Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Lambda Expressions, Functional Interfaces and Method References

Source: K. Sharan, Beginning Java 8 Language Features, Lambda Expressions, Inner Classes, Threads, I/O, Collections and Streams, Chapter 5, Apress, 2014

T. Cioara, V. Chifu, C. Pop
2025

Functional Programming

- Main abstractions to perform calculations
  - mathematical functions / functions with recursions and
  - lambda calculus
- Types of FP languages
  - **Pure**: ex. Haskel
    - Pure functions – no side effects, given the same input always produces the same output
  - **Impure**: ex. Java multiparadigm programming (OO + Functional)

Functional Programming

- **Main functional features** - **<u>be declarative</u>** (instead of imperative)

Imperative style

```
boolean found = false;
for(String city : cities) {
   if(city.equals("Cluj")) {
       found = true; break;
   }
}
System.out.println("Found Cluj?:"+found);
```

- Mutability and command driven programming
- Variables/object creation
- Modify their state
- Provide detailed commands/ instructions to execute (create a loop index, increment its value, etc.)

Declarative style

```
System.out.println("Found Cluj?:" + cities.contains("Cluj"));
```

- **cities – immutable collection**
- All the hard work and the lower-level details were moved into the library function

Functional Programming

- **Main functional features – promote immutability**
  - Imperative: code with multiple variables that change over time
    - Drawback: hard to understand, error prone, and difficult to parallelize
    - Java supports immutability but does not enforce it
  - How to define immutable entities in Java
    - Variables, fields, and parameters - declared as final
    - Promote immutable type fields in classes
    - When working with collections - create immutable or unmodifiable collections using functions
  - By avoiding mutability, we can create pure functions - that is, functions with no side effects.

Functional Programming

- **Main functional features – prefer declarative expressions over imperative statements**

Imperative style

```
final List<Laptop> laptops = Arrays.asList(new Laptop(…), …);
double totalDiscount = 0;
for(Laptop lt : laptops) {
  double p = lt.getPrice();
  if(p > 1000) totalDiscount += 0.1 * p;}

System.out.println("Total discount: " + totalDiscount);
```

Declarative style

```
final Double totalOfDiscount = laptops.stream()
   .filter(lt->lt.getPrice() > 1000)
   .map(lt -> lt.getPrice() * 0.1)
   .reduce(0.0, Double::sum);
```

- Avoids mutation
- Expressions (filter, map, reduce) promote immutability and function composition
- The code flows logically, as states the problem
- Code is easier to change if requirements change

Functional Programming

- **Main functional features – design with higher order functions**
  - Functions that take other functions as arguments or return functions as results
  - Higher-order functions
    - Powerful tools for building abstractions and composing behavior
  - Functional Programing
    - Reuse small, focused, cohesive, and well-written functions
    - Higher-order functions in FP do to functions what methods did to objects
  - OOP
    - Relies on objects and classes to promote reuse
    - Pass objects to methods, creating objects within methods, and returning objects from within methods.

Functional Programming

- **OOP versus FP**

Object-oriented Programming	Functional Programming
Imperative programming	Declarative programming
Mutable data	Immutable data
Main approach: What + How you do it	Main approach: What are you doing
Not suitable for parallel programming	Supports parallel programming
Many side effects	No side effects
Flow control with loops and conditionals	Flow of control with function calls & function calls with recursion
Execution order of statements is important	Execution order of statements is not that important

Lambda Expressions

- Anonymous block of code
  - Notation: We will use λ ex for lambda expression
  - Describes an anonymous function
  - Lambda body may
    - Declare local variables;
    - Use statements including break, continue, and return;
    - Throw exceptions, etc.

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

- **A λ ex has no:**
  - Name
  - Return type and Throws clause - is inferred by compiler from the context of its use and from its body
- **No generics with λ ex**

Lambda Expressions

- Examples

```
(int x) -> x + 1

(int x, int y) -> x + y

(int x, int y) -> { int max = x > y ? x : y; return max; }

() -> { }

(String msg) -> { System.out.println(msg); }

msg -> System.out.println(msg)

(String str) -> str.length()
```

- **Explicit-typed λ ex - declares the types of its parameters**
- **Implicit-typed λ ex – - the compiler will infer parameters from context**
- **The parentheses can be omitted only if the single parameter also omits its type**
- **A block statement is enclosed in braces; single expression – no braces**

Lambda Expressions

- λ expression type
  - Every expression must have a type
    - Standalone expressions – can be determined without knowing the context of use;
      - Examples: new Integer(3) => the type is Integer;
    - Poly Expressions – different types in different contexts
  - λ ex are poly expressions
  - **λ expression type is Functional Interface**
    - The exact type depends on the context in which it is used

Lambda Expressions

- The compiler infers the type of a λ ex
  - The context in which a λ ex is used expects a target type

```
T t = <LambdaExpression>;
```

The target type of the λ ex is T

Inferring rules used by compiler (they are close related to the abstract method of the Functional Interface)

- T must be a Functional Interface type
- λ ex has the same number and type of parameters as the abstract method of T
- For an implicit λ ex, parameters types are inferred from the abstract method of T
- The type of the returned value from the body of the λ ex should be assignment compatible to the return type of the abstract method of T
- If the body of the λ ex throws any checked exceptions, they must be compatible with the declared throws clause of the abstract method of T
- It is a compile-time error to throw checked exceptions from the body of a λ ex, if its target type's method does not contain a throws clause

Main objective of using λ ex : keep its syntax concise and let the compiler infer the details

Lambda Expressions

- Examples of inferring the target type

```
@FunctionalInterface
public interface Adder {
   double add(double n1, double n2);
}

@FunctionalInterface
public interface Joiner {
   String join(String s1, String s2);
}

Adder adder = (x, y) -> x + y; // the type of λ ex is Adder
Joiner joiner = (x, y) -> x + y; // // the type of λ ex is Joiner

double sum1 = adder.add(10.34, 89.11); // Adds two doubles
double sum2 = adder.add(10, 89); // Adds two ints

String str = joiner.join("Hello", " lambda"); // Joins two strings
```

Lambda Expressions

- Examples of passing Functional Interfaces as arguments to methods

```
public class Lambda1 {
  public void testAdder(Adder adder) {
    double x = 1.1; double y = 2.2; double sum = adder.add(x, y);
  }
  public void testJoiner(Joiner joiner) {
    String s1 = "Hello"; String s2 = "World"; String s3 = joiner.join(s1,s2);
  }
}

Lambda1 lbd1 = new Lambda1();
lbd1.testAdder((x, y) -> x + y);
```

- The compiler must infer the type of the λ ex
  - The target type of the λ ex is the type Adder because the argument type of the testAdder(Adder adder) is the Functional Interface Adder.
  - Compiler infers that type of the λ ex is Adder

Lambda Expressions

- Examples of passing Functional Interfaces as arguments to methods

```
Lambda1 lbd1 = new Lambda1();
lbd1.testJoiner((x, y) -> x + y);

// adds a space between the two strings
lbd1.testJoiner((x, y) -> x + " " + y);

// The Joiner - reverse the strings and join
// resulting strings
lbd1.testJoiner((x, y) -> {
  StringBuilder sbx = new StringBuilder(x);
  StringBuilder sby = new StringBuilder(y);

  sby.reverse().append(",").append(sbx.reverse());
  return sby.toString();
  });
```

- Each time testJoiner displays different results
- testJoin method was parametrized
- **Behavior parametrization (or passing code as data)**
  - Changing the behavior of a method through its parameters
  - The code is passed encapsulated in lambda expressions to methods as if it is data

Lambda Expressions

- The case of overloaded methods
  - When passing λ ex to overloaded methods the compiler cannot infer the type of a λ ex or may generate ambiguity
    - Those contexts do not allow the use of λ ex
    - Some contexts may allow using λ ex, but the use itself may be ambiguous to the compiler
  - Three methods to help the compiler resolve the ambiguity
    - If the λ ex is implicit, make it explicit by specifying the type of the parameters
    - Use a cast
    - Do not use the λ ex directly as the method argument. First, assign it to a variable of the desired type, and then, pass the variable to the method

Lambda Expressions

- Contexts where lambda expressions can be used
  - **Assignment context**: A lambda expression may appear to the right side of the assignment operator in an assignment statement.

```
ReferenceType variable1 = LambdaExpression;
```

  - **Method invocation context**: A lambda expression may appear as an argument to a method or constructor call.

```
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

```
return LambdaExpression;
```

  - **Cast context**: A lambda expression may be used if it is preceded by a cast which specifies its target type.

```
(Joiner) LambdaExpression;
```

Functional Interfaces

- Functional Interface (FI)
  - an interface that has **exactly one abstract method**
  - Other method types in an interface do not count for defining a FI
  - **Annotation**: @FunctionalInterface
  - A FI represents one type of operation in terms of its single abstract method
  - A lambda expression defines the body of the abstract method in FI

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);        // abstract method
    boolean equals(Object obj); // re-declaration of Object class equals
    // … Many other static and default methods - not shown here
}
```

Functional Interfaces

- Generic functional interface example

```
@FunctionalInterface
public interface Mapper<T> {
  int map(T source);    // the abstract method

  public static <U> int[] mapToInt(U[] list,  Mapper<? super U> mapper)  {
       int[] mappedValues = new int[list.length];
       for (int i = 0; i < list.length; i++) {
            mappedValues[i] = mapper.map(list[i]); }
       return mappedValues;
   }
}

String[] names = {“Popescu", “Pop", “’Popica"};
int[] lengthMapping = Mapper.mapToInt(names, (String name) -> name.length());

Integer[] numbers = {7, 3, 67};
int[] countMapping = Mapper.mapToInt(numbers, (Integer n) -> n * n);
```

Functional Interfaces

- Intersection types and lambda expression
  - New Java 8 type – a subtype of multiple types
    - **Type1 & Type2 & Type3**
  - Intersection type may appear as target type in casts

The intersection type Sensitive & Adder is a functional interface, and therefore, the target type of the lambda expression is a functional interface with one method from the Adder interface

```
public interface Sensitive {
      // marker interface (no methods)
}

Sensitive sen = (x, y) -> x + y; // A compile-time error
Sensitive sen = (Sensitive & Adder) (x, y) -> x + y; // OK
```

A lambda expression to be serialized:

```
Serializable ser = (Serializable & Adder) (x, y) -> x + y;
```

Functional Interfaces

- Common Functional Interfaces defined in java.util.function

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.

Functional Interfaces

- Interface Function

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t); // the abstract method
    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }
    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }
…
}
```

- **apply** - applies this function to the argument
- **andThen** - Returns a composed Function that first applies this Function to the argument and then applies the specified after function to the result
- **compose** - Returns a composed function that first applies the before function to the argument and then applies this function to the result

Functional Interfaces

- Interface Function examples

```
// Create two functions
Function<Long, Long> square = x -> x * x;
Function<Long, Long> addOne = x -> x + 1;

// Compose functions from the two functions
Function<Long, Long> squareAddOne = square.andThen(addOne);
// in Maths: squareAddOne = (addOne o square) (x) = addOne(square(x))

Function<Long, Long> addOneSquare = square.compose(addOne);
// in Maths: addOneSquare = (square o addOne) (x) = square(addOne(x))

// Test the functions
long num = 5L;
System.out.println("Number : " + num);
System.out.println("Square and then add one: " + squareAddOne.apply(num));
System.out.println("Add one and then square: " + addOneSquare.apply(num));
```

Functional Interfaces

- Interface Function examples
  - Chaining lambda functions

```
// Square the input, add one to the result, and square the result
Function<Long, Long> chainedFunction = ((Function<Long, Long>)(x -> x * x))
                                       .andThen(x -> x + 1)
                                       .andThen(x -> x * x);

System.out.println(chainedFunction.apply(3L));
```

  - Function<T, R> specializations:
    - IntFunction<R>, LongFunction<R>, DoubleFunction<R>
      - Take an argument of int, long or double and return a value of type R
    - ToIntFunction<T>, ToLongFunction<T> , ToDoubleFunction<T>
      - Take argument of type T and return an int, long or double

Functional Interfaces

- Interface Predicate
  - Abstract method

```
boolean test(T t)
```

  - Default and static methods
    - Allow to compose a predicate based on other predicates and logical operators NOT, AND, OR
    - The methods can be chained to create complex predicates

```
default Predicate<T> negate()
default Predicate<T> and (Predicate<? super T> other)
default Predicate<T> or(Predicate<? super T> other)
```

Functional Interfaces

- **Interface Predicate examples**

```
// Create some predicates
Predicate<Integer> greaterThanTen = x -> x > 10;
Predicate<Integer> divisibleByThree = x -> x % 3 == 0;
Predicate<Integer> divisibleByFive = x -> x % 5 == 0;
Predicate<Integer> equalToTen = Predicate.isEqual(10);

// Create complex predicates using NOT, AND, and OR on other predcates
Predicate<Integer> lessThanOrEqualToTen = greaterThanTen.negate();
Predicate<Integer> divisibleByThreeAndFive = divisibleByThree.and(divisibleByFive);
Predicate<Integer> divisibleByThreeOrFive = divisibleByThree.or(divisibleByFive);

int num = 10;
System.out.println("greaterThanTen: " + greaterThanTen.test(num));
System.out.println("divisibleByThree: " + divisibleByThree.test(num));

System.out.println("lessThanOrEqualToTen: " + lessThanOrEqualToTen.test(num));
System.out.println("divisibleByThreeAndFive: " + divisibleByThreeAndFive.test(num));
```

Functional Interfaces

- Interface Consumer definition

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

  - Used in situations when objects need to be consumed
    - takes an object and eventually perform an operation without returning any result
  - Abstract method: accept: (T) -> ()
  - Default method: andThen
    - Takes as input another instance of Consumer interface and returns as a result a new consumer interface which represents aggregation of both operations defined in the two Consumer interfaces

Functional Interfaces

- Interface Consumer example

```
Consumer<String> consumer = s -> System.out.print(s + " ");
Consumer<String> consumerWithAndThen =consumer.andThen(s -> System.out.print("(printed "+ s + ")"));

List<String> strList=Arrays.asList( new String("Ion"), new String("Vasile"), new String("Sandu"));

public static void printList(List<String> ls, Consumer<String> cons){
    for(String s : ls) {cons.accept(s); }
 }

printList(strList, consumer); //Output consumer 1: Ion Vasile Sandu

printList(strList, consumerWithAndThen);
// Output andThen: Ion (printed Ion) Vasile (printed Vasile) Sandu (printed Sandu)
```

- Interface Supplier example

```
@FunctionalInterface
public interface Supplier<T> {
    T get(); // abstract method
}
```

```
Supplier<String> hello = () -> new String(“Hello");
String sayHello = hello.get();
```

Method References

- Shorthand to create λ expressions using existing methods
  - Can only be used where a λ ex can be used
  - MR is not a new type in Java or a pointer to functions (as in other languages);
  - Syntax

```
<Qualifier>::<MethodName>
```

  - <Qualifier> depends on the type of the method reference
  - <MethodName> is the name of the method

- MR does not call the method when it is declared
  - The method is called later, when the method of its target type is called

Method References

- Examples – lambda expressions versus method references
  - Using *λ ex* to define an anonymous function that takes String argument and returns its length

```
import java.util.function.ToIntFunction;
...
ToIntFunction<String> lengthFunction = str -> str.length();

String name = “Popescu";
int len = lengthFunction.applyAsInt(name);
```

  - Example using MR to the method length of class String

```
import java.util.function.ToIntFunction;
...
ToIntFunction<String> lengthFunction = String::length;

String name = “Popescu";
int len = lengthFunction.applyAsInt(name);
```

Method References

- In a MR is not allowed to specify parameter and return type
  - is a shorthand of a λ ex and the target type determines the details
  - If the method is overloaded, compiler chooses the most specific method based on context
  - Types of Method References

Syntax	Description
`TypeName::staticMethod`	A method reference to a static method of a class, an interface, or an enum
`objectRef::instanceMethod`	A method reference to an instance method of the specified object
`ClassName::instanceMethod`	A method reference to an instance method of an arbitrary object of the specified class
`TypeName.super::instanceMethod`	A method reference to an instance method of the supertype of a particular object
`ClassName::new`	A constructor reference to the constructor of the specified class
`ArrayTypeName::new`	An array constructor reference to the constructor of the specified array type

Method References

- Static MRs examples

```
static String toBinaryString(int i)

//  Using a lambda expression
Function<Integer, String> func1 = x -> Integer.toBinaryString(x);
System.out.println(func1.apply(17)); // generates 10001

//  Using a static MR
Function<Integer, String> func2 = Integer::toBinaryString;
```

```
static int sum(int a, int b)
Using a lambda expression

BiFunction<Integer, Integer, Integer> func1 = (x, y) -> Integer.sum(x, y);
System.out.println(func1.apply(17, 15));

//Using a static MR
BiFunction<Integer, Integer, Integer> func2 = Integer::sum;
```

Method References

- Static MRs examples

```
//  Class Integer, method valueOf
static Integer valueOf(int i)
static Integer valueOf(String s)
static Integer valueOf(String s, int radix)

//Using a static MR
// Uses Integer.valueOf(int)
Function<Integer, Integer> func1 = Integer::valueOf;

// Uses Integer.valueOf(String)
Function<String, Integer> func2 = Integer::valueOf;

// Uses Integer.valueOf(String, int)
BiFunction<String, Integer, Integer> func3 = Integer::valueOf;

System.out.println(func1.apply(17));
System.out.println(func2.apply("17"));
System.out.println(func3.apply("10001", 2));
```

Method References

- Instance MRs
  - The object reference on which the instance method is invoked is known as the receiver of method invocation
  - Receiver of method invocation can be
    - Object reference or expression that evaluates to object reference
  - Instance MR has two forms
    - **Bound receiver -** Specify the receiver of the method invocation explicitly

```
objectRef::instanceMethod
```

    - **Unbound receiver -** Specify the receiver of the method invocation implicitly when the method is invoked

```
ClassName::instanceMethod
```

```
String name = “Popescu";
int len1 = name.length();    // name is the receiver of the length() method
int len2 = "Hello".length();  // "Hello" is the receiver of the length() method
int len3 = (new String(“Popescu”)).length();
```

Method References

- Instance MRs – examples for bound receiver

```
// As Lambda Expression
Supplier<Integer> supplier = () -> “Popescu".length();
System.out.println(supplier.get());  // prints 7

// As MR:  Re-write using Instance MR (The object “Popescu” is the bound
// receiver, Supplier<Integer> is the target type
Supplier<Integer> supplier = “Popescu"::length;
System.out.println(supplier.get());
```

```
// As Lambda Expression
Consumer<String> consumer = str -> System.out.println(str);
consumer.accept("Hello");

// As MR with System.out as bound receiver
Consumer<String> consumer = System.out::println;
consumer.accept("Hello");
```

Method References

- Instance MRs – examples for unbound receiver

```
// Using Lambda Expression
Function<Person, String> fNameFunc = (Person p) -> p.getFirstName();
// Use Instance MR
Function<Person, String> fNameFunc = Person::getFirstName;
```

- **The syntax is the same as the syntax for a method reference to a static method;**
  - **Clarify: Look at the method name and check whether it is a static or instance**
- **Which object is the receiver of the instance method invocation?**
  - **Clarify using the rule: the first argument to the function represented by the target type is the receiver of the method invocation**

```
Function<String, Integer> strLengthFunc = String::length;

String name =“Popescu"; // name is the receiver of String::length
int len = strLengthFunc.apply(name);

System.out.println("name = " + name + ", length = " + len);
```

Method References

- Constructor references
  - Body of a lambda expression may be an object creation expression

```
ClassName::new
ArrayTypeName::new
```

    - Keyword new refers to the constructor of the class;
    - A class may have multiple constructors
    - The compiler selects a specific constructor based on the context
      - target type and the number of arguments in the abstract method of the target type

```
Supplier<String> func1 = () -> new String();
Function<String,String> func2 = str -> new String(str);

// Re-write these statements by replacing Lambda Ex with constructor references:
Supplier<String> func1 = String::new;
Function<String,String> funct2 = String::new;
```