JAVA 1.8 Features (for ref https://www.scaler.com/topics/java/lambda-expression-in-java/)

1. Lambda Expressions:

Explanation: Lambda expressions allow you to express instances of single-method interfaces (functional interfaces) more concisely. It facilitates the writing of more readable and maintainable code by reducing boilerplate code.

Lambda expression in java is an anonymous (no name) function that does not need to define the data type of input parameters and does not need to have a return type. Lambda expression in java implements the functional interface and it can be treated as any other java object. It can be used to create threads, comparators and can be used to add event Listeners.

Example:1

// Before Java 8

Runnable runnable = new Runnable() {

@Override

public void run() {

System.out.println("Hello, World!");

}

};

// With Lambda Expression

Runnable runnableLambda = () -> System.out.println("Hello, World!");

Example 2:-

// Before Java 8

MyInter person = new MyInter() {

@Override

public void MyInterMethod() {

System.out.println("Before Java 8 without lambda");

}

};

// Java 8: Using lambda expression

MyInter p2 = () -> {

System.out.println("Lambda implemented");

};

Example:3

@FunctionalInterface

interface MyName{

public void SayMyName();

}

public class LambdaExpressionExample {

public static void main(String[] args) {

//without lambda, MyName implementation

// using anonymous class

MyName person = new MyName(){

@Override // is an optional keyword in this case

public void SayMyName(){

System.out.println("Himanshu");

}

};

person.SayMyName();

}

}

With lambda

Example 1

@FunctionalInterface //It is optional

interface MyName{

public void SayMyName();

}

public class LambdaExpressionExample {

public static void main(String[] args) {

//implementing with lambda

MyName p2 = () -> {

System.out.println("Himanshu Yadav");

};

// function call.

p2.SayMyName();

}

}

1. Functional Interfaces:

Explanation: Functional interfaces are interfaces with only one abstract method. They are a fundamental part of lambda expressions, as they provide the target type for lambda expressions.

Example:

// Functional Interface

@FunctionalInterface

interface MyFunctionalInterface {

void myMethod();

}

1. Streams API:

Explanation: Streams provide a declarative way to process sequences of elements. They allow operations on data to be expressed concisely and can be parallelized to enhance performance.

Example:

List<String> myList = Arrays.asList("Java", "is", "awesome");

// Using Streams to filter and print elements

myList.stream()

.filter(s -> s.startsWith("a"))

.forEach(System.out::println);

1. Default Methods in Interfaces:

Explanation: Default methods allow interfaces to have method implementations. This feature enables adding new methods to interfaces without breaking existing implementations.

Example:

interface MyInterface {

void existingMethod();

default void newMethod() {

System.out.println("Default implementation of newMethod");

}

}

1. Method References:

Explanation: Method references provide a shorthand syntax for lambda expressions, making the code more concise. They can reference methods or constructors using a shorter syntax.

Example:

List<String> myList = Arrays.asList("Java", "is", "awesome");

// Using Method Reference to print elements

myList.forEach(System.out::println);

1. Optional Class:

Explanation: The Optional class is a container object that may or may not contain a non-null value. It helps prevent null pointer exceptions and encourages more explicit handling of null values.

Example:

Optional<String> optionalValue = Optional.of("Hello, World!");

String result = optionalValue.orElse("Default Value");

1. New Date and Time API:

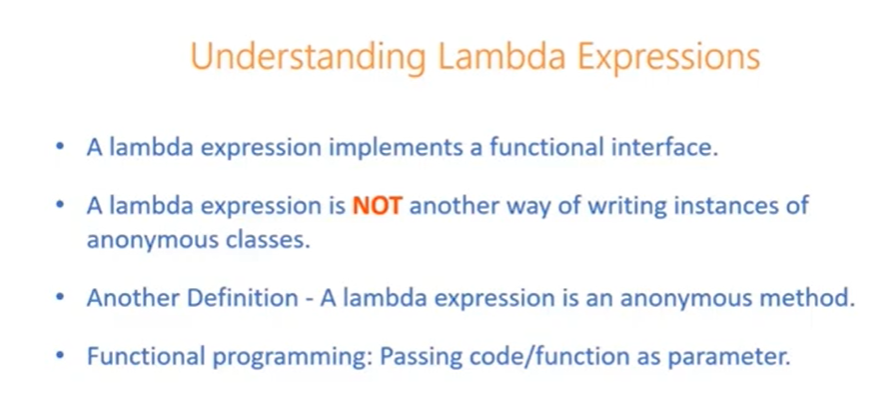
Explanation: The new Date and Time API provides a comprehensive, immutable, and thread-safe API for date and time manipulation, overcoming the shortcomings of the older Date and Calendar classes.

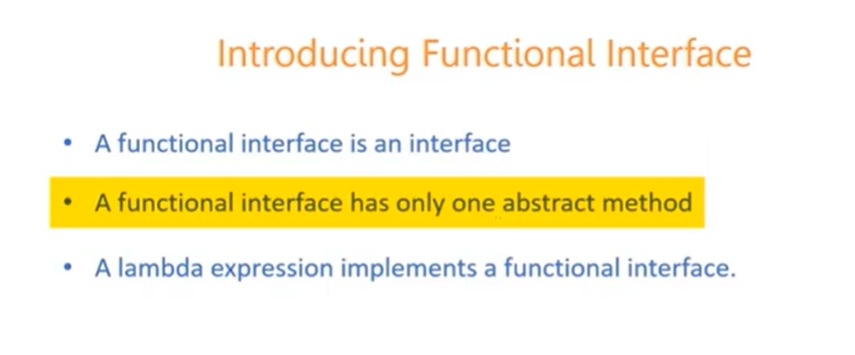
LocalDateTime now = LocalDateTime.now();

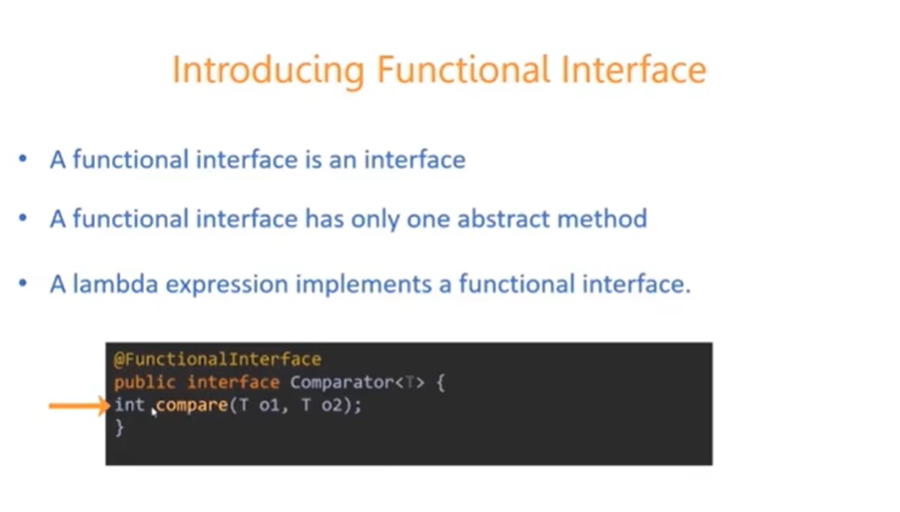
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

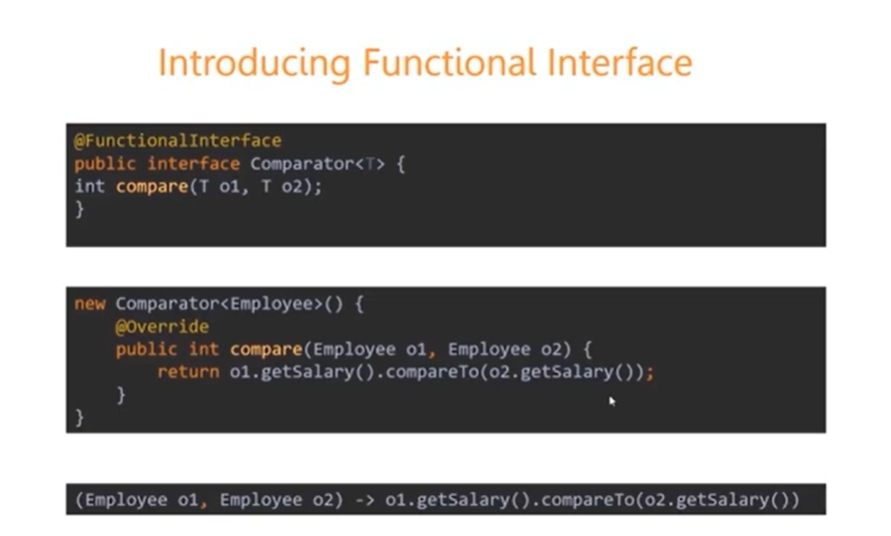
String formattedDateTime = now.format(formatter);

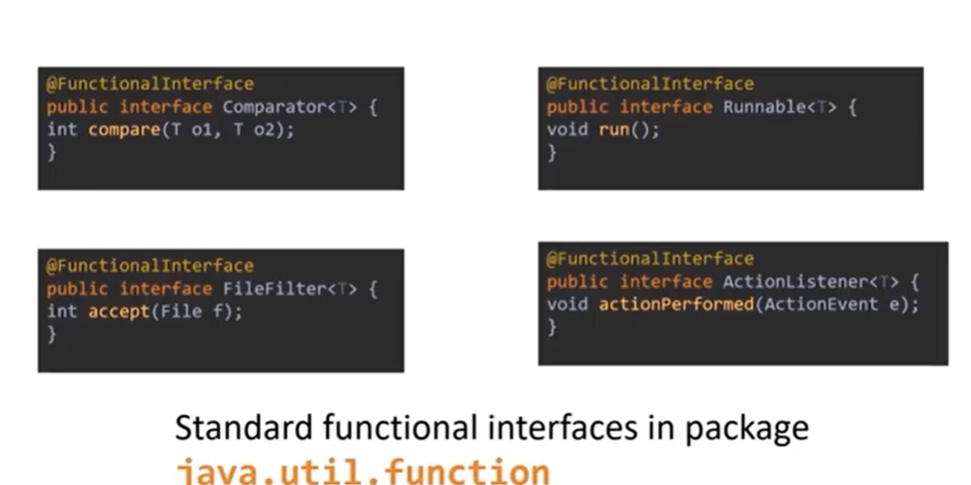
Project on lambda expression

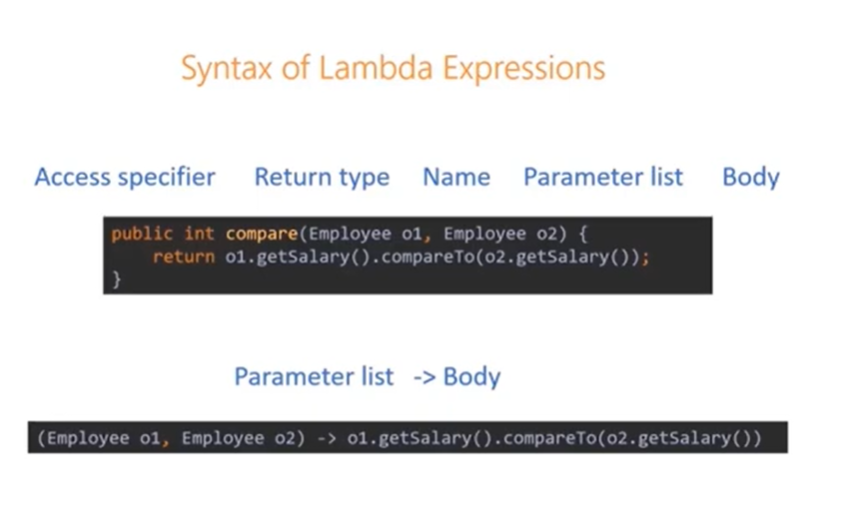


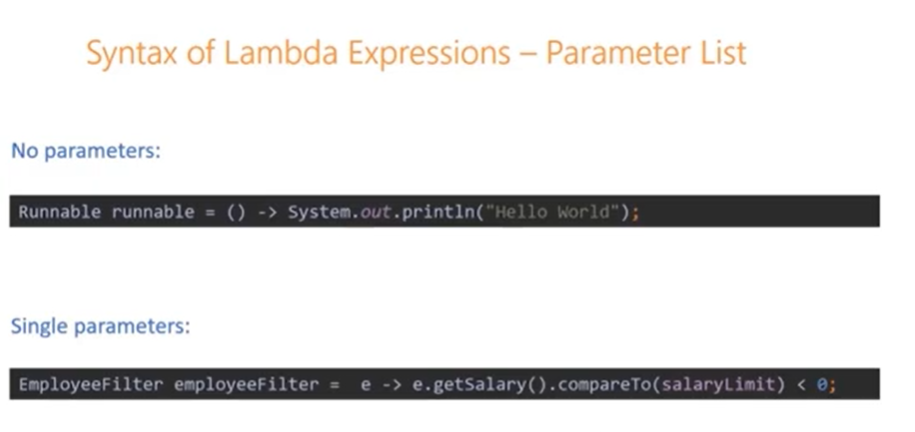






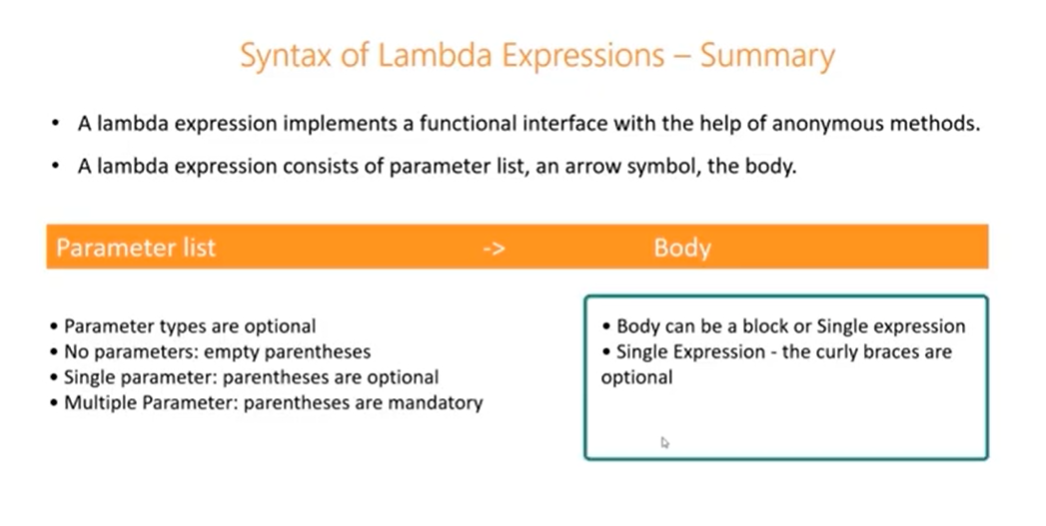


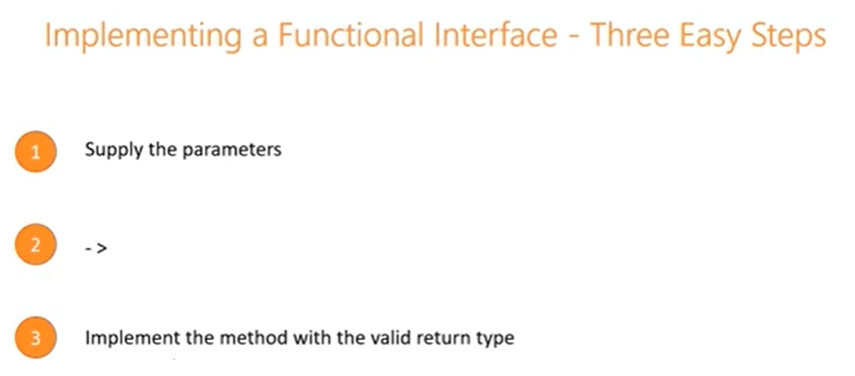


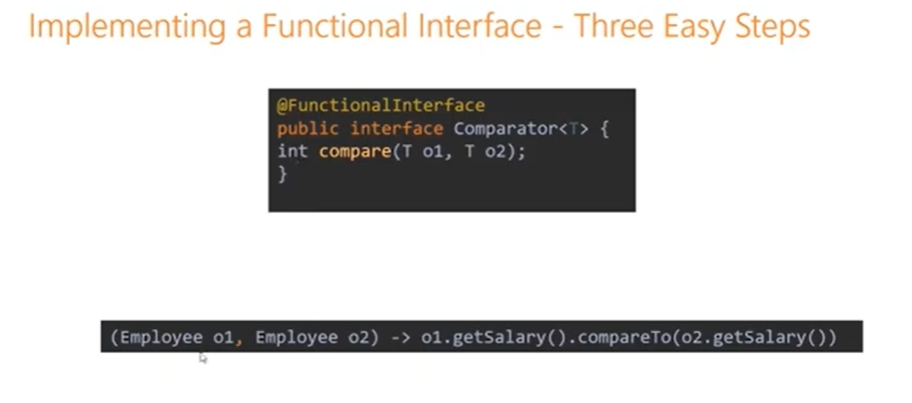


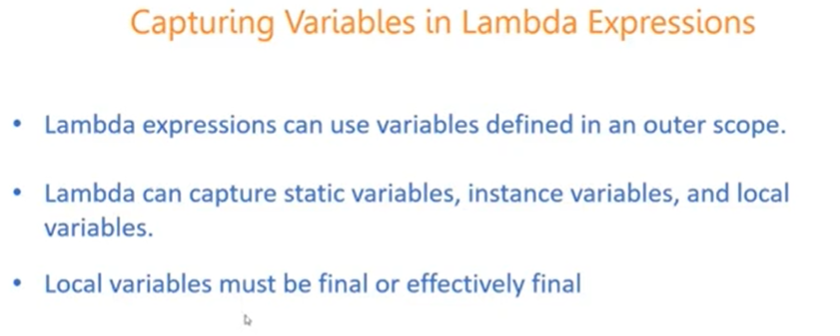


Recap









Exception Handling in lambda Expression

Understanding the method reference

When lambda expression called an existing method then we can replace this method as a method reference

Method reference is just like pointer to a methods.method reference implement functional interface and that is usefull in where you and have an existing method to replace with method refernces.

**package** com.module2Example;

**import** java.util.List;

**import** com.employee.Employee;

**import** com.employee.EmployeeData;

**public** **class** LambdaMethodReferenceExample8 {

**public** **static** **void** main(String arg[]) {

List<Employee> employees = EmployeeData.*getEmployee*();

employees.forEach(e->System.***out***.println(e));

}

}

We can revome println() method with method reference

employees.forEach(System.***out***::println); //this is method reference

