Lambda Expressions

Venkatesh

WDC

October 29, 2018

Outline

Introduction

Anonymous Classes

- shows you how to implement a base class or `interface` without giving it a name

Introduction

Anonymous Classes

- shows you how to implement a base class or `interface` without giving it a name
- enable you to declare and instantiate a `class` at the same time

Introduction

Anonymous Classes

- shows you how to implement a base class or `interface` without giving it a name
- enable you to declare and instantiate a `class` at the same time
- If an interface contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear

Anonymous class

- It is always an inner class, it is never static

Anonymous class

- It is always an inner class, it is never static
- can implement only one `interface` at a time

Anonymous class

- It is always an inner class, it is never static
- can implement only one `interface` at a time
- can extend a `class` (may be abstract or concrete) only one at a time

Anonymous class

- It is always an inner class, it is never static
- can implement only one `interface` at a time
- can extend a `class` (may be abstract or concrete) only one at a time
- cannot have an explicitly declared constructor

Anonymous Inner class

Syntax

`new` *[TypeArguments]*
ClassOrInterfaceTypeToInstantiate (*[ArgumentList]*)
{*ClassBody*}

Anonymous Inner class

Syntax

`new` *[TypeArguments]*
ClassOrInterfaceTypeToInstantiate (*[ArgumentList]*)
{*ClassBody*}

capture variables

can access local variables in its enclosing scope that are `final` or "effectively final"

Anonymous class that extends a class

Example

```
// Here we are using Anonymous Inner class
// that extends a class i.e. Here a Thread class

Thread t = new Thread() {
      public void run() {
          System.out.println("Child Thread");
      }
  };
t.start();
```

Anonymous class that implements a interface

Example

```
// Here we are using Anonymous Inner class
// that implements a interface i.e. Here Runnable interface

Runnable r = new Runnable() {
    public void run() {
        System.out.println("Child Thread");
    }
};
Thread t = new Thread(r);
t.start();
```

Functional Interface

Definition

An interface that contains *one and only one* abstract method (which is not public method of `Object`)

Functional Interface

Definition

An interface that contains *one and only one* abstract method (which is not public method of `Object`)

Example

```
interface Runnable {
  void run();
}
```

Functional Interface Examples

Example

```
/* Non Functional : equals is a member of Object */
interface NonFunc {
  boolean equals(Object obj);
}
```

Functional Interface Examples

Example

```
/* Non Functional : equals is a member of Object */
interface NonFunc {
  boolean equals(Object obj);
}

/* Functional Interface : an abstract method which is not a
    member of Object */
interface Func extends NonFunc {
  int compare(String o1, String o2);
}
```

Functional Interface Examples

Example

```
/* Non Functional : equals is a member of Object */
interface NonFunc {
  boolean equals(Object obj);
}

/* Functional Interface : an abstract method which is not a
    member of Object */
interface Func extends NonFunc {
  int compare(String o1, String o2);
}

/* java.util.Comparator<T> is functional because it has one
    abstract non-Object method */
interface Comparator<T> {
  boolean equals(Object obj);
  int compare(T o1, T o2);
}
```

Creating Instances of Functional Interface

- Declaring and instantiating a class that `implements` the interface

Creating Instances of Functional Interface

- Declaring and instantiating a class that `implements` the interface
- Lambda expressions

Creating Instances of Functional Interface

- Declaring and instantiating a class that `implements` the interface
- Lambda expressions
- Method reference expressions

Lambda Expressions

- A lambda expression is like a method
- It forms the implementation of the abstract method defined by functional interface

LambdaExpression Syntax:

```
LambdaParameters -> LambdaBody
```

Lambda Expressions ...

- Lambda expressions are always poly expressions (not standalone expressions)

Lambda expression can occurs only in

- assignment context
- invocation context or
- casting context

Lambda Expressions ...

- Lambda expressions are always poly expressions (not standalone expressions)
- Value of a lambda-expr is an instance of a class that `implements` the functional interface type

Lambda expression can occurs only in

- assignment context
- invocation context or
- casting context

Lambda Parameters

formal parameters of a lambda expression

- either *declared types* or *inferred types*
- styles cannot be mixed
- Only parameters with declared types can have modifiers

Lambda Body

- A lambda body is either a *single expression* or *a block*

Lambda Body

- A lambda body is either a *single expression* or *a block*
- `this` and `super` keywords in a lambda body are same as in the surrounding context

Lambda Body

- A lambda body is either a *single expression* or *a block*
- `this` and `super` keywords in a lambda body are same as in the surrounding context
- It can't use a local variable/formal parameter of its enclosing scope unless that variable is `final` or "effectively final"

Lambda Body

- A lambda body is either a *single expression* or *a block*
- `this` and `super` keywords in a lambda body are same as in the surrounding context
- It can't use a local variable/formal parameter of its enclosing scope unless that variable is `final` or "effectively final"
- It cannot modify the local variable/formal parameter of the enclosing scope

Lambda Expressions ...

- Type of the *abstract method* and the type of the lambda expression must be compatible

Lambda Expressions ...

- Type of the *abstract method* and the type of the lambda expression must be compatible
- Type and number of the lambda parameters must be comptaible with method's parameters

Lambda Expressions ...

- Type of the *abstract method* and the type of the lambda expression must be compatible
- Type and number of the lambda parameters must be comptaible with method's parameters
- Any *checked exceptions* that can be thrown in the body of a lambda must be declared

Lambda Expressions Syntax Examples

```
() -> 42  // No parameters, expression body
() -> {return 42;}  // No parameters, block body with return
```

Lambda Expressions Syntax Examples

```
() -> 42  // No parameters, expression body
() -> {return 42;}  // No parameters, block body with return

(int x, int y) -> x + y  // Multiple declared-type parameters
(x, y) -> x + y  // Multiple inferred-type parameters
```

Lambda Expressions Syntax Examples

```
() -> 42  // No parameters, expression body
() -> {return 42;}  // No parameters, block body with return

(int x, int y) -> x + y  // Multiple declared-type parameters
(x, y) -> x + y  // Multiple inferred-type parameters

(x, int y) -> { return x + y; } /* Illegal: can't mix inferred
    and declared types */
(x, final y) -> x+y // Illegal: no modifiers with inferred types
```

Example

```
interface MyNumber {
  double getValue();
}

// Create a reference to a MyNumber instance.
MyNumber myNum;

// Use a lambda in an assignment context.
myNum = () -> 123.45;

/* Call getValue(), which is implemented by the previously
     assigned lambda expression. */
System.out.println(myNum.getValue());
```

More Examples

- BlockLambdaDemo.java

More Examples

- BlockLambdaDemo.java
- LambdaExceptionDemo.java

More Examples

- BlockLambdaDemo.java
- LambdaExceptionDemo.java
- VarCapture.java

More Examples

- BlockLambdaDemo.java
- LambdaExceptionDemo.java
- VarCapture.java
- VarCapture-2.java

Standard Functional Interfaces

- `java.util.function` provides several predefined functional interfaces

Interface	**Function signature**
`UnaryOperator<T>`	`T apply(T t)`
`BinaryOperator<T>`	`T apply(T t1, T t2)`
`Predicate<T>`	`boolean test(T t)`
`Function<T,R>`	`R apply(T t)`
`Supplier<T>`	`T get()`
`Consumer<T>`	`void accept(T t)`

Example

- UseFunctionInterfaceDemo.java

Type of a Lambda Expression

A lambda of the form `() -> stmt-expr` is interpreted as either

```
() -> { return stmt-expr; }
```

or

```
() -> { stmt-expr; }
```

depending on the target type

Example

```
// Predicate has a boolean result
java.util.function.Predicate<String> p = s -> list.add(s);
// Consumer has a void result
java.util.function.Consumer<String> c = s -> list.add(s);
```

Anonymous classes vs Lambdas

- Lambdas are limited to functional interface

Anonymous classes vs Lambdas

- Lambdas are limited to functional interface
- Using an anonymous class, we can create an instance of
  - an abstract class
  - interface with multiple abstract methods

Anonymous classes vs Lambdas

- Lambdas are limited to functional interface
- Using an anonymous class, we can create an instance of
  - an abstract class
  - interface with multiple abstract methods
- In anonymous class, `this` refers to class instance itself

Effective Java

- Prefer lambdas to anonymous classes

Effective Java

- Prefer lambdas to anonymous classes
- Favor the use of standard functional interfaces

References

- Java The Complete Reference 9th Edition, Herbert Schildt
- The Java Language Specification Java SE 10 Edition
- Effective Java, 3rd Edition, Joshua Bloch

Thank You