double value)
IntFunction<R>	R apply(int value)
IntToDoubleFunction	double applyAsDouble(int value)
IntToLongFunction	long applyAsLong(int value)
LongFunction<R>	R apply(long value)
LongToDoubleFunction	double applyAsDouble(long value)

Function Interface (con’t)

Function Name	Function Method
LongFunction<R>	R apply(long value)
LongToDoubleFunction	double applyAsDouble(long value)
LongToIntFunction	int applyAsInt(long value)
ToDoubleFunction<T>	double applyAsDouble(T value)
ToIntFunction<T>	int applyAsInt(T value)
ToLongFunction<T>	long applyAsLong(T value)

- Demo **UtilFunctionExample.java**

Consumer Interface

- Represents **an operation that accepts a single input argument and returns no result**. Unlike most other functional interfaces, **Consumer is expected to operate via side-effects.**

Function Name	Function Method
Consumer<T>	void **accept**(T t) default Consumer<T> **andThen**(Consumer<? super T> after)
DoubleConsumer	void **accept**(double value) default DoubleConsumer **andThen**(DoubleConsumer after)
IntConsumer	void **accept**(int value) default IntConsumer **andThen**(IntConsumer after)
LongConsumer	void **accept**(long value) default LongConsumer **andThen**(LongConsumer after)
ObjDoubleConsumer<T>	void **accept**(T t, double value)
ObjIntConsumer<T>	void **accept**(T t, int value)
ObjLongConsumer<T>	void **accept**(T t, long value)

- Demo **UtilConsumerExample.java**

Predicate interface

- Represents a condition that **returns true or false for the specified argument.**

Function Name	Function Method
Predicate<T>	boolean **test**(T t) default Predicate<T> **and**(Predicate<? super T> other) default Predicate<T> **negate**() default Predicate<T> **or**(Predicate<? super T> other) static <T> Predicate<T> **isEqual**(Object targetRef)
DoublePredicate	boolean test(double value) default DoublePredicate **and**(DoublePredicate other) default DoublePredicate **negate**() default DoublePredicate **or**(DoublePredicate other)
IntPredicate	boolean **test**(int value) default IntPredicate **and**(IntPredicate other) default IntPredicate **negate**() default IntPredicate **or**(IntPredicate other)
LongPredicate	boolean **test**(long value) default LongPredicate **and**(LongPredicate other) default LongPredicate **negate**() default LongPredicate **or**(LongPredicate other)

UnaryOperator Interface

- Represents an operation on a single operand that **produces a result of the same type as its operand.**
- It a **specialization of Function** (extends from) for the case where the operand and result are of the same type.

Function Name	Function Method
UnaryOperator<T> extends Function<T,T>	R apply(T t) default <V> Function<V,R> compose(Function<? super V,? extends T> before) default <V> Function<T,V> andThen(Function<? super R,? extends V> after) static <T> Function<T,T> identity() static <T> UnaryOperator<T> identity()
DoubleUnaryOperator	double applyAsDouble(double operand) default DoubleUnaryOperator compose(DoubleUnaryOperator before) default DoubleUnaryOperator andThen(DoubleUnaryOperator after) static DoubleUnaryOperator identity()
IntUnaryOperator	int applyAsInt(int operand) default IntUnaryOperator compose(IntUnaryOperator before) default IntUnaryOperator andThen(IntUnaryOperator after) static IntUnaryOperator identity()
LongUnaryOperator	long applyAsLong(long operand) default LongUnaryOperator compose(LongUnaryOperator before) default LongUnaryOperator andThen(LongUnaryOperator after) static LongUnaryOperator identity()

- Demo **UtilUnaryOperatorExample.java**

Binary Interface

- Represents a **function that accepts two arguments and produces a result.** This is the two-arity specialization of **Function**.

Function Name	Function Method
BiFunction<T,U,R>	R apply(T t,U u) default <V> BiFunction<T,U,V> andThen(Function<? super R,? extends V> after)
BiConsumer<T,U>	void accept(T t,U u) default BiConsumer<T,U> andThen(BiConsumer<? super T,? super U> after)
BiPredicate<T,U>	boolean test(T t, U u) default BiPredicate<T,U> and(BiPredicate<? super T,? super U> other) default BiPredicate<T,U> negate() default BiPredicate<T,U> or(BiPredicate<? super T,? super U> other)
BinaryOperator<T> extends BiFunction<T,T,T>	static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator) static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)
DoubleBinaryOperator	double applyAsDouble(double left, double right)

Binary Interface (con’t)

Function Name	Function Method
IntBinaryOperator	int **applyAsInt**(int left,int right)
LongBinaryOperator	long **applyAsLong**(long left,long right)
ToDoubleBiFunction<T,U>	double **applyAsDouble**(T t,U u)
ToIntBiFunction<T,U>	int **applyAsInt**(T t,U u)
ToLongBiFunction<T,U>	long **applyAsLong**(T t, U u)

Intersection Type

- Java 8 introduced a new type called **an intersection type**
- An intersection type is an intersection of multiple types.
- An intersection type may appear as the target type in a cast.
- An **ampersand(&)** is used between two types as **Type1 & Type2** to represent a new type that is an intersection of Type1, Type2.
- Demo **IntersectionTypeExample.java**

Method References

- A method reference is shorthand to create a lambda expression using an existing method.
- Uses in case of your lambda expression does nothing but call an existing method.
- Makes your lambda expressions more readable and concise.
- Syntax:
  - **<Qualifier>::<MethodName>**
  - The **<Qualifier>** depends on the type of the method reference.
  - **Two consecutive colons** act as a separator.
  - The **<MethodName>** is the name of the method.

Kinds of Method References

	Syntax	Description
Static Method References	ContainingClass::staticMethodName	Reference to a static method
Instance Method References - Bound receiver	containingObject::instanceMethodName	Reference to an instance method of a particular object
Instance Method References - Unbound receiver	ContainingType::methodName	Reference to an instance method of an arbitrary object of a particular type
Constructor References	ClassName::new	Reference to a constructor

Static Method References

- A static method reference is used to use a static method of a type as a lambda expression. The type could be a class, an interface, or an enum.
- Demo **MethodReferenceExample.java.**

Supplier Interface

- Represents a function that **takes no argument and returns a result of type T.**

Function Name	Function Method
Supplier<T>	T **get**()
BooleanSupplier	boolean **getAsBoolean**()
DoubleSupplier	double **getAsDouble**()
IntSupplier	int **getAsInt**()
LongSupplier	long **getAsLong**()

- Demo **UtilSupplierExample.java**

Instance Method References

- An instance method is invoked on an object's reference.
- **Bound receiver** - specify the receiver of the method invocation explicitly.
- **Unbound receiver** - provide the receiver implicitly when the method is invoked
- The syntax for an instance method reference supports two variants:
  - **objectRef::instanceMethod**
  - **ClassName::instanceMethod**
- **Bound Receiver** - use the **objectRef.instanceMethod** syntax.
- **Unbound Receiver** - use the **ClassName::instanceMethod** syntax
- Rule - The first argument to the function represented by the **target type** is the receiver of the method invocation.
- Demo **MethodReferenceExample.java.**

Constructor References

- Use when the body of a lambda expression just an object creation expression.
- The syntax for using a constructor is
  - **ClassName::new**
  - **`ArrayTypeName::new`**
- Demo MethodReferenceConstructorExample.java

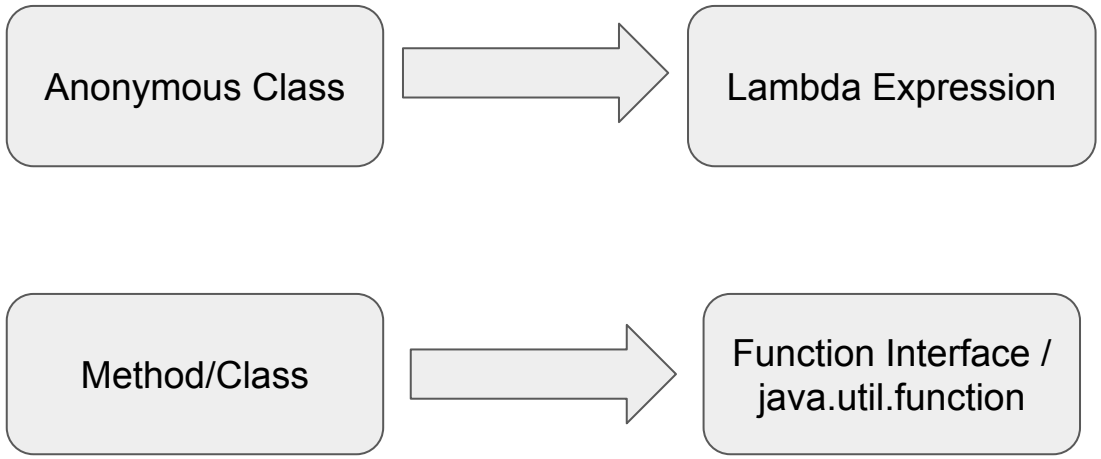
“this” and “super”

- The keywords **this** and **super** are the same inside the lambda expression and its enclosing method.
- That different from scope of the local and anonymous class.
- Variable names in a scope must be unique.
- Demo **ScopeExample.java**

Recursive Lambda Expressions

- A lambda expression **does not support recursive** invocations.
- **Use a method reference or an anonymous inner class**.
- Demo **RecursiveWithMethodRefExample.java**
- Demo **RecursiveWithAnonymousExample.java**

Recap


Anonymous Class
Lambda Expression
Method/Class
Function Interface /
java.util.function