**Lambda Expressions**

Think of a Lambda expression as a shorthand for an anonymous class that implements a functional interface – an interface that contains only a single abstract method. This makes your code more concise and easier to understand.

Lambda expressions allow you to pass blocks of code as parameters, offering a powerful and flexible way to write cleaner and more functional code with minimal effort. These compact and expressive constructs can greatly simplify your code.

Records are static like enums and interfaces.

Lambdas can be assigned to a variable. But assigning it isn’t going to print it. We need to call the method that prints the lambda and assign it there for example:

BiConsumer<Double,Double> p1 =

(lat, lng) -> System.***out***.printf(**"[lat:%.3f lon:%.3f]%n"**, lat,lng);  
**var** firstPoint = coords.get(0);  
*processPoint*(firstPoint[0], firstPoint[1], p1);//print 1st coordinate

System.***out***.println(**"--------"**);  
  
coords.forEach(s -> *processPoint*(s[0],s[1], p1));//prints all the coordinates

*//creating a generic method without the return type with BiConsumer***public static** <T> **void** processPoint (T t1, T t2, BiConsumer<T,T> consumer) {  
 consumer.accept(t1,t2);  
}

**The Lambda Expression**

(o1, o2) -> o1.lastName().compareTo(o2.lastName()));

Syntax: (paramter1, paramter2,…) -> expression;

For a lambda expression, **the method is inferred by Java**. Java takes its clue from the reference type, in the context of the Lambda expression usage.

**The Functional Interface:**

A functional Interface is an interface that has one, and only one, abstract method. This is how java can infer the method to derive the parameters and return type, for the Lambda expression.

We can use @FuntionalInterface for functional methods. Java provides a library of functional interfaces in the java.util.function package. One of them is Consumer interface. The other one is the Binary Operator.

**Lambda use in foreach loop:**

List.foreach (element -> System.out.println(element)) //valid

List.foreach ((element) -> System.out.println(element)) //valid

List.foreach ((String element) -> System.out.println(element)) //valid

List.foreach ((var element) -> System.out.println(element)) //valid

**int** result = *calculator*((**var** a, **var** b) -> {**return** a + b;}, 5, 2);// valid

**int** result = *calculator*((**var** a, **var** b) -> {**var** c = a + b; **return** c;}, 5, 2);

**Use of Lmabda expression variations, the Lambda body:**

list.forEach((**var** myString) -> System.***out***.println(myString));//valid

list.forEach((**var** myString) -> {  
 **char** first = myString.charAt(0);  
 System.***out***.println(myString + **" means "** + first);  
}); //valid

**The Four categories of Functional Interfaces:**

It’s a good idea to know the four basic types of functional interfaces in the java.util.function package. There are over forty interfaces in this package. The following table shows the four categories, with the simplest method shown.

|  |  |  |
| --- | --- | --- |
| **Interface Category** | **Basic Method Signature** | **Purpose** |
| Consumer | Void accept (T t) | Execute code without returning data |
| Function | R apply (T t) | Return a list of an operation or function |
| Predicate | Boolean test (T t) | Test if a condition is true or false |
| Supplier | T get () | Return an instance of something |

**The Consumer Interface:**

The Consumer interface is in the Jave.util.function Package.

It has one abstract method that takes a single argument and doesn’t return anything.

**Void accept (T t)**

The two most common Cosnumer interfaces, and the functional method are the following

The Consumer interface takes one argument of any type.

The BiConsumer interface takes two arguments of two different types

|  |  |
| --- | --- |
| Interface Name | Method Signature |
| Consumer | Void accept (T t) |
| BiConsumer | Void accept (T t, U u) |

**Example Lambda Expression for Consumer and Consumer Method:**

S -> System.out.println(S); void accept (T t);

**The Function Interface:**

The functional interface has a return type, shown below as either T or R which stands for result, meaning a result is expected for any of these. In addition to Function and BiFunction, there is also UnaryOperator and Binary Operator

|  |  |  |  |
| --- | --- | --- | --- |
| Interface Name | Method Signature | Interface Name | Method Signature |
| Function<T, R> | R apply(T t) | UnaryOperator<T> | T apply (T t) |
| BiFunction<T, U, R> | R apply (T t, U u) | BinaryOperator<T> | T apply (T t1, T t2) |

Example:

//public static method with three arguments, instance of interface Operations and two values

public static <T> T calculator(BinaryOperator<T> functions, T value1, T value2){ //instead of using our own interface we are using BinaryOperator.

T result = functions.apply(value1,value2); //The functional method name of the Binary Operator is "apply"

System.out.println("Result of operation: " + result);

return result;

int result = calculator((var a, var b) -> {var c = a + b; return c;}, 5, 2);

var result2 = calculator((a, b) -> a/b, 10.0, 2.5);

var result3 = calculator(

(a,b) -> a.toUpperCase() + " " + b.toUpperCase(), "Ralph", "Kramden"

);

**The Predicate Interface**

The predicate interfaces take one or two arguments, and always returns a Boolean value. They are used to test a condition, and if the condition is true to perform an operation.

|  |  |
| --- | --- |
| Interface name | Method Signature |
| Predicate | Boolean test (T t) |
| BiPredicate | Boolean test (T t, U u) |

A predicate Lambda Expression Example:

In the following example, the expression takes a String, and tests if its equal to the literal text “Hello”, ignoring case. It returns either true or false

Example Lambda Expression for Predicate

S -> S.equalsIgnoreCase(“Hello”);

**The Supplier Interface:**

The supplier interface takes no arguments but returns an instance of some type T.

|  |  |
| --- | --- |
| Interface Name | Method Signature |
| Supplier | T get () |

Example-> () -> random.nextInt(1,100)

Some of the list interfaces are mentioned below. These wont work on a list backed by an array.

**List.removeif():**

Used to remove contents if the list.

list.removeIf(s -> s.equalsIgnoreCase(**"bravo"**));//using it with Lambda

//removing contents of the list starting with “ea”

list.addAll(List.*of*(**"echo"**, **"easy"**, **"earnest"**));  
list.forEach(s -> System.***out***.println(s));  
  
System.***out***.println(**"------"**);  
  
list.removeIf(s-> s.startsWith(**"ea"**));

**List.replaceAll():**

list.replaceAll(s -> s.charAt(0) + **" - "** + s.toUpperCase());

**output:**

**a - ALPHA**

**c - CHARLIE**

**d – DELTA**

**Arrays.setAll():**

*//creating an array of 10 elements*String[] emptyStrings = **new** String[10];  
System.***out***.println(**"------"**);  
System.***out***.println(Arrays.*toString*(emptyStrings)); // [null, null, null, null, null, null, null, null, null, null]  
  
Arrays.*fill*(emptyStrings,**" "**);  
System.***out***.println(Arrays.*toString*(emptyStrings));// [ , , , , , , , , , ]  
  
Arrays.*setAll*(emptyStrings, (i) -> **" "** + (i + 1) + **"."**);  
System.***out***.println(Arrays.*toString*(emptyStrings));// [ 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.]

Arrays.*setAll*(emptyStrings, (i) -> **" "** + (i + 1) + **"."** + **switch** (i) {  
 **case** 0 -> **"one"**;  
 **case** 1 -> **"two"**;  
 **case** 2 -> **"three"**;  
 **default** -> **""**;  
  
 }  
);  
System.***out***.println(Arrays.*toString*(emptyStrings));// [ 1.one, 2.two, 3.three, 4., 5., 6., 7., 8., 9., 10.]

**Method Reference**

In Java, a method reference is a shorthand syntax for creating a lambda expression that calls a specific method. It allows you to directly refer to a method without explicitly writing the lambda expression body.

|  |  |
| --- | --- |
| Lambda Expression | MethodReference |
| s -> System.out.println(s) | System.out::println |

A method reference abstracts the Lambda expression even further, eliminating the need to declare formal parameters. We also don’t have to pass arguments to the method in question, in this case println. A method reference has double colons, between the qualifying type, and the method name.

In this example of a Consumer interface, not only is the method inferred, but the parameter are as well.

Method which can be used as method references are based on the context of the Lambda expression.

This means the method reference, is again dependent on the targeted interface’s method. You can reference a static method on a class.

You can reference an instance method from either an instance external to the expression, or an instance passed as one of the arguments.

Or you can reference a constructor by using new as the method. Method references can be used to increase the readability of your code.

**Terminology for the Reference Type**

A Type reference refers to a class name, an interface name, an enum name, or a record name.

Static methods are usually called using Type references but can also be called by instances in our code.

This is NOT true for method references. Static methods, in method references and Lambda expressions, must be invokes using a reference type only.

**Bounded and Unbounded in JAVA**

In Java, the terms "bounded" and "unbounded" receiver are primarily used in the context of method references.

Bounded Receiver:

* A bound method reference refers to a non-static method that is associated with a specific object (the receiver).
* The syntax for a bound method reference is object::methodName.

Example:

class MyClass {

public void printMessage() {

System.out.println("Hello from MyClass");

}

}

public class Main {

public static void main(String[] args) {

MyClass obj = new MyClass();

Runnable runnable = obj::printMessage; *// Bound method reference*

runnable.run();

}

}

Unbounded Receiver:

* An unbounded method reference refers to a non-static method that is not associated with a specific object.
* The syntax for an unbounded method reference is ClassName::methodName.

Example:

class MyClass {

public void printMessage(String msg) {

System.out.println(msg);

}

}

public class Main {

public static void main(String[] args) {

Consumer<String> consumer = MyClass::printMessage; *// Unbounded method reference*

consumer.accept(“Hello from Main”);

}

}

Key Differences:

* **Bounded:** The method reference is tied to a particular object. When the method is invoked through the reference, it’s called on that specific object.
* **Unbounded:** The method reference is not tied to a particular object. When the method is invoked through the reference, the object it’s called on is supplied as an argument.

Use Cases:

* **Bounded:** Useful when you want to repeatedly call a method on the same object.
* **Unbounded:** Useful when you want to call a method on different objects dynamically.

**Convenience Methods**

These are default methods on some of the functional interfaces. The Consumer, Predicate, and Function interfaces all come with these methods, as does the Comparator.

|  |  |  |
| --- | --- | --- |
| Category of Interface | Convenience method example | Notes |
| Function | Function.andThen(Function2) | Not implemented on IntFunction, DoubleFunction, Long Funtion |
| Function | Function2.compose(function1) | Only implemented on Function & UnaryOperator |
| Consumer | Consumer1.andThen(consumer2) |  |
| Predicate | Predicate1.and(predicate2) |  |
| Predicate | Predicate1.or(predicate2) |  |
| Predicate | Predicate1.negate() |  |

For andThen, and Compose, on the Function category of interfaces, any interim functions are not required to have the same type arguments.

Instead, one function’s output becomes the next function’s input, and the next function’s output is not constrained to any specific type, except the last function executed in the chain.

**Comparator’s additional helper methods**

|  |  |
| --- | --- |
| Type of Method | Method Signature |
| static | Comparator comparing(Function keyExtractor) |
| static | Comparator naturalOrder() |
| static | Comparator reverseOrder() |
| default | Comparator thenComparing(Comparator other) |
| default | Comparator thenComparing(Function keyExtractor) |
| default | Comparator reversed() |