A lambda expression is an anonymous function (not 100% true for Java but lets assume it for time being). Simply put, it’s a method without a declaration, i.e., access modifier, return value declaration, and name.

* A lambda expression can have zero, one or more parameters.
* The type of the parameters can be explicitly declared or it can be inferred from the context. e.g. (int a) is same as just (a)
* Parameters are enclosed in parentheses and separated by commas. e.g. (a, b) or (int a, int b) or (String a, int b, float c)
* Empty parentheses are used to represent an empty set of parameters. e.g. () -> 42
* When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a\*a
* The body of the lambda expressions can contain zero, one or more statements.
* If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
* When there is more than one statement in body than these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

Lamda Expression will replace anonymous inner classes

Zero parameter Lamba expression:

() -> System.out.println("Zero parameter lambda");

//Before Java 8: **new** **Thread**(**new** Runnable() {

**@Override** **public** **void** **run**() {

System.out.println("Before Java8, too much code for too little to do"); } }).start();

One parameter LAmba Expression

(param) -> System.out.println("One parameter: " + param);

Multiple parameter Lambda Expression:

(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

(Car car) -> System.out.println("The car is: " + car.getName());

**Returning a Value From a Lambda Expression**

(param) -> {

System.out.println("param: " + param);

return "return value";

}

(a1, a2) -> { return a1 > a2; }

Return is optional:

(a1, a2) -> a1 > a2;

//Java 8 way: **new** **Thread**( () -> System.out.println("In Java8, Lambda expression rocks !!") ).start();

Syntax:

(params) -> expression  
(params) -> statement  
(params) -> { statements }  
  
for example, if your method don't change/write a parameter and just print something on console, you can write it like this :

() -> **System**.out.println("Hello Lambda Expressions");

If your method accept two parameters then you can write them like below :  
  
(int even, int odd) -> even + odd  
  
// Before Java 8:

JButton show = **new** JButton("Show");

show.addActionListener(**new** ActionListener() {

**@Override**

**public** **void** **actionPerformed**(ActionEvent e) {

System.out.println("Event handling without lambda expression is boring");

}

});

**Iterating over List using Lambda expressions**

// Java 8 way:

show.addActionListener((e) -> {

System.out.println("Light, Camera, Action !! Lambda expressions Rocks");

});

//Prior Java 8 :

**List** features **=** **Arrays**.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");

**for** (**String** feature **:** **features**) { **System**.out.println(feature);

} //In Java 8:

**List** features **=** **Arrays**.asList("Lambdas", "Default Method", "Stream API", "Date and Time API");

features.forEach(n -> **System**.out.println(n));

// Even better use Method reference feature of Java 8 // method reference is denoted by :: (double colon) operator // looks similar to score resolution operator of C++

features.forEach(**System**.out::println);

class Demo {

**public** **static** **void** **main**(args[]){

List languages = Arrays.asList("Java", "Scala", "C++", "Haskell", "Lisp"); System.out.println("Languages which starts with J :");

filter(languages, (str)->str.startsWith("J"));

System.out.println("Languages which ends with a ");

filter(languages, (str)->str.endsWith("a"));

System.out.println("Print all languages :");

filter(languages, (str)->**true**);

System.out.println("Print no language : ");

filter(languages, (str)->**false**);

System.out.println("Print language whose length greater than 4:");

filter(languages, (str)->str.length() > **4**); }

**public** **static** **void** **filter**(List names, Predicate condition) {

**for**(String **name:** names) {

**if**(condition.test(name)) {

System.out.println(name + " ");

} } } }  
  
//betterway

/\*

**public** **static** **void** **filter**(List names, Predicate condition)

{ names.stream().filter((name) -> (condition.test(name))).forEach((name) -> { System.out.println(name + " ");

}); }

\*/

Map()

This example is about one of the popular functional programming concept called map. It allows you to transform your object. 

Reduce()

reduce is not a new operation,  you might have been already using it. If you can recall SQL aggregate functions like sum(), avg() or count(), they are actually reduce operation because they accept multiple values and return a single value. Stream API defines reduce() function which can accept a lambda expression, and combine all values. Stream classes like IntStream has built-in methods like average(), count(), sum() to perform reduce operations and mapToLong(), mapToDouble() methods for transformations