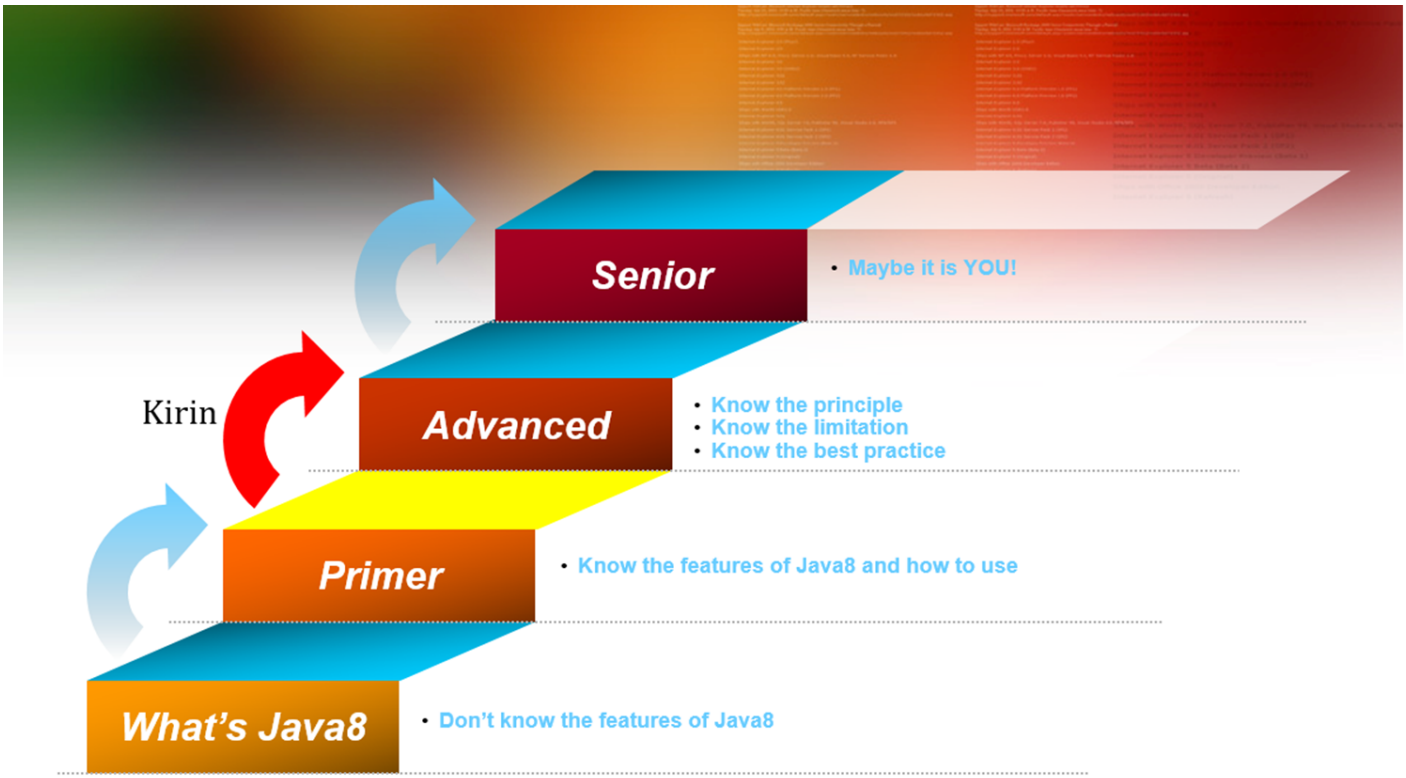
Java8 --Lambda & Method Reference

Deng Zhiquan



Agenda

1.Basic Concept

What's the difference?

```
//before java8
new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("before Java8");
    }
}).start();
```

```
//in java 8
new Thread(()->System.out.println("In Java8")).start();
```

This is a typical and widely used in libraries use case that Java API defines an interface and expecting the user to provide an instance of the interface when invoking the API.

A Code snippet from IOT service:

```
records
    .stream()
    .filter(
        record ->
            !(record
                    .getPartitionKey()
                    .equalsIgnoreCase("techmdev/6cb46dda44184dd5a41a685ecb1c7de8/temp2")
                || record
                    .getPartitionKey()
                    .equalsIgnoreCase("evodev/0040b87d82a7470fa335d94d6327d363/Pitch")))
    // collecting data by linked hash map, which would maintain the order of the data that came in
    .collect(
        Collectors.groupingBy(Record::getPartitionKey, LinkedHashMap::new, Collectors.toList()))
    .entrySet()
    .stream()
    .map(
        entry ->
            CompletableFuture.runAsync(
                () -> processBatchedData(entry.getKey(), entry.getValue()), executorService))
    .map(CompletableFuture::join)
    .collect(Collectors.toList());
```

1.1 Lambda

> Anonymous method(method without name) used to implement a method defined by a *functional interface*.

It's really important to understand what lambda is to avoid some basic errors. lambda expressions can only appear in contexts that have target types. The following contexts have target types:

> - Variable declarations
> - Assignments
> - Return statements
> - Array initializers
> - Method or constructor arguments
> - Lambda expression bodies
> - Conditional expressions (?:)
> - Cast expressions

1.2 Method Reference

> Compact, easy-to-read *lambda expressions* for methods that already have a name.

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

1.3 Functional Interface

> The interface that just has a single abstract method. Which provide target types for lambda expressions and method references

here is a sampling of some of the functional interfaces already in Java SE 7:

```
java.lang.Runnable
java.util.concurrent.Callable
java.security.PrivilegedAction
java.util.Comparator
java.io.FileFilter
java.beans.PropertyChangeListener
```

Java SE 8 adds a new package, java.util.function, which contains functional interfaces that are expected to be commonly used, such as: :

```
Predicate<T> -- a boolean-valued property of an object
Consumer<T> -- an action to be performed on an object
Function<T,R> -- a function transforming a T to a R
Supplier<T> -- provide an instance of a T (such as a factory)
UnaryOperator<T> -- a function from T to T
BinaryOperator<T> -- a function from (T, T) to T
```

Of course, we can create any functional interface as we actually did.
But before we write a functional interface, please first have a glimpse on JDK collections, these can meet most user needs exactly.
If we have to create a new one, please keep in mind use the annotation *@FunctionalInterface* on it.

2. A Peek Under the Hood

2.1 Compile

> Dependency knowledge:
> several features from JSR 292, including invokedynamic, method handles and callsite

> What java compiler did to delegate the lambda real construction approach to runtime during compiling phase are:
>
> - It lowers (desugars) the lambda body into a method whose argument list and return type match that of the lambda expression, possibly with some additional arguments (for values captured from the lexical scope, if any.). New Mehtod naming rule: *lambda${methodname}${increasing int number}*
> - The bootstrap method information
> - Target type information (the funcitonal interface type e.g. Runnable or Consumer etc)

```
public class LambdaApplication {

    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("before Java8" + 1 / 0);  //"1/0" is added to trigger exception to see
            }
        }).start();
```

```
        new Thread(() -> System.out.println("In Java8" + 1 / 0)).start();
    }
}
```

Let's see what happened after compile:

```
D:\txwkp6\coding\DemoLamda\src\main\java\com\example\lamda>javac LambdaApplication.java
D:\txwkp6\coding\DemoLamda\src\main\java\com\example\lamda>dir

LambdaApplication.class    //this is compiled from our source file
LambdaApplication$1.class  //this is from the inner class


//To see what happend for lambda
D:\txwkp6\coding\DemoLamda\src\main\java\com\example\lamda>javap -p LambdaApplication.class
Compiled from "LambdaApplication.java"
public class com.example.lamda.LambdaApplication {
  public com.example.lamda.LambdaApplication();
  public static void main(java.lang.String[]);
  private static void lambda$main$0();
}
```

To double conform, we can run the class we just complied to see the callstack:

```
Exception in thread "Thread-1" java.lang.ArithmeticException: / by zero
at com.example.lamda.LambdaApplication$1.run(LambdaApplication.java:9)
at java.lang.Thread.run(Thread.java:748)
Exception in thread "Thread-2" java.lang.ArithmeticException: / by zero
at com.example.lamda.LambdaApplication.lambda$main$0(LambdaApplication.java:13)
at java.lang.Thread.run(Thread.java:748)
```

What's more hiding works:

> c.2.1 dissemble results

2.2 Runtime Capture

> it generates an invokedynamic call site, which, when invoked, returns an instance of the functional interface to which the lambda is being converted
>
> For this callsite, the dynamic arguments are values captured from the lexical scope. the bootstrap method (lambda metafactory) is offered by java runtime library, bootstap method's static arguments(the functional interface to which it will be converted, a method handle for the desugared lambda body, information about whether the SAM type is serializable, etc.) are from compile time.

```
//the bootstap method from JRE
public static CallSite metafactory(MethodHandles.Lookup caller, //com.example.lamda.LambdaApplication
                                   String invokedName,  //run
                                   MethodType invokedType, //()Runnable
                                   MethodType samMethodType, //()void
                                   MethodHandle implMethod, //MethodHandle()void
                                   MethodType instantiatedMethodType) //()void
```

> What is done during generating the including:
>
> - generate a new class which is the implementation of functional interface
> - generate instance of the new class, use to generate a MethodHandle
> - create and return the dynamic callsite which return the MethodHandle on previous step.
>
> Method references are treated the same way as lambda expressions, except that most method references do not need to be desugared into a new method; we can simply load a constant method handle for the referenced method and pass that to the metafactory

2.3 Performance costs

> Any translation scheme imposes costs at several levels:
>
> - Linkage cost – one-time cost of setting up capture
> - Capture cost – cost of creating a lambda
> - Invocation cost – cost of invoking the lambda method
>
> For inner class instances, these correspond to:
>
> - Linkage: loading the class
> - Capture: invoking the constructor
> - Invocation: invokeinterface

Performance example – capture cost

> Oracle Performance Team measured capture costs
>
> - 4 socket x 10 core x 2 thread Nehalem EX server
> - All numbers in ops/uSec
>
> Worst-case lambda numbers equal to inner classes
>
> Best-case numbers much better

	Single-threaded	**Saturated**
Inner class	160	1407

	Single-threaded	Saturated
Non-capturing lambda	636	23201
Capturing lambda	160	1400

For a full performance output of lambdas, please refer to: JDK 8: Lambda Performance study

3. Best Practice

3.1 prefer lambdas to anonymous classes

> - Omit the types of all lambda parameters unless their presence makes your program clearer.
> - lambdas lack names and documentation; if a computation isn't self-explanatory, or exceeds a few lines, don't put it in a lambda (**less than 3 lines, 1 line is perfect**)
> - you should rarely, if ever, serialize a lambda
> - Don't use anonymous classes for function objects unless you have to create instances of types that aren't functional interfaces

3.2 prefer method references to lambdas

> - Where method references are shorter and clearer, use them; where they aren't, stick with lambdas.

4. References

> - Translation of Lambda Expressions
> - State of Lambda
> - LambdaMetafactory
> - <Effective Java 3rd Edition>
> - <Java 8 in action>
> - <深入理解Java虚拟机>

Attachment

c.2.1 dissemble results:

```
Constant pool:
#7 = InvokeDynamic      #0:#35         // #0:run:()Ljava/lang/Runnable;
#9 = InvokeDynamic      #1:#41         // #1:get:()Ljava/util/function/Supplier;

{
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=4, locals=1, args_size=1
     0: new           #2                  // class java/lang/Thread
     3: dup
     4: new           #3                  // class com/example/lamda/LambdaApplication$1
     7: dup
     8: invokespecial #4                  // Method com/example/lamda/LambdaApplication$1."<init>":()V
    11: invokespecial #5                  // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
    14: invokevirtual #6                  // Method java/lang/Thread.start:()V
    17: new           #2                  // class java/lang/Thread
    20: dup
    21: invokedynamic #7,  0              // InvokeDynamic #0:run:()Ljava/lang/Runnable;
    26: invokespecial #5                  // Method java/lang/Thread."<init>":(Ljava/lang/Runnable;)V
    29: invokevirtual #6                  // Method java/lang/Thread.start:()V
    32: return

private static void lambda$main$1();
descriptor: ()V
flags: ACC_PRIVATE, ACC_STATIC, ACC_SYNTHETIC
Code:
  stack=2, locals=0, args_size=0
     0: getstatic     #8                  // Field java/lang/System.out:Ljava/io/PrintStream;
     3: invokedynamic #9,  0              // InvokeDynamic #1:get:()Ljava/util/function/Supplier;
     8: invokevirtual #10                 // Method java/io/PrintStream.println:(Ljava/lang/Object;)V
    11: return

private static java.lang.String lambda$null$0();
descriptor: ()Ljava/lang/String;
flags: ACC_PRIVATE, ACC_STATIC, ACC_SYNTHETIC
Code:
  stack=1, locals=0, args_size=0
     0: ldc           #11                 // String In Java8
     2: areturn

}
SourceFile: "LambdaApplication.java"
InnerClasses:
 static #3; //class com/example/lamda/LambdaApplication$1
 public static final #70= #69 of #72; //Lookup=class java/lang/invoke/MethodHandles$Lookup of class java/
BootstrapMethods:
```

```
    0: #32 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$L
Method arguments:
  #33 ()V
  #34 invokestatic com/example/lamda/LambdaApplication.lambda$main$1:()V
  #33 ()V
    1: #32 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodHandles$L
Method arguments:
  #38 ()Ljava/lang/Object;
  #39 invokestatic com/example/lamda/LambdaApplication.lambda$null$0:()Ljava/lang/String;
  #40 ()Ljava/lang/String;
```