Chapter NINE

Lambda Expressions

What is a lambda expression?

- The term lambda expression comes from lambda calculus, written as "λ-calculus"
- λ is the Greek letter lambda. This form of calculus deals with defining and applying functions.

In JAVA

Lambda expressions, are a short-form replacement for anonymous classes. Lambda expressions simplify the use of interfaces that declare a single abstract method, which are also called functional interfaces., a single method interface can be implemented with one of the following options.

A lambda expression can be used to implement a functional interface without creating a class or an anonymous class. Lambda expressions can be used only with interfaces that declare a single method.

Benefits of lambda expressions

- Concise syntax
- Method references and constructor references
- Convenient for new streams library
- Reduced runtime overhead compared to anonymous classes

Understanding Lambda Syntax

Figure 9.1

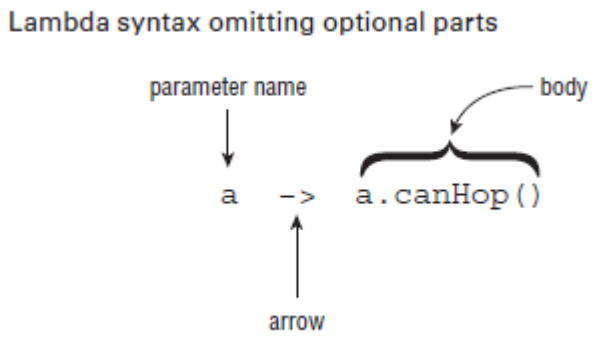
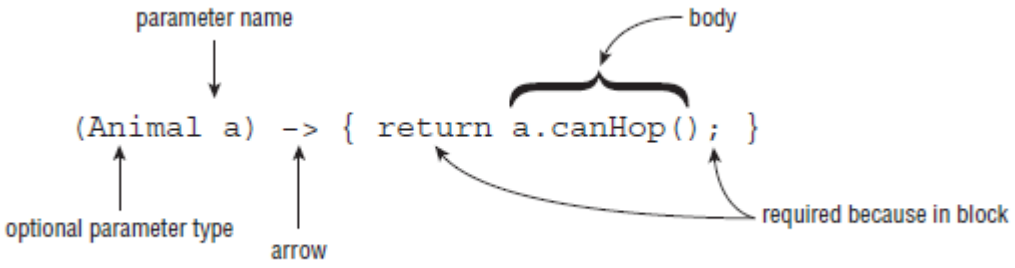


Figure 9.2



A lambda expression has three parts:

A list of parameters

A lambda expression can have zero (represented by empty parentheses), one or more parameters:

```
() -> System.out.println("Hi");
(String s) -> System.out.println(s);
(String s1, String s2) -> System.out.println(s1 + s2);
```

The type of the parameters can be declared explicitly, or it can be inferred from the context:

```
(s) -> System.out.println(s);
```

If there is a single parameter, the type is inferred and it is not mandatory to use parentheses:

```
s -> System.out.println(s);
```

If the lambda expression uses a parameter name which is the same as a variable name of the enclosing context, a compile error is generated:

```
// This doesn't compile
String s = ""; s -> System.out.println(s);
```

An arrow

Formed by the characters `-` and `>` to separate the parameters and the body.

A body

The body of the lambda expressions can contain one or more statements.

If the body has one statement, curly brackets are not required and the value of the expression (if any) is returned:

```
() -> 4; (int a) -> a*6;
```

If the body has more than one statement, curly brackets are required, and if the expression returns a value, it must be returned with a return statement:

```
() -> {
      System.out.println("Hi");
      return 4;
}
(int a) -> {
      System.out.println(a);
      return a*6;
}
```

If the lambda expression doesn't return a result, a return statement is optional. For example, the following expressions are equivalent:

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

For example, the following are all valid lambda expressions, assuming that there are valid functional interfaces that can consume them:

```
() -> new Duck();

d -> {return d.getType();}

(Duck d) -> d.getType()

(Animal a, Duck d) -> d.getType()

() -> true

a -> {return a.startsWith("test");}

(String a) -> a.startsWith("test")

(int x) -> {}

(int y) -> {return;}

(a, b) -> a+b

(String a, String b) -> a.equals(b)

(y, z) -> { int x=y; return z+x; }

(String s, int z) -> { return s.length()+z; }

(a, b, c) -> a.getName()

(a, b) -> { int a = 0; return 5;} // DOES NOT COMPILE
```

Lambda Expressions: Java 8

Some examples of function definitions:

- in Javascript : **function () { return x + 1 };**
- in LISP: **(lambda (x) x + 1)**
- in C++11: **[](int x) { return x + 1; }**
- in Scala: **x => x + 1 or just _ + 1**
- in Java 8: **(int x) -> x + 1**

Examples of Lambda Expressions and Equivalent Methods

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`
	`}`

How are functional interfaces related to all this?

The signature of the abstract method of a functional interface provides the signature of a lambda expression (this signature is called a ***functional descriptor***).

This means that to use a lambda expression, you first need a functional interface. For example, using the interface of the previous chapter:

```
interface Searchable {
     boolean test(Car car);
}
```

We can create a lambda expression that takes a `Car` object as argument and returns a `boolean`:

```
Searchable s = (Car c) -> c.getCostUSD() > 20000;
```

In this case, the compiler inferred that the lambda expression can be assigned to a `Searchable` interface, just by its signature.

In fact, lambda expressions don't contain information about which functional interface they are implementing. The type of the expression is deduced from the context in which the lambda is used. This type is called ***target type***.

If we were using the lambda as an argument to a method, the compiler would use the definition of the method to infer the type of the expression:

```
class Test {
     public void find() {
          find(c -> c.getCostUSD() > 20000);
     }
     private void find(Searchable s) {
          // Here goes the implementation
     }
}
```

Because of this, the same lambda expression can be associated with different functional interfaces if they have a compatible abstract method signature. For example:

```
interface Searchable {
     boolean test(Car car);
}
interface Saleable {
     boolean approve(Car car);
}
//...
Searchable s1 = c -> c.getCostUSD() > 20000;
Saleable s2 = c -> c.getCostUSD() > 20000;
```

For reference, the contexts where the target type (the functional interface) of a lambda expression can be inferred are:

- A variable declaration

```
public class LambdaExpressionTest1 {
    Supplier<Car> SupplierCas = ()-> new Car(1, "Twenty");

}
```

- An assignment

```
Supplier<Car> SupplierCas = ()-> new Car(1, "Twenty");
Car car1 = SupplierCas.get();
```

- A return statement

```
public static Searchable Test(List<Car> cars) {
    return car -> car.getType().equals(CarTypes.COMPACT);
}
```

- An array initializer

```
interface IArrayInitializer<T> {
    public T[] apply(int size);
}
```

```
IArrayInitializer<String> initializer = (a) -> new String[a];
String[] stringList =  initializer.apply(10);
```

- Method or constructor arguments

```
public static List<Car> findCars(List<Car> cars) {
     return findCars(cars , car -> car.getType().equals(CarTypes.COMPACT));
}
```

```
public class LambdaExpressionTest implements Searchable {
    Supplier<LambdaExpressionTest> insSearchable;

    public LambdaExpressionTest(){
        insSearchable = () -> new LambdaExpressionTest();
    }
}
```

- A ternary conditional expression

```
  Searchable checkCar = e -> e != null ? true: false;
```

- A cast expression

```
interface CastCar<T> {
    public Car castCarTest(T t);
}
```

```
SubCar subCar = new SubCar(1, "A");
CastCar<SubCar> castCar = z -> (Car)z;
Car car = castCar.castCarTest(subCar);
```

***However, if you understand the concept, you don't need to memorize this list.

Lambda expressions VS Anonymous classes

- They have some similarities:
  - Local variables (variables or parameters defined in a method) can only be used if they are declared final or are effectively final.

```
//Anonymous classes
boolean test = false;
Searchable search = new Searchable() {
    @Override
    public boolean test(Car car) {
        return test; //compilation Error
    }
};
test = true;


//Lambda expressions
boolean test1 = false;
Searchable search1 = c -> {return test1;}; //compilation Error
test1 = true;
```

- You can access instance or static variables of the enclosing class.

```
public class Test {
    static boolean testinstance  = false;

    public static void main(String[] args) {
        //Anonymous classes
        Searchable search = new Searchable() {
            @Override
            public boolean test(Car car) {
                return testinstance;
            }
        };
        testinstance = true;

        //Lambda expressions
        Searchable search1 = c -> {return testinstance;};
        testinstance = true;
```

- They must not throw more exceptions than specified in the throws clause of the functional interface method. Only the same type or a supertype.

```
@FunctionalInterface
interface MyException {
    public void getException() throws FileNotFoundException;
}
```

```
public static void main(String[] args) {
    //Anonymous classes
    try {
        getException(new MyException() {
            @Override
            public void getException() throws FileNotFoundException {
                throw new FileNotFoundException();
            }
        });
    } catch (IOException e1) {
        e1.printStackTrace();
    }

    //Lambda expressions
    try {
        getException( () -> {throw new FileNotFoundException();} );
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static void getException(MyException ex) throws FileNotFoundException{
    ex.getException();
}
```

- And some significant differences:
  - Scoping

```
1  public static void main(String[] args) {
2      final int cnt = 0;
3      Runnable r = new Runnable() {
4          @Override
5          public void run() {
6              int cnt = 5;
7              System.out.println("in run" + cnt);
8          }
9      };
10
11     Thread t = new Thread(r);
12     t.start();
13 }
```

```
1 public static void main(String[] args) {
2     final int cnt = 0;
3     Runnable r = ()->{
4         int cnt = 5; //compilation error
5         System.out.println("in run"+cnt);};
6     Thread t = new Thread(r);
7     t.start();
8 }
```

  - Default methods of a functional interface cannot be accessed from within lambda expressions. Anonymous classes can.

```
@FunctionalInterface
interface AnInterface {
    default int aMethod() { return 0; }
    int anotherMethod();
}
```

```
//Anonymous classes
AnInterface aa = new AnInterface() {
    @Override
    public int anotherMethod() {
        return 1;
    }

    public int aMethod(){
        return 1;
    }
};

//Lambda expressions
AnInterface a1 = () -> aMethod(); //Lambda not access default aMethod() in Function Interface
System.out.println(a1.anotherMethod());
```

- Performance

At runtime anonymous inner classes require class loading, memory allocation and object initialization and invocation of a non-static method while lambda expression is pure compile time activity and don't incur extra cost during runtime. So performance of lambda expression is better as compare to anonymous inner classes.

Key Points

- Lambda expressions have three parts: a list of parameters, and arrow, and a body:

```
(Object o) -> System.out.println(o);
```

- You can think of lambda expressions as anonymous methods (or functions) as they don't have a name.
- A lambda expression can have zero (represented by empty parentheses), one or more parameters.
- The type of the parameters can be declared explicitly, or it can be inferred from the context.
- If there is a single parameter, the type is inferred and is not mandatory to use parentheses.
- If the lambda expression uses as a parameter name which is the same as a variable name of the enclosing context, a compile error is generated.
- If the body has one statement, curly brackets are not required, and the value of the expression (if any) is returned.
- If the body has more than one statement, curly brackets are required, and if the expression returns a value, it must return with a `return` statement.
- If the lambda expression doesn't return a result, a `return` statement is optional.
- The signature of the `abstract method` of a functional interface provides the signature of a lambda expression (this signature is called a *functional descriptor*).
- This means that to use a lambda expression, you first need a functional interface.
- The type of the expression is deduced from the context in which the lambda is used. This type is called *target type*.
- The contexts where the target type of a lambda expression can be inferred include an assignment, method or constructor arguments, and a cast expression.
- Like anonymous classes, lambda expressions can access instance and static variables, but only final or effectively final local variables.
- Also, they cannot throw exceptions that are not defined in the `throws` clause of the function interface method.

- Default methods of a functional interface cannot be accessed from within lambda expressions.