**Java Basics**

* **package** [chapter]: file structure, usually organization domain name in reverse structure. Related to specific business topic.
* **Class** [paragraph]: noun, should focused on only one topic.
* **Method/functions** [sentence]: verb, how to achieve a certain thing in class. 20 line at max. 3 to 5 parameter.
* **Variable:** place holder for any value.
* **@:** annotation.
* **Objects:** created while executing programs which value can change throughout the execution
* **Constructor:** special method to create object from class

**Maven project Structure:**

* Maven uses a set of identifiers, also called coordinates, to uniquely identify a project and specify how the project artifact should be packaged.
* groupId – a unique base name of the company or group that created the project.

<groupId>com.rockstar.digital.ms.businessactivity.creditscore</groupId>

* artifactId – a unique name of the project.

<artifactId>businessactivity-creditscore-parent</artifactId>

Once we use above identifiers, project name will be businessactivity-creditscore-parent and it can have packages having common name started with com.rockstar.digital.ms.businessactivity.creditscore

* version – a version of the project

If we are working with static variables in method, then method must be static. It’s not compulsory but it is a good practice.

**Java 8**

**Lambda:**

Pass behavior as an argument and let rest happen based on that.

**Java 7:** We can do it using Java7 as well. suppose we have an interface which have multiple implementation, so we create object for the type (behavior) that we want to achieve and call interface method with this object so it will give result based on the implementation that we have for this class. Here we can change behavior as many types, as we want (give a different implementation for all these methods), but meanwhile we need these many implementation classes. Actually, we are not passing behavior here, we are passing thing (Object) as a behavior.

**Java 8:** Lambda allow us to pass action instead of object. This is just function, doesn’t belong to a class. These functions can be treated as values. We can assign a block of code/ (function) to a variable. Variable is having a block of code not the execution result of this function. So, wherever this variable goes, this code goes with it.

As we are assigning this function to a variable (writing an isolation function) so we can skip below things from this function.

* **public :** as function is no more related to class so we can remove this access modifier.
* **Function name:** we are assigning function to a variable and later we will use this variable name to call this function, so there is no need of function name.
* **Return type:** we can see the function and find out what value it will return (so does compiler find out). That’s why there is no need to write a return type in lambda.
* **Return:** as we are not writing return type for lambda, we don’t write return identifier as well.
* **Changes in function body:** If we know that our function will have body of just one line then we can even remove { } from body.

Key points:

* We remove these many things while writing lambda function and just include only one thing that is an arrow sign **->**
* We can pass inline lambda function like we pass inline variables in java.

**Java 7 code :**

*public void* greets() {  
 System.out.println("Hello World");  
}

**Java 8 code:**

**interface for lambda:**

*interface* LambdaInterface {  
 *public void* lambdaFunction();  
}

**Implementation :**

LambdaInterface lambda = () -> System.out.println("Hello world lambda");

**Calling:**

lambda.lambdaFunction()

* Lambda Developers didn’t create a new type to declare lambda function. They used already existing Interface concepts. As we can declare methods in Interface. So, we can have a *functional interface* which has a method having similar return type as lambda has. So, we can assign this interface as a type of lambda.
* Here interface name can be anything and method describing in interface can be anything as name is not making any sense.

ICalculateStringLength LengthLambda = (String S) -> S.length();

* As we have type of lambda as interface, compiler already knows which method it is having. So basically, compiler knows that argument type in the interface method is a String so we can get rid-off from it.

ICalculateStringLength LengthLambda = S -> S.length();

* Lambda developers didn’t introduce new interface to use as lambda type because of many reasons. Most important one is **backward compatibility.** If they would have introduced a new type called lambda type/ function type. We could have used this lambda type on in other code which also use the same lambda type.

For example, if we want to use a library and passing lambda type to the library, then we have to rewrite this library to accept this function type as well.

The advantage of using interface is that, we can use lambda in place of all these anonymous inner classes and all the other method signatures which accept the interface. We don’t have to rewrite them for new function type.

**For example:**

While creating a new thread using Runnable, we have to create a new instance of Runnable and use this instance in order to create new thread. (can create using Anonymous inner class as well). Since Runnable is a functional interface, it works just as well with lambda.

So, we can use Lambda in order to create a new thread.

* While writing a functional interface we can use **@FunctionalInterface** to indicate other developer to not to make it abstract interface by adding another abstract method to it.
* Sort people based on their firstName: **Java7**

**Collections.sort(obj, object of sorting class[we can have user define sorting class as well]); or**

Collections.sort(people, *new* Comparator<Person>() {  
 *@Override  
 public int* compare(Person o1, Person o2) {  
 *return* o1.getFirstName().compareTo(o2.getFirstName());  
 }  
 }  
);

**Java8**

Collections.sort(people, (o1, o2) -> o1.getFirstName().compareTo(o2.getFirstName()));

* If we need to pass behavior as argument that means we pass behavior type (functional interface) to a method and implement it according to our requirements. This can be written as below:

*private static void* printConditionally(List<Person> l, ConditionInterface conditionInterface) {  
 *for* (Person p : l) {  
 *if* (conditionInterface.test(p))  
 System.out.println(p);  
 }  
}

now we can call this method from main method and provide implementation of this:

System.out.println("print all names start with A");  
 printConditionally(people, list -> list.getFirstName().startsWith("A"));  
  
System.out.println("print all names which contain i in last name");  
 printConditionally(people, list -> list.getLastName().contains("i"));

While writing all these methods we are using a user created Interface named as ConditionInterface.

*interface* ConditionInterface {  
 *boolean* test(Person P);  
}

Its not required to write these Interfaces as Java developers already provide different Interfaces for used cases. As we need an interface here which should have one abstract method with return type boolean so we can use **Predicate<T>** Interface. Predicate is a generic interface so we can use it according to our requirements as in this example we are sending a List of Persons as method argument, that means operation in working on Person object so we can define Predicate with Person as below.

*private static void* printConditionally(List<Person> l, Predicate<Person> conditionalInterface) {  
 *for* (Person p : l) {  
 *if* (conditionalInterface.test(p))  
 System.out.println(p);  
 }  
 }

As we made changes of Interface here, there is no changes in Lambda functions as lambda care only about type of interface and return type of method it has. Not the name.

So, we haven’t updated anything in lambda, just updated out Interface in method definition.

* Exception Handling in lambda can be achieved in multiple ways. We can achieve few as below where it is throwing Arithmetic Exception because of divide by 0.
* Whenever classA is extending classB so we can call classB static methods using classA, classB and object of classA but if both have same static methods (that mean classB static method is getting override in classA) then we can call classB static method only using classB reference. If we try to more restrict classB static methods overridden in classB then we will get error as overridden methods can’t be more strict.
* If a classA is implementing interfaceB which is having a static method, then this method can be called only with reference of interfaceB not using classA or it’s object. If we have same static method in both classA and interfaceB then there is no issue as static method of interface is not overridden in class, it will be treated as different method. As there is no relation between class static method and interface static method so we can write any access modifier for class static methods. [Interface’s static method is by default public static].
* **Functional interface:** Have only one abstract method like Runnable interface, Comparable interface. @FunctionalInterface. Below are few functional interfaces designed by Java Developers for Java8 and above versions.
* **Predicate Interface:**

Predicate is a functional interface, which accepts an argument and returns a Boolean. Usually, it used to apply in a filter for a collection of objects. It has test() abstract method which will take any input values and return Boolean values based on its condition defined. We have 2 other methods as or(), and() which are useful when we want to bind multiple Predicates for a single check. We have negate() method to logical negation any predicate.

We have package8 in JavaBrain\_Java8 for Predicate examples.

Methods: test(), negate(), and(), or(), isEqual()

<https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>

* **Function Interface:**

Function is a functional interface in Java8 which take any type of input and return any other type of output after implementing its required methods per our requirement. It has an abstract method known as apply().

Identity() : it returns same value whatever we pass it as input.

//Take a String as input and find its length here this function will return length in int then convert it to Integer as defined in Integer.

Function<String, Integer> function = input -> input.length();

To overcome above problem, we have other interfaces which return primitive values.

These functions will return output in primitive int ToIntFunction, ToLongFunction, TODoubleFunction.  
Below will take different primitive type input and return them back with different primitive type.  
IntToLongFunction, IntToDoubleFunction, LongToIntFunction, LongToDoubleFunction, DoubleToIntFunction, DoubleToLongFunction.  
There are multiple Bi functions for same as well.

Methods: apply(), andThen(), compose(), identity()

BiFunction will take 2 input and return 1 output value.

<https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html>

* **Consumer:**

 It represents a function which takes in one argument and produces a result but don’t return anything.

We use this whenever we need to perform any action but no need to return anything. It has abstract method accept() and chaining method andThen().

Consumer also have primitive type functions which can be used per requirements.

We have Consumer interfaces where we required 2 input parameters as:

ObjIntConsumer<String> objIntConsumer = (input1, input2) -> System.*out*.println("objIntConsumer find String length and add it to second element: "+(input1.length() + input2));

objIntConsumer.accept("ashvani", 10);

Here second parameter will always be int type per interface name and 1st can be of any Object type. (as String type here)

Methods: accept(), andThen()

* **Supplier :**

It represents a function which does not take in any argument but produces a value of type T [T: object type].

It has abstract method get() which won’t take any input but return value per its implementation.

Chaining is not allowed for Supplier interface.

Method: get()

* For all above functional interfaces, we are taking 1 input argument, but if we need to pass 2 arguments for these functions, so we have their Bi-interfaces as BiPredicate, BiFunction and BiConsumer.
* We have Stream for primitive datatype as well and it supports int, long and double. Stream for them are IntStream, LongStream and DoubleStream.

If we want to convert int array into Stream, we can use below two ways:

a. converts in particular type of Object data type based on primitive

int array[] = {1,2,3,4,5,6};

IntStream = Arrays.stream(array);

b. convert array into generic type of Stream.

Stream S = Stream.of(array);

<https://medium.com/swlh/understanding-java-8s-consumer-supplier-predicate-and-function-c1889b9423d>

Need to check below links:

<https://www.youtube.com/watch?v=vHwToYEYvsU&list=PLTyWtrsGknYdqY_7lwcbJ1z4bvc5yEEZl&ab_channel=TechPrimers>

<https://www.youtube.com/watch?v=cICw5xmO2CI&list=PLJc-LD5TzDQR2fxBM65I3XgOfxkQAhURv&index=25&frags=wn&ab_channel=ekumeedhelp>

<https://www.youtube.com/watch?v=1xCxoOuDZuU&ab_channel=Amigoscode>

<https://www.youtube.com/watch?v=zolbIZS4SRQ&ab_channel=OracleDevelopers>

<https://medium.com/swlh/understanding-java-8s-consumer-supplier-predicate-and-function-c1889b9423d#:~:text=A%20Predicate%20interface%20represents%20a,new%20stream%20satisfying%20the%20predicate>.

|  |  |
| --- | --- |
| **Interface** | **Description** |
| [BiConsumer<T,U>](https://www.javatpoint.com/java-biconsumer-interface) | It represents an operation that accepts two input arguments and returns no result. |
| [Consumer<T>](https://www.javatpoint.com/java-consumer-interface) | It represents an operation that accepts a single argument and returns no result. |
| [Function<T,R>](https://www.javatpoint.com/java-function-interface) | It represents a function that accepts one argument and returns a result. |
| [Predicate<T>](https://www.javatpoint.com/java-predicate-interface) | It represents a predicate (boolean-valued function) of one argument. |
| BiFunction<T,U,R> | It represents a function that accepts two arguments and returns a a result. |
| BinaryOperator<T> | It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands. |
| BiPredicate<T,U> | It represents a predicate (boolean-valued function) of two arguments. |
| BooleanSupplier | It represents a supplier of boolean-valued results. |
| DoubleBinaryOperator | It represents an operation upon two double type operands and returns a double type value. |
| DoubleConsumer | It represents an operation that accepts a single double type argument and returns no result. |
| DoubleFunction<R> | It represents a function that accepts a double type argument and produces a result. |
| DoublePredicate | It represents a predicate (boolean-valued function) of one double type argument. |
| DoubleSupplier | It represents a supplier of double type results. |
| DoubleToIntFunction | It represents a function that accepts a double type argument and produces an int type result. |
| DoubleToLongFunction | It represents a function that accepts a double type argument and produces a long type result. |
| DoubleUnaryOperator | It represents an operation on a single double type operand that produces a double type result. |
| IntBinaryOperator | It represents an operation upon two int type operands and returns an int type result. |
| IntConsumer | It represents an operation that accepts a single integer argument and returns no result. |
| IntFunction<R> | It represents a function that accepts an integer argument and returns a result. |
| IntPredicate | It represents a predicate (boolean-valued function) of one integer argument. |
| IntSupplier | It represents a supplier of integer type. |
| IntToDoubleFunction | It represents a function that accepts an integer argument and returns a double. |
| IntToLongFunction | It represents a function that accepts an integer argument and returns a long. |
| IntUnaryOperator | It represents an operation on a single integer operand that produces an integer result. |
| LongBinaryOperator | It represents an operation upon two long type operands and returns a long type result. |
| LongConsumer | It represents an operation that accepts a single long type argument and returns no result. |
| LongFunction<R> | It represents a function that accepts a long type argument and returns a result. |
| LongPredicate | It represents a predicate (boolean-valued function) of one long type argument. |
| LongSupplier | It represents a supplier of long type results. |
| LongToDoubleFunction | It represents a function that accepts a long type argument and returns a result of double type. |
| LongToIntFunction | It represents a function that accepts a long type argument and returns an integer result. |
| LongUnaryOperator | It represents an operation on a single long type operand that returns a long type result. |
| ObjDoubleConsumer<T> | It represents an operation that accepts an object and a double argument, and returns no result. |
| ObjIntConsumer<T> | It represents an operation that accepts an object and an integer argument. It does not return result. |
| ObjLongConsumer<T> | It represents an operation that accepts an object and a long argument, it returns no result. |
| Supplier<T> | It represents a supplier of results. |
| ToDoubleBiFunction<T,U> | It represents a function that accepts two arguments and produces a double type result. |
| ToDoubleFunction<T> | It represents a function that returns a double type result. |
| ToIntBiFunction<T,U> | It represents a function that accepts two arguments and returns an integer. |
| ToIntFunction<T> | It represents a function that returns an integer. |
| ToLongBiFunction<T,U> | It represents a function that accepts two arguments and returns a result of long type. |
| ToLongFunction<T> | It represents a function that returns a result of long type. |
| UnaryOperator<T> | It represents an operation on a single operand that returnsa a result of the same type as its operand. |

Stream examples:

\* int sum = widgets.stream()  
\* .filter(w -> w.getColor() == RED)  
\* .mapToInt(w -> w.getWeight())  
\* .sum();

* Stream<T> filter(Predicate<? *super* T> predicate);
* <R> Stream<R> map(Function<? *super* T, ? *extends* R> mapper);
* <R> Stream<R> flatMap(Function<? *super* T, ? *extends* Stream<? *extends* R>> mapper);
* Stream<T> distinct();
* Stream<T> sorted();
* Stream<T> sorted(Comparator<? *super* T> comparator);
* Stream<T> peek(Consumer<? *super* T> action);
* Stream<T> limit(*long* maxSize);
* *void* forEach(Consumer<? *super* T> action);
* Object[] toArray();
* <R> R collect(Supplier<R> supplier,  
   BiConsumer<R, ? *super* T> accumulator,  
   BiConsumer<R, R> combiner);
* <R, A> R collect(Collector<? *super* T, A, R> collector);
* Optional<T> min(Comparator<? *super* T> comparator);
* Optional<T> max(Comparator<? *super* T> comparator);
* *long* count();
* *boolean* anyMatch(Predicate<? *super* T> predicate);
* *boolean* allMatch(Predicate<? *super* T> predicate);
* Optional<T> findFirst();

Optional<T> findAny();

* *public static*<T> Builder<T> builder() {  
   *return new* Streams.StreamBuilderImpl<>();  
  }
* *public static*<T> Stream<T> empty() {  
   *return* StreamSupport.stream(Spliterators.<T>emptySpliterator(), *false*);  
  }
* *public static*<T> Stream<T> of(T t) {  
   *return* StreamSupport.stream(*new* Streams.StreamBuilderImpl<>(t), *false*);  
  }

List<String> relatedOrderItemIdList = null;

            relatedOrderItemIdList = flowContextDTO.getRelatedOrderItem().stream().map(p -> p.getId())

                    .collect(Collectors.toList());

            notifyOttPublisher(correlationId, relatedOrderItemIdList);