Lambda

awantikdas@gmail.com

Agenda

What lambda expressions are

Why we need lambda expressions

The syntax for defining lambda expressions

Target typing for lambda expressions

Commonly used built-in functional interfaces

Method and Constructor references

Lexical scoping of lambda expressions

What ?

A lambda expression is not a method, although its declaration looks similar to a method. As the name suggests, a lambda expression is an expression that represents an instance of a functional interface.

```
// Takes an int parameter and returns the parameter value incremented by 1
(int x) -> x + 1

// Takes two int parameters and returns their sum
(int x, int y) -> x + y

// Takes two int parameters and returns the maximum of the two
(int x, int y) -> { int max = x > y ? x : y;
return max;
}

// Takes no parameters and returns void
() -> { }
```

```
// Takes no parameters and returns a string "OK"
() -> "OK"

// Takes a String parameter and prints it on the standard output
(String msg) -> { System.out.println(msg); }

// Takes a parameter and prints it on the standard output
msg -> System.out.println(msg)

// Takes a String parameter and returns its length
(String str) -> str.length()
```

Why ?

```
// Using an anonymous class
StringToIntMapper mapper = new StringToIntMapper() {
@Override
public int map(String str) {
return str.length();
}
};

// Using a lambda expression
StringToIntMapper mapper = (String str) -> str.length();
```

Syntax for Lambda Expressions

(<LambdaParametersList>) -> { <LambdaBody> }

- A lambda expression does not have a name.
- A lambda expression does not have a return type. It is inferred by the compiler from the

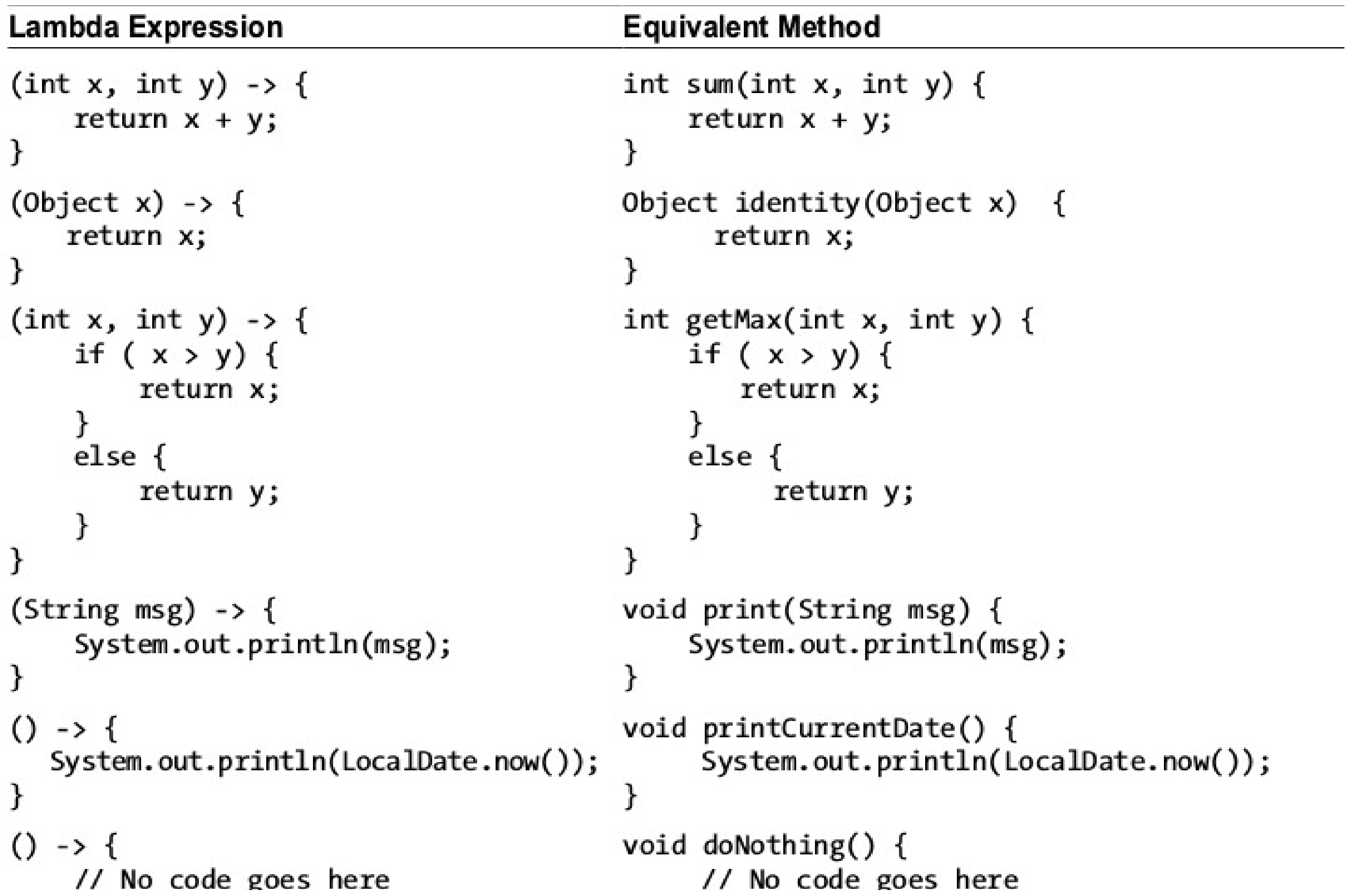
context of its use and from its body.

- A lambda expression does not have a throws clause. It is inferred from the context of its use

and its body.

- A lambda expression cannot declare type parameters. That is, a lambda expression cannot be generic.

Expression



Lambda Expression	Equivalent Method
`(int x, int y) -> { return x + y; }`	`int sum(int x, int y) { return x + y; }`
`(Object x) -> { return x; }`	`Object identity(Object x) { return x; }`
`(int x, int y) -> { if ( x > y) { return x; } else { return y; } }`	`int getMax(int x, int y) { if ( x > y) { return x; } else { return y; } }`
`(String msg) -> { System.out.println(msg); }`	`void print(String msg) { System.out.println(msg); }`
`() -> { System.out.println(LocalDate.now()); }`	`void printCurrentDate() { System.out.println(LocalDate.now()); }`
`() -> { // No code goes here`	`void doNothing() { // No code goes here`

Omitting Parameter Types

```
// Types of parameters are declared
(int x, int y) -> { return x + y; }

// Types of parameters are omitted
(x, y) -> { return x + y; }

// A compile-time error
(int x, y) -> { return x + y; }
```

Declaring No Parameters

```
// Takes no parameters
() -> { System.out.println("Hello"); }
```

It is not allowed to omit the parentheses when the lambda expression takes no parameter. The following declaration will not compile:

```
-> { System.out.println("Hello"); }
```

Declaring a Single Parameter

```
// Declares the parameter type
(String msg) -> { System.out.println(msg); }

// Omits the parameter type
(msg) -> { System.out.println(msg); }

// Omits the parameter type and parentheses
msg -> { System.out.println(msg); }
```

The parentheses can be omitted only if the single parameter also omits its type. The following lambda expression
will not compile:

```
// Omits parentheses, but not the parameter type, which is not allowed.
String msg -> { System.out.println(msg); }
```

Declaring Body of Lambda Expressions

```
// Uses a block statement. Takes two int parameters and returns their sum.
(int x, int y) -> { return x + y; }

// Uses an expression. Takes a two int parameters and returns their sum.
(int x, int y) -> x + y
// Uses a block statement
(String msg) -> { System.out.println(msg); }

// Uses an expression
(String msg) -> System.out.println(msg)
```

Target Typing

A poly expression is an expression that has different types in different contexts. The compiler determines the type of the expression. The contexts that allow the use of poly expressions are known as poly contexts. All lambda expressions in Java are poly expressions.

Adder adder = (double x, double y) -> x + y;

The compiler now verifies the compatibility of the returned value from the lambda expression and the return type of the add() method. The return type of the add() method is double. The lambda expression returns x + y, which

would be of a double as the compiler already knows that the types of x and y are double. The lambda expression does not throw any checked exceptions. Therefore, the compiler does not have to verify anything for that. At this point, the compiler infers that the type of the lambda expression is the type Adder.

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. For example,

ReferenceType variable1 = LambdaExpression;

Method Invocation Context: A lambda expression may appear as an argument to a method or constructor call. For example,

util.testJoiner(LambdaExpression);

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. For example,

return LambdaExpression;

Cast Context: A lambda expression may be used if it is preceded by a cast. The type specified in the cast is its target type. For example,

(Joiner) LambdaExpression;

Functional Interface

A functional interface is simply an interface that has exactly one abstract method. The following types of methods in

an interface do not count for defining a functional interface:

- Default methods
- Static methods
- Public methods inherited from the Object class

```
@FunctionalInterface
public interface Comparator<T> {
// An abstract method declared in the interface
int compare(T o1, T o2);
// Re-declaration of the equals() method in the Object class
boolean equals(Object obj);
/* Many static and default methods that are not shown here. */
}
```

- Using the @FunctionalInterface Annotation – More of assurance from the compiler

Interface Name	Method	Description
`Function<T,R>`	`R apply(T t)`	Represents a function that takes an argument of type `T` and returns a result of type `R`.
`BiFunction<T,U,R>`	`R apply(T t, U u)`	Represents a function that takes two arguments of types `T` and `U`, and returns a result of type `R`.
`Predicate<T>`	`boolean test(T t)`	In mathematics, a predicate is a boolean-valued function that takes an argument and returns true or false. The function represents a condition that returns true or false for the specified argument.
`BiPredicate<T,U>`	`boolean test(T t, U u)`	Represents a predicate with two arguments.
`Consumer<T>`	`void accept(T t)`	Represents an operation that takes an argument, operates on it to produce some side effects, and returns no result.
`BiConsumer<T,U>`	`void accept(T t, U u)`	Represents an operation that takes two arguments, operates on them to produce some side effects, and returns no result.
`Supplier<T>`	`T get()`	Represents a supplier that returns a value.
`UnaryOperator<T>`	`T apply(T t)`	Inherits from `Function<T,T>`. Represents a function that takes an argument and returns a result of the same type.
`BinaryOperator<T>`	`T apply(T t1, T t2)`	Inherits from `BiFunction<T,T,T>`. Represents a function that takes two arguments of the same type and returns a result of the same.