**What** is **Lambda** Expression?

A L**ambda Expression** is an anonymous function that lets you express instances of single-method classes more compactly and thus enabling you to treat functionality as method argument, or code as data.

**Why** **Lambda** Expression?

-   Implementation of anonymous class may seem **unwieldy** and unclear for an interface that contains only one method.

- Compiler generates a **new class** file for each anonymous inner class, thus

-- Each class file needs to be **loaded** and **verified** before being used

-- Where loading incurs expensive operations, such as disk I/O and JAR decompression.

**What** is **Functional Interface**?

A **Functional Interface**

- is an interface with exactly **one abstract** method.

- It may contain **multiple** **default** or **static** methods.

- Abstract methods **overriding** public methods of “java.lang.**Object**” does not count toward the interface's abstract method count.

- Instances of functional interfaces can be created with lambda expressions, method references, or constructor references.

- May be annotated with “@FunctionalInterface” annotation.

- JDK defines several **standard** functional interfaces, which you can find in the package “java.util.function”.

λ expressions **syntax**

[**Lambda Expression syntax**](lambda%20syntax.docx)

**new** Predicate<Person>(){

**public** **boolean** test(Person p){

**return** p.getGender() == Person.Sex.***MALE***

&& p.getAge() >= 18

&& p.getAge() <= 25;

}

};

**Anonymous Inner Class**

Lambda expression ***syntax*** consists of the following:

- Comma-separated list of formal parameters enclosed in parentheses •

- You can omit the parentheses if there is only one parameter •

- You can omit the data type of the parameters •

- The arrow token, -> •

(**Person** p**)** **->**{

- Body, which consists of a single **return** p.getGender() == Person.Sex.***MALE***

expression or a statement block • && p.getAge() >= 18

&& p.getAge() <= 25;

**}**;

**λ Expression**

- You must enclose statements in braces ({})

- For expressions braces ({}) can be omitted

- Exceptionally you do not have to enclose a void method invocation in braces

- If you specify a single expression, then the JRE evaluates the expression and then returns its value.

**Use case** scenario for **λ** expressions

<Person.java>

<FilterUtil.java>

Points to **ponder** about lambdas

- Like local and anonymous classes, lambda expressions can ***capture*** ***variables***.

- Lambda expressions are ***lexically scoped***, thus eliminating ***shadowing issues***

- Lambda expressions can only be used in situation where complier can determine the ***target type*** (target type denotes functional interface whose instance this lambda represents) for them.

- Lambda expressions can be ***serialized*** if its target type and its captured arguments are serializable.

- Lambda expressions are not simply syntactic sugar for anonymous classes, their internal implementation uses ***invokedynamic*** byte code.

**What** are **method references**?

- Sometimes, however, a λ expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this.

- Through method references we specify the name of method which is to be invoked in λ expression’s body.

[**Method references syntax**](method%20refrences%20syntax.docx)

**Anonymous Inner Class implementation**

Function<String, Integer> f = **new** Function<String, Integer>()

{

@Override

**public** Integer apply(String arg0)

{

**return** Integer.*parseInt*(arg0);

}

};

**Corresponding λ Expression** (**standard syntax) implementation**

Function<String, Integer> f = (String number) ->

{

**return** Integer.*parseInt*(number);

};

**Corresponding λ Expression** (**minimal syntax) implementation**

Function<String, Integer> f = number -> Integer.*parseInt*(number);

**Corresponding method reference implementation**

Function<String, Integer> f = Integer::*parseInt*;

**Method reference logical internal implementation**

Function<String, Integer> f = (String number) ->

{

**Return**  Integer::*parseInt* (number);

};

- behind the curtain logical implementation (not actual, for actual implementation refer de- compilation for method references )

**-** There are four kinds of method references

|  |  |
| --- | --- |
| **Kind** | **Example** |
| Reference to a static method | ContainingClass::staticMethodName |
| Reference to an instance method of a particular object | containingObject::instanceMethodName |
| Reference to an instance method of an arbitrary object of a particular type | ContainingType::methodName |
| Reference to a constructor | ClassName::new |

**Use case** scenario for **method references**.

<MethodReferencesTest.java>

[**λ** Expressions **internal implementation**](Lambda%20Byte%20Code%20Implementation.docx)

**Lambda Byte Code Implementation**

**Prerequisites carry off excerpt**

## *Linking* is the process of taking a class or interface and combining it into the run-time state of the Java Virtual Machine so that it can be executed

## *LambdaMetafactory* contains methods to facilitate the creation of simple "function objects" that implement one or more interfaces by delegation to a provided [MethodHandle](https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html" \o "class in java.lang.invoke).

## Linkage occurs when the methods in LambdaMetafactory class are invoked.

## Linkage produces a *[CallSite](https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/CallSite.html" \o "class in java.lang.invoke)* whose target can be used to create suitable function objects.

## Linkage may involve dynamically loading a new class that implements the target interface.

## The *CallSite* can be considered a "factory" for function objects and so these linkage methods are referred to as "metafactories".

## The VM will call the LambdaMetafactory only once per *capture site*; thereafter it will link the call site and get out of the way.

## *Capture site* is place where lambda is defined e.g.

## Function<String, Integer> f = s -> Integer.*parseInt*(s);

## Call sites are linked lazily, so factory sites (*capture site*) that are never invoked are never linked.

## Capture occurs when the CallSite's target is invoked, typically through an invokedynamic call site, producing a function object. This may occur many times for a single factory CallSite. Capture may involve allocation of a new function object, or may return an existing function object.

## Invocation occurs when an implemented interface method is invoked on a function object. This may occur many times for a single function object. The method referenced by the *behavior MethodHandle* is invoked with the captured arguments and any additional arguments provided on invocation

## *Behavior method*/ *implMethod* - a direct method handle describing the implementation method which should be called at invocation time, is a DE sugared representation of lambda body

## *samMethodType* a direct method handle describing signature and return type of method to be implemented by the function object

**Anonymous class sample excerpt**

**import** java.util.function.Function;

**public** **class** AnonymousClassExample {

Function<String,Integer> format = **new** Function<String,Integer>() {

**public** Integer apply(String input){

**return** Integer.*parseInt*(input);

}

};

}

**Correspondent byte code implementation for Anonymous class**

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: aload\_0

5: **new** #2 // class AnonymousClassExample$1

8: dup

9: aload\_0

10: invokespecial #3 // Method AnonymousClass$1."<init>":(LAnonymousClassExample;)V

13: putfield #4 // Field format:Ljava/util/function/Function;

16: **return**

**This code shows the following:**

* 5: An object of type AnonymousClassExample$1 is instantiated using the byte code operation new. A reference to the newly created object is pushed on the stack at the same time.
* 8: The operation dup duplicates that reference on the stack.
* 10: reference of recently pushed instance is popped and instance is initialized by a call to its constructor/"<init>"
* 13: another reference copy of same instance is popped from stack so as to assign this instance to ORV (object reference variable) format.

**Lambda expression sample excerpt**

**import** java.util.function.Function;

**public** **class** Lambda {

Function<String, Integer> f = s -> Integer.*parseInt*(s);

}

**Correspondent byte code implementation for Lambda expression**

0: aload\_0

1: invokespecial #1 // Method java/lang/Object."<init>":()V

4: aload\_0

5: invokedynamic #2, 0 // InvokeDynamic Ljava/util/function/Function;

10: putfield #3 // Field f:Ljava/util/function/Function;

13: **return**

## Note that byte code instruction invokedynamic is used this time instead of new to create instance of functional interface.

## Consider the continuous intensity levels for a moment as to hereby perceive lambda translation.

## As following being the code excerpt illustrating non capturing and capturing lambda expressions respectively.

**Lambda expression sample excerpt**

**import** java.util.function.Function;

**public** **class** Lambda{

// non capturing lambda

Function<String, Integer> f = s -> Integer.*parseInt*(s);

**int** offset = 100;

// capturing lambda

Function<String, Integer> fc = s -> Integer.*parseInt*(s) + offset;

## }

## Activity sequence followed at compile time is:

## Defining the *implementation method* (*behavior method*) which should be called (with suitable adaptation of argument types, return types, and with captured arguments prepended to the invocation arguments) at invocation time.

## - *implementation method* is DE sugared representation of lambda body.

## - e.g. for code excerpt implementation methods generated would be

**import** java.util.function.Function;

**public** **class** Lambda{

// non capturing lambda

Function<String, Integer> f = s -> Integer.*parseInt*(s);

**static** Integer lambda$1(String s){

**return** Integer.*parseInt*(s);

}

**int** offset = 100;

// capturing lambda

Function<String, Integer> fc = s -> Integer.*parseInt*(s) + offset;

**static** Integer lambda$2(**int** offset, String s){

**return** Integer.*parseInt*(s) + offset;

}

}

1. Replacing lambda at ***capture site*** with invokedynamic call, for which pseudo code representation looks like

## - *capture site* is code excerpt defining lambda e.g.

## Function<String, Integer> f = s -> Integer.*parseInt*(s);

**import** java.util.function.Function;

**public** **class** Lambda{

// non capturing lambda

Function<String, Integer> f = indy((MH(metaFactory), MH(invokeVirtual Function.apply),

MH(invokeStatic Lambda.lambda$1) ( ));

**static** Integer lambda$1(String s){

**return** Integer.*parseInt*(s); ***implMethod*** ***samMethodType***

***captured arguments***

***}***

**int** offset = 100;

// capturing lambda

Function<String, Integer> fc = indy((MH(metaFactory), MH(invokeVirtual Function.apply),

MH(invokeStatic Lambda.lambda$2))( offset )));

**static** Integer lambda$2(**int** offset, String s){

**return** Integer.*parseInt*(s) + offset;

}

}

## Activity sequence followed at runtime time is:

## On first invocation linkage occurs (lazy linkage); if no invocation is present linkage will not happen.

## - due to invokedynamic *translation strategy* is postponed till runtime.

## - *translation strategy* defines how lambda is converted to corresponding functional interface instance at runtime.

## 1.2 On rest of the invocations no linkage is needed.

## 2. On every invocation capture occurs to instantiate functional interface.

## 3. And finally invocation occurs over instance produced through call site during capture.

## 

**Performance analysis**

* + The **first part** in λ is the **linkage** step whereas in anonymous class it class **loading**.
* It takes time to warm up(linkage process) the lambda factory approach, during which it is **initially slower**
* Performance comes **into line** with class loading when there are enough call sites linked,  if the code is on a **hot path**
* If it’s a **cold path** the lambda factory approach can be up to 100x **faster**.
* When **calling** actual method both anonymous inner classes and lambda expressions perform the exact same operation so there is **no difference** in performance.
* Performance for **non-capturing** lambda expressions is **far ahead** of the hoisted anonymous inner class equivalent.
* For **capturing lambdas** it takes **same** performance **hit** as in anonymous inner classes, which should be managed if possible by hoisted implementations.

**References**

[**https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html**](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)

[**http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html**](http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html)

[**https://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood**](https://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood)

[**https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/LambdaMetafactory.html#metafactory-java.lang.invoke.MethodHandles.Lookup-java.lang.String-java.lang.invoke.MethodType-java.lang.invoke.MethodType-java.lang.invoke.MethodHandle-java.lang.invoke.MethodType-**](https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/LambdaMetafactory.html#metafactory-java.lang.invoke.MethodHandles.Lookup-java.lang.String-java.lang.invoke.MethodType-java.lang.invoke.MethodType-java.lang.invoke.MethodHandle-java.lang.invoke.MethodType-)

[**https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html**](https://docs.oracle.com/javase/8/docs/api/java/lang/invoke/MethodHandle.html)

[**https://dzone.com/articles/why-we-need-lambda-expressions**](https://dzone.com/articles/why-we-need-lambda-expressions)

**###Person.java**

import java.time.LocalDate;

import java.time.chrono.IsoChronology;

import java.util.ArrayList;

import java.util.List;

public class Person

{

public enum Sex

{

MALE, FEMALE

}

String name;

LocalDate birthday;

Sex gender;

String emailAddress;

Person(String nameArg, LocalDate birthdayArg, Sex genderArg, String emailArg)

{

name = nameArg;

birthday = birthdayArg;

gender = genderArg;

emailAddress = emailArg;

}

public int getAge()

{

return birthday.until(IsoChronology.INSTANCE.dateNow()).getYears();

}

public void printPerson()

{

System.out.println(name + ", " + this.getAge());

}

public Sex getGender()

{

return gender;

}

public String getName()

{

return name;

}

public String getEmailAddress()

{

return emailAddress;

}

public LocalDate getBirthday()

{

return birthday;

}

public static int compareByAge(Person a, Person b)

{

return a.birthday.compareTo(b.birthday);

}

public static List<Person> createRoster()

{

List<Person> roster = new ArrayList<>();

roster.add(new Person("Fred", IsoChronology.INSTANCE.date(1980, 6, 20), Person.Sex.MALE, "fred@example.com"));

roster.add(new Person("Jane", IsoChronology.INSTANCE.date(1990, 7, 15), Person.Sex.FEMALE, "jane@example.com"));

roster.add(

new Person("George", IsoChronology.INSTANCE.date(1991, 8, 13), Person.Sex.MALE, "george@example.com"));

roster.add(new Person("Bob", IsoChronology.INSTANCE.date(2000, 9, 12), Person.Sex.MALE, "bob@example.com"));

return roster;

}

}

**###FilterUtil.java**

import java.util.List;

import java.util.function.Consumer;

import java.util.function.Predicate;

/\*\*

\*

\* @author kuldeepsin

\*

\*/

public class FilterUtil

{

/\*\*

\* 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 Methods That Search for Members That Match Specified

\* Search Criteria

\*/

public static void printPersons(List<Person> roster, Predicate<Person> tester)

{

for (Person p : roster)

{

if (tester.test(p))

{

p.printPerson();

}

}

}

/\*\*

\* Approach 3: Create Methods That Search for Members That Match Specified

\* Search Criteria and take specified action over them

\*/

public static void processPersons(List<Person> roster, Predicate<Person> tester, Consumer<Person> block)

{

for (Person p : roster)

{

if (tester.test(p))

{

block.accept(p);

}

}

}

public static void main(String... a)

{

List<Person> roster = Person.createRoster();

// Approach 2#A: Specify Search Criteria Code in an Anonymous Class

printPersons(roster, new Predicate<Person>()

{

public boolean test(Person p)

{

return p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25;

}

});

// Approach 2#B: Specify Search Criteria Code with a Lambda Expression

{

// Syntax Approach 2#B.0: standard approach

printPersons(roster, (Person p) ->

{

return p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25;

});

// Syntax Approach 2#B.1: you can omit the data type of the

// parameters

printPersons(roster, (p) ->

{

return p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25;

});

// Syntax Approach 2#B.2: omit the parentheses() as there is only

// one parameter

printPersons(roster, p ->

{

return p.getGender() == Person.Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25;

});

// Syntax Approach 2#B.3: return statement can be omitted because If

// you specify a single expression, then the JRE evaluates the

// expression and then returns its value

// and for expressions braces ({}) can be omitted too.

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

}

// Approach 3#A:Use Lambda Expressions Throughout Your Application,

// Specify Search Criteria and Action Code with a Lambda Expression

{

// Syntax Approach 3#A.0:

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

{

p.printPerson();

});

// Syntax Approach 3#A.0: because a single method call can be

// written as expression, parenthesis{} can be omitted.

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

p -> p.printPerson());

}

}

}

**MethodReferencesTest.java**

import java.util.List;

import java.util.Arrays;

import java.util.Comparator;

import java.util.Collection;

import java.util.function.Supplier;

import java.util.Set;

import java.util.HashSet;

/\*\*

\*

\* @author kuldeepsin

\*

\*/

public class MethodReferencesTest

{

// The method transferElements copies elements from one collection to

// another

public static <T, SOURCE extends Collection<T>, DEST extends Collection<T>> DEST transferElements(

SOURCE sourceCollection, Supplier<DEST> collectionFactory)

{

DEST result = collectionFactory.get();

for (T t : sourceCollection)

{

result.add(t);

}

return result;

}

public static void main(String... args)

{

List<Person> roster = Person.createRoster();

for (Person p : roster)

{

p.printPerson();

}

Person[] rosterAsArray = roster.toArray(new Person[roster.size()]);

class PersonAgeComparator implements Comparator<Person>

{

public int compare(Person a, Person b)

{

return a.getBirthday().compareTo(b.getBirthday());

}

}

// Without method reference, approach 1

Arrays.sort(rosterAsArray, new PersonAgeComparator());

// With lambda expression

Arrays.sort(rosterAsArray, (Person a, Person b) ->

{

return a.getBirthday().compareTo(b.getBirthday());

});

// With lambda expression, approach 2

Arrays.sort(rosterAsArray, (Person a, Person b) ->

{

return Person.compareByAge(a, b);

});

// With method reference

Arrays.sort(rosterAsArray, Person::compareByAge);

// 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);

// 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);

// Reference to a constructor

Set<Person> rosterSetLambda = transferElements(roster, () ->

{

return new HashSet<>();

});

Set<Person> rosterSet = transferElements(roster, HashSet::new);

System.out.println("Printing rosterSet:");

rosterSet.stream().forEach(p -> p.printPerson());

}

}

**###Functional.java**

/\*\*

\* although having multiple abstract methods but still its a functional

\* interface because methods overriding public methods of “java.lang.Object”

\* does not count toward the interface's abstract method count.

\*

\* @author kuldeepsin

\*

\*/

@FunctionalInterface

public interface Functional

{

abstract void someMethod();

abstract String toString();

abstract boolean equals(Object other);

}

**### LambdaSerialization.java**

import java.io.Serializable;

/\*\*

\* just in case you’d like to send it(lambda expression) over wire and execute

\* it somewhere else, as we do in RMI

\*

\* @author kuldeepsin

\*

\*/

public class LambdaSerialization

{

public static void main(String[] args)

{

// Lambda is Runnable but not Serializable

execution(() -> System.out.println("Hey, I am Lambda from Vanaheimr"));

// Lambda is Serializable now but not Runnable

//execution((Serializable) (() -> System.out.println("Hey, I am Lambda from Vanaheimr")));

// #Approach 1

execution((Runnable & Serializable) (() -> System.out.println("Hey, I am Lambda from Vanaheimr")));

// #Approach 2

remoteExecution(() -> System.out.println("Hey, I am Lambda from Asgard"));

}

public static void execution(Runnable t)

{

// send t over wire

// lets check before sending whether its Serializable or not

if (t instanceof Serializable)

{

// send(t,IP address)

System.out.println("Lambda sent to Niflheim");

} else

{

System.err.println("Lambda is not Serializable");

return;

}

// execute t over there

// lets check before executing whether its Runnable or not

if (t instanceof Runnable)

{

t.run();

} else

{

System.err.println("Lambda is not Runnable");

return;

}

}

public static <T extends Runnable & Serializable> void remoteExecution(T t)

{

// send t over wire

// lets check before sending whether its Serializable or not

if (t instanceof Serializable)

{

// send(t,IP address)

System.out.println("Lambda sent to Midgard");

} else

{

System.err.println("Lambda is not Serializable");

return;

}

// execute t over there

// lets check before executing whether its Runnable or not

if (t instanceof Runnable)

{

t.run();

} else

{

System.err.println("Lambda is not Runnable");

return;

}

}

}

**### MemoryProfilingScenario.java**

public class MemoryProfilingScenario

{

public static void main(String... a)

{

System.out.println(MemoryProfilingScenario.class);

fun();

// fun1();

// fun2();

}

// no memory usage issues as call site returns same instance of Runnable on

// each call to nonCapturingLambdaFactory

static void fun()

{

SomeClass object = new SomeClass();

while (true)

object.nonCapturingLambdaFactory().run();

}

// high memory usage issues because call site generates new instance of

// Runnable on each call to capturingLambdaFactory so as to hold latest copy

// captured ORV heavyStateObject

static void fun1()

{

SomeClass object = new SomeClass();

while (true)

object.capturingLambdaFactory().run();

}

// solution to high memory usage issue in case of capturing lambdas with

// help of hoisted instance

static void fun2()

{

SomeClass object = new SomeClass();

Runnable lambda = object.capturingLambdaFactory();

while (true)

lambda.run();

}

}

class SomeClass

{

private Object[] heavyStateObject = new Object[100\_000\_000];

public Runnable capturingLambdaFactory()

{

return () ->

{

System.out.println("SomeClass.capturingLambdaFactory"+this);

};

}

public Runnable nonCapturingLambdaFactory()

{

return () ->

{

System.out.println("SomeClass.nonCapturingLambdaFactory");

};

}

}