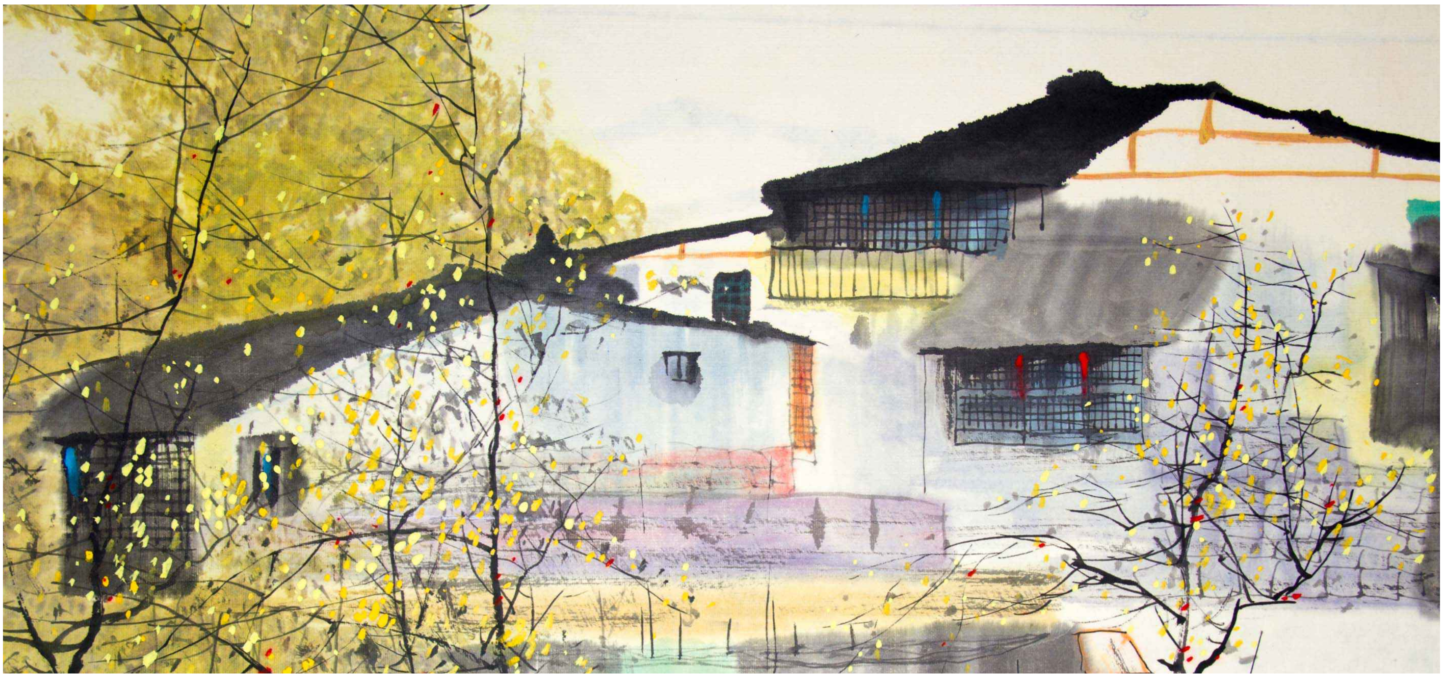


30-Java 8: Lambda Expressions

Author: Vincent Lau

Note: This material is intended for educational purposes only. All rights reserved. Any unauthorized sharing or copying of this material, in any form, to any individual or party, for any use without prior permission, is strictly prohibited.

Learning Objectives

> Understand how a lambda expression works

> List the built-in 戈 interfaces

> Write lambda expressions for each built-in functional interface

> Identify the correct built-in functional interface given the number of parameters, return type, and method name and vice versa

> List common methods which use lambdas as arguments

> Use default methods to combine lambda expressions

> Use method references to shorten lambda expressions

What is a Lambda Expression?

- ***Lambda expressions*** was first introduced in **Java 8 (1.8).**

What we learned before:

```
interface Age {
    int x = 21; // implicit public static
    void getAge();
    //default or/and static method ... which include implementation
}

// The old way to implement the methods of Age Interface
class WhatWeLearnedBefore implements Age {

    @Override
    public void getAge() {
        // printing the age
        System.out.print("Age is " + x); // implementation
    }
}

class Week4 {
    public static void main(String[] args) {
        // Myclass is implementation class of Age interface
        WhatWeLearnedBefore obj = new WhatWeLearnedBefore();
        // calling getage() method implemented at Myclass
        obj.getAge();
    }
}
```

- An example to give you a quick idea of what Labdma is.

```
interface FunctionalInterface {
    // only one abstract method
    void getAge(int a);

    // A non-abstract (or default) function
    default void sayHello() {
        System.out.println("Hello");
    }
}

//
class Dog implements FunctionalInterface {
    String name;

```

```java
    @Override
    public void getAge(int x) {
        System.out.println(2 * x);
    }
}

class Test {
    public static void main(String args[]) {
        // lambda expression to implement above
        // functional interface.
        // Also, we can write x -> System.out.println(2 * x);
        FunctionalInterface fobj = (int x) -> System.out.println(2 * x);

        // This calls above lambda expression and prints 10.
        fobj.getAge(5); // 10
        fobj.sayHello(); // default method
    }
}
```

- A *lambda expression* is a function which can be created without belonging to any class. It is thus an alternative to writing full-blown functions.

```java
interface FunctionalInterface {
    // a method that accepts two integers and calculates the sum
    int sum(int a, int b);
}
class Test {
    public static void main(String args[]) {
        // lambda expression with 2 parameters
        FunctionalInterface fobj = (a, b) -> {
            int c = a + b;
            // if more than 1 line of code inside the codeblock,
            // we need to have keyword "return"
            return c / 2;
        };

        FunctionalInterface fobj2 = (a, b) -> a + b / 2; // skip "return" keyword
    }
}
```

- A *lambda expression* can be passed around as if it was an object and executed on demand. It allows functionality to be passed into methods as an argument.

```
List<String> names = Arrays.asList("Peter", "Paul", "Mary");

// print names in a for loop
for (String n : names) {
    System.out.println("Hello " + n);
}

// an equivalent one-liner by passing a lambda expression into forEach()
names.forEach(
// Lambda Expression, represent the implementation of Consumer.accept()
n -> {
        System.out.println("Hello " + n);
        System.out.println("Hello2 " + n);
}
);

/*
Hello Peter
Hello Paul
Hello Mary
Hello Peter
Hello2 Peter
Hello Paul
Hello2 Paul
Hello Mary
Hello2 Mary
*/
```

Syntax Variations

- A *lambda expression* has the following characteristics:
    - **Optional type declaration**
    - **Optional parentheses around a single parameter**
    - **Optional curly braces for one-liner**
    - **Optional return keyword for one-liner**
- A lambda expression can be written in various ways:

```
// No parameter, return int or String
() -> 12
() -> "abc"
```

```java

// One parameter, return its square
// Optional parentheses around a single parameter
// Optional curly braces for one-liner
// Optional return keyword for one-liner
x -> x * x
// (x) -> x * x
// (int x) -> x * x

// Two parameters, return their difference
(x, y) -> x - y

// One parameter with explicit type, print its value with no return value
// (String s) -> System.out.println("Hello " + s)
s -> System.out.println("Hello " + s)

// Two parameters with explicit type, return their sum
(int x, int y) -> x + y

// One parameter with explicit type, a method body enclosed in curly braces
// and an explicit return statement
public interface interfaceName {
    String print(int x);
}
// main
n -> {
    StringBuilder sb = new StringBuilder();
    sb.append("[");
    for (int i = 0; i < n; i++) {
        sb.append(i);
        if (i < n - 1) {
            sb.append(", ");
        }
    }
    sb.append("]");
    return sb.toString();
};
```

Example

- ***Lambda expressions*** are used primarily to **represent the instance of a functional interface** (i.e. an interface with a single abstract method).

- ***Lambda expression*** **eliminates the need to create anonymous classes** in order to implement a functional interface.

```java
@FunctionInterface
interface MathOperation {
    int compute(int a, int b);

    default static String method() {
        return "abc";
    }
}

@FunctionalInterface
interface ChatBox {
    void speak(String message);
}

public class MathOperationTester {
    // op.compute(a, b), polymorphism.
    // How to implement calculate method if we don't use MathOperation as parameter
    private static int calculate(int a, int b, MathOperation op) {
        return op.compute(a, b);
    }

    public static void main(String[] args) {
       // Anonymous class to implement the MathOperation interface
       MathOperation subtractionWithoutLambda = new MathOperation() {
            @Override
            public int compute(int a, int b) {
                return a - b;
            }
        };
        // Explicit types & No return keyword, still fine
        MathOperation subtraction = (int a, int b) -> a - b;

        // Implicit lambda expression, No parameter type
        // No return keyword, most common syntax
        MathOperation addition = (a, b) -> a + b;

        MathOperation average = (a, b) -> (a + b) / 2;
        MathOperation max = (a, b) -> a > b ? a : b;

        // Explicit types & With return keyword, still fine
        MathOperation multiplication = (int a, int b) -> { return a * b; };

        // Lambda expression passed in as a method argument
        System.out.println("12 + 4 = " + calculate(12, 4, addition));
        System.out.println("12 - 4 = " + calculate(12, 4, subtraction));
        System.out.println("12 * 4 = " + calculate(12, 4, multiplication));

        // Lambda expression without a reference variable
        System.out.println("12 / 4 = " + calculate(12, 4, (a, b) -> a / b));

        // No parentheses for single param, most common syntax
        ChatBox chatBox1 = message -> System.out.println("Hello " + message);
        // With parentheses
        ChatBox chatBox2 = (message) -> System.out.println("Goodbye " + message);
        chatBox1.speak("Peter");
        chatBox1.speak("Mary");
    }
}

/*
12 + 4 = 16
12 - 4 = 8
12 * 4 = 48
12 / 4 = 3
Hello Peter
Hello Mary
*/
```

Functional Interfaces

- A *functional interface* is an interface that contains **ONLY one abstract method**.
- A *functional interface* can have **any number of default and static methods**.
- It's **recommended** to add the `@FunctionalInterface` annotation to declare a *functional interface.* This clearly communicates the purpose of the interface, and also **allows a compiler to generate an error if the annotated interface does not satisfy the conditions**.
- Let's start with **built-in functional interfaces** supported in Java.

Function and *BiFunction*

- The most simple and general use case of a *lambda*. A *Function* receives one value and returns one value. It is defined as:

```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

- A ***BiFunction*** receives two values and returns one value. It is defined as:

```java
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

Example of *Function*

- One of the usages of the *Function* type in the standard library is the ***`Map.computeIfAbsent()`*** method. This method **returns a value from a map by key, but calculates the value if the key is not already present in the map**. To calculate the value, the method uses the *Function* implementation passed in from the method argument:

```java
Map<String, Integer> nameLengthMap = new HashMap<>();

// "String::length" equivalent to "s -> s.length()"
// Function<String, Integer>, Integer is return type, while String is input type
Function<String, Integer> computeLength = t -> t.length(); // String::length;
// Why computeLength will calculate length of "Peter"? Pls check source of Map.computeIfAbsent
// computeLength is a contract for map.computeIfAbsent to execute the behavior according to your logic
// So, means you can control part of the behavior of the method map.computeIfAbsent
Integer computedValue = nameLengthMap.computeIfAbsent("Peter", computeLength);

System.out.println(computedValue);  // prints 5
```

- *HashMap* is empty, the key "Peter" does not exist inside the map initially. We calculate the value by applying the *function* to the key (i.e. "Peter"), put the computed value into the map, then return from the method call.

- For the *function* implementation, when **the parameter type and the number of parameters are matched between the input and output parameters**, we may **replace the *lambda expression* with a *method reference expressed in*** `::` **.**

```
// 源碼 of computeIfAbsent
default V computeIfAbsent(K key,
      Function<? super K, ? extends V> mappingFunction) {
    // 判断这个映射函数是否为空，为空抛出NullPointerException
    Objects.requireNonNull(mappingFunction);
    V v;
    // 判断指定的额key对应的val是否为空
    if ((v = get(key)) == null) {
      V newValue;
      // 判断映射函数计算出来的newValue是否为空
      if ((newValue = mappingFunction.apply(key)) != null) {
        // 不为空将这个key和计算出来的val存到集合
        put(key, newValue);
        // 并返回这个计算出来的val
        return newValue;
      }
    }
    // 判断指定的额key对应的val不为空直接返回val
    return v;
}
```

Example of *BiFunction*

- The ***`Map.merge()`*** method **returns a value from a map by key, but merges the existing value with the new value if the key already exists in the map**. To merge the value, the method uses the *BiFunction* implementation passed in from the method argument:

```
Map<String, Integer> nameFrequencyMap = new HashMap<>();
nameFrequencyMap.put("Peter", 2);

// equivalent to (oldValue, newValue) -> oldValue + newValue
BiFunction<Integer, Integer, Integer> mergeFrequencyByName = (x, y) -> x * y;
//Integer::sum;
Integer mergedValue = nameFrequencyMap.merge("Peter", 3, mergeFrequencyByName);

System.out.println(mergedValue); // 6
```

- Again, the *BiFunction* implementation can be replaced with a method reference because **the parameter type and the number of parameters are matched between the input and output parameters.**

```
// 源碼 of Map.merge()
default V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction) {
     Objects.requireNonNull(remappingFunction);
     Objects.requireNonNull(value);

     V oldValue = get(key);
     V newValue = (oldValue == null) ? value :
             remappingFunction.apply(oldValue, value);

     if(newValue == null) {
        remove(key);
     } else {
        put(key, newValue);
     }
     return newValue;
}
```

UnaryOperator and *BinaryOperator*

- `UnaryOperator` and `BinaryOperator` are a special case of *Function and BiFunction.*
- They **require all parameters to be the same type**, thus:
    - A *UnaryOperator* transforms its value into one of the same type
    - A *BinaryOperator* merges two values into one of the same type

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {}

@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {}

// This means their method signatures are the same as that of Function and BiFunction
T apply(T t);

T apply(T t1, T t2);
```

Example

```java
// same as x -> x.toUpperCase()
UnaryOperator<String> strUpperCase = String::toUpperCase;
System.out.println(strUpperCase.apply("rawr"));  // prints "RAWR"

// same as (x, toAdd) -> x.concat(toAdd)
BinaryOperator<String> strConcat = String::concat;
System.out.println(strConcat.apply("hello ", "world"));  // prints "hello world"
```

Supplier

- A `Supplier` is used when you want to generate or supply values without taking any input, which is defined as:

```java
@FunctionalInterface
public interface Supplier<T> {
     T get();
}
```

Example

```java
Supplier<Integer> randomInteger = () -> new Random().nextInt(10) + 1;
System.out.println(randomInteger.get());  // returns any number between 1 and 10

// equivalent to Supplier<LocalDate> dateOfToday = () -> LocalDate.now();
Supplier<LocalDate> dateOfToday = () -> LocalDate.now();
System.out.println(dateOfToday.get());  // returns the date of today
```

Consumer and BiConsumer

- You use a `Consumer` when you want to do something with one **parameter but not return anything** (also known as a *side effect*). `BiConsumer` does the same thing except that it takes **two parameters**.

```java
@FunctionalInterface
public interface Consumer<T> {
    void accept(T, t);
}

@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u);
}
```

Example

- The `Collection.forEach()` method performs some action for each element in a collection in an iteration. The action is specified by the *lambda* that is passed to the method which implements the *Consumer* interface.
- Note that `ConcurrentModificationException` will still be thrown if any structural changes are made to the collection during the iteration (e.g. adding or removing elements).

```java
List<String> names = Arrays.asList("Peter", "Paul", "Mary");

// lambda passed in as *Consumer*
Consumer<String> printName = name -> System.out.println("Hello " + name);
printName.accept("World"); // Hello World
names.forEach(printName); // list.forEach(lambda expression)

Map<String, Integer> ages = new HashMap<>();
ages.put("Peter", 18);
ages.put("Paul", 19);
ages.put("Mary", 20);

// lambda passed in as BiConsumer
BiConsumer<String, Integer> printNameAndAge = (name, age) ->
                System.out.println(name + " is " + age + " years old");
ages.forEach(printNameAndAge);

/*
Hello Peter
Hello Paul
Hello Mary
Peter is 18 years old
Paul is 19 years old
Mary is 20 years old
*/
```

Predicate

- A ***Predicate*** is a function that receives a value and returns a boolean value.
- It is often used with filtering or matching, e.g. *Collection.removeIf().* Something like conditional statement.
- *A* BiPredicate does the same thing except it takes two parameters.

```
@FunctionalInterface
public interface Predicate<T> {
        boolean test(T t);
}

@FunctionalInterface
public interface BiPredicate<T, U> {
        boolean test(T t, U u);
}
```

Example - *Predicate*

```
// Naming Conversion: names (end with "s") for List. Just a norm.
// Technically you can use nameList also
List<String> names = Arrays.asList("Alex", "Amy", "Ben", "Charlotte", "Dicky");

//
Predicate<String> startsWithA = name -> name.startsWith("A");
Predicate<String> lengthLongerThan10 = name -> name.length > 10;
Predicate<String> elementIsAlex = name -> "Alex".equals(name);

if (startsWithA.test("Alex")) {
    System.out.println("Yes");
}

// lambda passed in to filter() as Predicate
List<String> filteredNames = names.stream()
                                .filter(s -> s.length > 5 && s.startsWith("A"))
                                .collect(Collectors.toList());

/*
Hello Alex
Hello Amy
*/
```

Example - *BiPredicate*

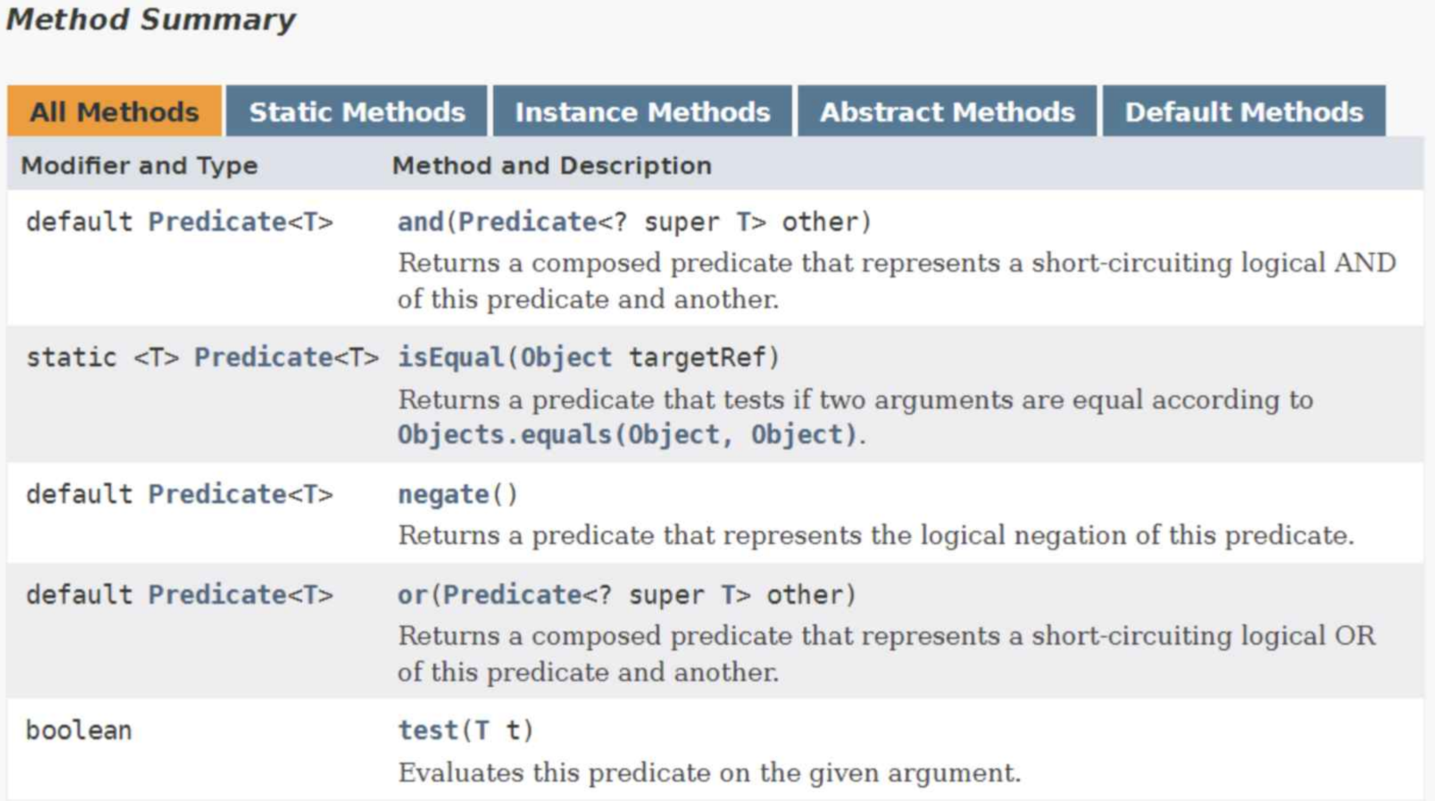
- Useful for writing test cases. We will cover it in the later section.

```
// equivalent to String::startsWith
BiPredicate<String, String> startsWithPrefix
                            = (s, prefix) -> s.startsWith(prefix);

System.out.println(startsWithPrefix.test("World", "Hello")); // prints false
System.out.println(startsWithPrefix.test("Hello", "Hello")); // prints true
System.out.println(startsWithPrefix.test("HelloWorld", "Hello")); // prints true
System.out.println(startsWithPrefix.test("", "Hello")); // prints false
System.out.println(startsWithPrefix.test(null, "Hello")); // NullPointerException
```

Combining Predicates

- The ***`Predicate`*** interface provides *default* and *static helper* methods for us to combine multiple predicates by chaining.



Method Summary

Modifier and Type	Method and Description
`default Predicate<T>`	`and(Predicate<? super T> other)` Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
`static <T> Predicate<T>`	`isEqual(Object targetRef)` Returns a predicate that tests if two arguments are equal according to `Objects.equals(Object, Object)`.
`default Predicate<T>`	`negate()` Returns a predicate that represents the logical negation of this predicate.
`default Predicate<T>`	`or(Predicate<? super T> other)` Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
`boolean`	`test(T t)` Evaluates this predicate on the given argument.

- The ***`and()`*** method allows us to create a new *Predicate* by combining two existing *Predicates*:

```
Predicate<String> startsWithV = a -> a.startsWith("V");
Predicate<String> endsWithT = x -> x.endsWith("T");

```

```java
Predicate<String> combined = startsWithV.and(endsWithT);
Predicate<String> combined2 = startsWithV.or(endsWithT);

System.out.println(combined.test("VINCENT")); // prints true
System.out.println(combined2.test("OINCENM")); // prints false
```

Combining Comparators

- We can even combine *Comparators* by using *Comparator's default* method *thenComparing()*.
- Recall the *Customer* class we have from an earlier session:

```java
public class Customer {
    private int id;
    private String name;
    private LocalDate joinDate;
    // constructors + getters ...
}

Comparator<Customer> sortByName = (c1, c2) ->
              c1.getName().compareTo(c2.getName());

Comparator<Customer> sortByJoinDate = (c1, c2) ->
        c1.getJoinDate().compareTo(c2.getJoinDate());

Comparator<Customer> sortByNameAndJoinDate = sortByName.thenComparing(sortByJoinDate);
```

Method References

- *Method References* provide a more concise way to express a *lambda expression.*
- Method references can be used for the following types of methods:
  - **Static methods**
  - **Instance methods**

Example

```java
public class MethodReferenceExample {
    public static void main(String[] args) {
        List<String> namesList = new ArrayList<>();
        namesList.add("Alex");
        namesList.add("Benny");
        namesList.add("Carl");

        // method reference as static method
        List<String> helloNamesList = namesList.stream()
                .map(MethodReferenceExample::process)
                // .map(e -> MethodReferenceExample.process(e))
                .collect(Collectors.toList());

        System.out.println(helloNameList); // ["hello Alex", "hello Benny", "hello Carl"]
        // method reference as instance method
        List<String> uppercaseNamesList = namesList.stream()
                .map(String::toUpperCase)
                // .map(e -> e.toUpperCase)
                .collect(Collectors.toList());

        System.out.println(uppercaseNamesList); // ["ALEX", "BENNY", "CARL"]
    }

    private static String process(String s) {
        return "hello " + s;
    }
}
```

Constructor References

- `Constructor Reference` is used to refer to a constructor **without instantiating the named class**.

```java
@FunctionalInterface
interface EmployeeEmpty {
    Employee get();
}

@FunctionalInterface
interface EmployeeWithName {
    Employee get(String name);
}

class Employee {
    private String name;

    Employee() {
        System.out.println("Empty Constructor");
    }

    Employee(String name) {
        System.out.println("Create constructor with name");
        this.name = name;
    }

    public String toString() {
        return "name: " + name;
    }
}

// main
EmployeeEmpty empEmpty = Employee::new; // it actually doesn't call constructor
// EmployeeEmpty empEmpty = () -> new Employee();
System.out.println("Constructor isn't called yet");
System.out.println(empEmpty.get()); // instead, call constructor here

EmployeeWithName empWithName = Employee::new; // it actually doesn't call constructor
// EmployeeEmpty empEmpty = () -> new Employee(String s);
System.out.println("Constructor isn't called yet");
System.out.println(empWithName.get("VenturenixLAB").toString()); // instead, call constructor here

// Output
// Constructor isn't called yet
// Empty Constructor
// Constructor isn't called yet
// Create constructor with name
// name: VenturenixLAB
```

Questions

- Express a lambda expression in various syntax variations.
- What are the **built-in functional interfaces** supported in Java? Give an example of each interface.

- Write lambda expressions to work with common methods such as:
  - *Map.merge()*
  - *Map.computeIfAbsent()*
  - *Collections.sort()*
  - *Collections.forEach()*
  - *Collections.removeIf()*