Java : Functional

Java Interview Guide : 200+ Questions
Java New Features (Java 12, Java 11, Java 10, Java 9, Java 8, Java 7)

Introduction

- Treats computation as the evaluation of mathematical functions and avoids changing state and mutable data
  - No changes to state, No synchronization
- Example
  - List<Integer> numbers = Arrays.asList(1,3,4,6,2,7);
  - Int sum = numbers.stream().filter(number → (number % 2 != 0)).reduce(0, Integer::sum);

Lambda Expressions

- Lambda Expression
  - An anonymous function – left hand side input, right hand side output
  - Shorthand for writing a method in the same place you will use it.
  - Especially used in places where a methods is only being used once.
- Example
  - (a, b) → a + b;
  - () → System.out.println(“Hello”);
- Example
  - interface AddInterface {
    - public int addTowNumbers(int a, int b);
  - }
  - Int x =   AddInterface addInterface = (a, b) → a + b;   // Instead of needing an Anonymous Class or class just rewrite the expression
- Example
  - public class Utils { public static String transform(String s, StringFunction f) {return (f.apply(s)) } }
  - String result2 = transform(someString, String::toUpperCase)
- Lambda Expression Rules
  - Type can be determined only from context →
  - Can only be used with FunctionalInterface
  - Single Line Curly Braces are optional and so is the return
  - Must have curly braces if we have return keyword in lambda expression
  - must end with a semi colon

Lambda Expressions

- Example
  - Create method that contains an ArrayList of people objects
  - In the person class have a method public static int compare(Person p1, Person p2)
  - In the main class write a Collections.sort(people, Person :: compare) // :: separates the class name from the name of the method you are // calling
  - Results.foreach(p → System.out.printlnf(p));
- Example
  - Start with ArrayList of 5 Person Objects
  - Predicate member of java.util.function
  - Predicate<Person> personFilter = (p) → p.age() > 65
  - Predicate<Person> personFilter2 = (p) → p.age() > 65
  - Persons.foreach(p → { if (personFilter2.test(p)) System.out.println(p); };
- Example
  - Collections.sort(ArrayOfString, (str1, str2) → str1.compareToIgnoreCase(str);

Method References & Lambda scoping

- Method References
  - Method References can be used to refer either a static or non static method
  - Method is very short
- A method reference is a function that replace a lambda
- Method Reference can only be used with the FunctionalReference (One abstract Method)
- Example
  - interface MethodReference { // Must be a Functional Interface
    - void helloMethodReference() {
      - System.out.println(“From hellMethodReference!”);
    - }
  - public class StaticMehtodReferenceDemo {
    - static void helloMethodReference() { System.out.println(“From Hello Method Reference”); }
    - public static void main(String args[]) {
      - MethodReference methodReference = StaticMethodReferenceDemo::helloMethodReference;
      - methodReference.helloMethodReference();
  - }
-
-

Method References & Lambda scoping

- Scoping
  - The this variable refer to the outer class not the inner class that the lambda is turned into
  - Lambdas do not introduce a new level of scoping
  - Lambdas cannot introduce new variables with same name as variables in method that creates the lambda
    - Error : double x = 1.2;
    - Error : someMethod (x → doSomethingWith(x));
    - Ok: someMethod (y → x = 3.4);
  - Lambdas can refer to, but not modify local variables in method that create the lambda
    - double x = 1.2;
    - someMethod (y → x + y);
  - Lambdas can refer to and modify instance variables from the surrounding class
    - private double x = 1.2;
    - public void foo() { someMethod (y → x = 3.4; }
- Example
  - Enhancement – AddInterface addInterface1 (var a, var b) → var(a+b)
    - Reason 1 : Allow the use of final in the parameter list
    - Reason 2: Add validation to the parameters
  - @Min(value = 10)		is from javax.validation.constraint

Method References & Lamda scoping

- Constructor References
  - A specialized method reference which refer to the constructor reference
  - Example <ClassName::New>
  - Example – Need to double check this example
    - An Employee class with a constructor of Employee(String name, Integer Age)(
    - Interface EmployeeFactory → public abstract Employee getEmployee(String name, Integer age);
    - EmployeeFactory empFactory == Employee::new
    - emp = empFactory.getEmployee(“test”, 25)

@Functional Interface

- When ever we create a Lambda Expression. We are defining a function which implements a pre-defined/custom defined Functional Interface
- Different type of functional interface

Predicate	object → boolean	Has a test()
Consumer	object → void	Has a function called accept()
Function	object → different object	
BiFunction	Two Objects → Some other object	
Supplier	No Input and returns an output	Has a function called test

- Reduce → binary operator (another functional interface)
- ForEach → accepts an object, but not returns anything back (consumes it) and provides iterable Collections
  - Example – Arrays.asList(1,2,3,4,5,6,7,8,9.10).forEach(System::println)
- An interface which has only one abstract is a functional Interface
  - interface MyInterface {
    - public void method1(); }
  - A method without the body
  - May have any number of
    - default and static methods
    - can have methods of java .lang.object
- @FunctionalInterface annotation – Check whether the interface is a functional interface or not
  - If not then compilation error
  - @FuncitonalInterface is optional

@Functional Interface

- If we add @FunciotnalInterface annotation to any interface that interface will not become a Functional Interface, it will only check whether the interface is a Functional Interface or not
- Examples of Functional Interfaces: Runnable, Callable,
- Lambda expressions can be used only with Functional Interface in the code
- Example
- @FunctionalInterface
- public interface MyInterface1() {
  - void method1();
  - default void method2();
- }

Higher Order Functions

- Higher order functions in predicate
  - and → True if both the original and argument Predicate return true
  - or → True if either the original and argument predicate return true
  - negate → Returns the opposite of the original predicate
  - isequal → True if the original and current Predicate are the same
- Higher order functions in function
  - f1.compose(f2)
    - run f2 then pass the result of f1
    - Produce a function whose apply method when called first pass the argument to the apply method of f2 then passes the result to the apply method of f1
  - f1.andThen(f2)
    - first run f1 and then run f2 means to first run f1 then pass the result to f2
  - f2.andThen(f1) is the same as f1.compose(f2)
    - Many people usually think of the compose method
  - Function.identity – The apply method creates a function whose apply method returns the argument unchanged

Streams

- A stream is a source of object where both the intermediate operations can be stateful (sort, distinct) and stateless (map and filter) stateful → comparing with other elements
- Stream is a sequence of elements supporting sequential and parallel aggregate operation for processing objects from collections
  - Can have infinite (unbounded) streams where you can designate a generator function
  - Not a Data Structures since the don’t have storage they carry value from source through the pipeline
  - Wrapper around the Data Structures
  - To re-stream a list creating a stream just points at the existing data structure behind the scenes
  - streams have no storage they carry values from a source through a pipeline of operations and they never modify the underlying data structure
- stream is a source of object where both the intermediate operations can be stateful (sort, distinct) and stateless (map and filter) and stateful → comparing with other elements
  - Functions : map , filter, distinct,
  - Terminal : forEach, collect (group elements into a collection using functions toList, toMap, toSet, averagingDouble, groupingBy, partitionBy), sorted, sum, min, max
  - Parts of Stream : Source, Intermediate Operation, Terminal Operation
    - Intermediate operations return a stream
      - Filter from the above example is an intermediate operation
      - Usually return a new stream back
    - Reduce is a terminal operation – Might be a side effect (save tot he database) or a result
  - lazy operation. Not executed until a terminal operation is called or Short Circuit Method cause the earlier intermediate methods to be processed only until the short circuit method can be evaluated.
    - few exceptions : flatmap https://jaxenter.com/java-8-streams-lazy-136183.html
  - Steams have intermediate operations and terminal operations
  -

Streams

- A stream carries values from a source to a pipeline.
- Advantages
  - More expressive : with a for loop we need to go deep into a data structure, but with a steam we can use filter and map
  - Paralleled
  - Lazy loading, streams create other streams and will not be proceed until terminal operations is called
    - Can be short circuited by using method findAny
- Disadvantage
  - Low thousands and below streams and for loops do not differ much
  - Parallel streams have to manage the thread life cycle and most of their time is their, usually sequential streams are faster

Stream Operators

- map is an intermediate stream that modifies each element in a collection
  - map() is 1 to 1 Mapping
  - flatMap() is a one to many mapping
  - returns multiple objects of a single element
- Each function application produces a Stream then the Stream element are combined into a single stream
  - Ex A company is a list of department this produces a Stream of all combined employees
  - company.stream().flatMap((dept) → dept.employeeList().stream()).collect(Collectors.toList());
- Example
  - List<String> javaVersionList = new ArrayList<>();
  - javaVersionList.add(“java 7”);
  - javaVersionList.add(“java 8”);
  - javaVersionList.add(“java 9”);
  - List<String> javaVersionUpperCaseList = javaVersionList.stream().map(javaVersion → javaVersion.toUpperCase()).collectors.toList();
  - List<String> javaVersionUpperCaseList = javaVersionList.stream().
  - flatMap(javaVersion → Stream.of(javaVersion.toUpperCase, javaVersion..toLowerCase, javaVersion.concat(“ JFF”))).collectors.toList())
- // The output is three elements from the the flatMap for each item.
- Filter – Produces a new Stream that contains only the element of the original test that pass a given test
- There is a similar method found in the list, but it delete all that al that fail the test
- Example
  - Stream.of(nums).filter(n → n % 2 == 0)

How Streams Work

- Stream defer most operations until a terminal operation
- Operations that appear to traverse Stream Multiple Times actually traverse it once
  - Stream.map(someOp).filter(someTest).findFirst().get()
  - Does the map and filter options one element at a time
  - Continues only until first match on the filter test
    - Short Circuit → anyMatch, allMatch, noMatch, findFirst, findAny, limit
- Method Types
  - Intermediate methods
    - These are metho0ds that produce other streams. These methods don’t get processed until there is some method is some terminal method called
  - Terminal Methods
    - After one of these methods is invoked, the Stream is considered consumed and no more operations can be performed on it
  - Short Circuit Methods
    - cause intermediate methods to be processed only until the short-circuit method an be evaluated
    - Can be intermediate (limit, skip) or terminal (findFirst, allMatch)
    - Only Filters until it finds the first match
    - Stream.of(someArray).filter(e -> someTest(e)).findFirst().orElse(default)

How Streams Work

- Code
  - Stream.of(idArray).map(EmployeeUtils::findById).filter(e -> e != null).filter(e -> e.getSalary() > 500_000).findFirst().orElse(null));
- Apparent behavior
  - findById on all,
  - check all for null,
  - call getSalary on all non-null (& compare to $500K) on all remaining,
  - find first,
  - return it or null
- Actual behavior
  - findById on first,
  - check it for null,
  - if pass, call getSalary,
  - if salary > $500K,
  - return and done.
  - Repeat for second, etc if not found.
  - Return null if you get to the end and never got match.

Streams Limiting Stream Size, Sorting, Min, Max Distinct

- limit(n) → returns a Stream of the first n elements
  - short Circuit Operation
- skip(n) returns a Stream starting with element (throws away the first n elements)
- sorted
  - sorted with a Comparator works just like Arrays.sort
  - sorted with no arguments works only if the Stream element implement Comparable
  - sorting streams is more flexible than sorting array because you can do filter and mapping operations before and/or after
    - Note that inconsistency that method is called sorted not sort
  - example : empStream.sorted((e1, e2) -> e1.getSalary() - e2.getSalary())
  - Doing limit or skip after sorting does not short-circuit in the same manner as in the previous section
    - The system does ot know which are the first or last element until after sorting
    - If the stream element implement Comparable you may omit the lambda and just someStream.sorted()
- min and max than to sort forward or backward then take first element
  - empStream.max((e1, e2) -> e1.getSalary() – e2.getSalary()).get()
  - min and max is Omage(1) and sorted is Omega(n log n)
- distinct use equals as it comparison

Streams Limiting Stream Size, Sorting, Min, Max Distinct
Operations that check matches: allMatch, anyMatch, noneMatch, count

- example
  - List<Employee> emp3 =
    - sampleEmployee().sorted(Person::firstNameComparator)
      - .limit(2). collect(Collectors.toList())
    - limit2() does not limit the number of times firstNameComparator
- Operations that check matches: allMatch, anyMatch, noneMatch, count
  - allMatch, anyMatch, NonMatch take a predicate and return a boolean
    - Stop processing once an answer can be determined
  - count returns the number of elements

Number Specialized Steams

- A specialization of Steam makes it easier to deal with ints. Does not extend Stream but instead extends BaseStream on which Stream is built
- min, max,sum, average takes no arguments unlike the Stream that needs a Comparator
- Output as int[]
- Similar Interfaces : DoubleStream, LongStream
- Example : double totalCost = carList.stream().mapToDouble(Car::getPrice).sum()
- Example: int population - countryList.stream().filter(Utils:inRegion).mapToInt(Country::getPopulation).sum()
- Example: employeeList.stream().mapToDouble(Employee::salary).average().orElse(-1)
- regularStram.mapToInt – Assume that getAge returns an int.
  - Example produces an IntStream : personList.stream().mapToInt(Person::getAge)
- IntSteram.of
  - IntStream.of(int1, int2, int3),
  - IntStram.of(intArray) – Can also use Arrays.stream for this
- IntStream.range, IntStream.rangeClosed → IntStram.range(5,10)
- Random.ints → new Random().ints().anyInstanceOfRandom.ints() – Can apply limit to amek a finit stream

Number Specialized Steams

- Specific to Number Stream
  - min(), max() : No arguments, but output is OptionalInt
  - sum(): No arguments output is int. Returns 0 for an empty int stream
  - average: No arguments output is OptionalDouble
  - toArray(): No arguments, output is int[]
    - Although building an int[] from an intStream is more convenient than building an Integer[] from a Stream<Integer> turning an IntStream into a List<Stream>
      - cannot do yourStream.collect(Collectors.toList())
  - map, maToDouble, mapToObject-→ Function for the map must produce int
- Specific to DoubleStream
  - Creating
    - Creating regularStream.mapToDouble
    - DoubleStream.of
    - SomeRandom.doubles
- Long Stream
  - Creating – regularStream.mapToLong, LongStream.of, SomeRandom.longs
  - Methods : min, max, sum, average (no args, output is long)
  - toArray(no args, output is long[])
  - Correct Incorrect Attempts at Making IntStream
    - Stream.of(1, 2, 3, 4)   // Builds Stream<Integer> not IntStream
    - Integer[] nums = { 1, 2, 3, 4 }; Stream.of(nums)   // Builds Stream<Integer>, not IntStream
    - Stream.of({ 1,2,3,4})   // Build Stream containing one element is an int

Streams

- Example
  - allString.stream().
  - .filter(s → s.startsWith(“Ro”))
  - .map(String:to::lowerCase)
  - String::toLowerCase is a method Reference
  - .sorted()
  - .forEach(System.out::println)
- Given List<Integer> numberList
  - List<Integer> numbers = numberList.stream().filter javaVersion → javaVersion % 2).collect(Collectors.toList());
- Example of an infinite steam
  - Stream<Integer> infiniteStream = Stream.iterate(0, i -> i + 2); // Given
  - List<Integer> collect = infiniteStream.limit(10).collect(Collectors.toList()); // When
  - assertEquals(collect, Arrays.asList(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)); // Then
  -
- Operations
  - stream() → Convert a collection to a Stream
  - Immediate: filter, limit, sorted, map, flatMap, peek
  - Terminal: collect, reduce, min, max

Streams

- Way to make a stream
  - of (Array of Objects) // Not array of primitives
  - stream() // Inherits from collection
  - Stream.generate(Supplier<T> s) // Have a function produce the values
  - Stream.of(val1, val2, …) // From Individual Values
  - SteamBuilder.build() // Use the builder pattern (add, accept)
  - String.chars, Stream.of(someString.split(…)) // another example of the of operator
  - Stream.iterate(T seed, UnaryOpertor<T> f) // A seed which the value and then function to computer the value and continues on with new value
  - StreamBuilder // Allows the elements to be added then call the build (no more elements cannot be added)

Java 9 – Stream Enhancements

- takeWhile
  - It will stop if condition false for any value
  - It may or may not take all values in the collection
- dropWhile
  - It will start if condition is false
- ofNullable
  - It prevents null pointer exception
- Examples
  - List<Integer> numbersList = Arrays.asList(50, 60, 80, 90, 40, 30);
  - List<Integer> tempList = numberList.stream().takeWhile(j → j < 80).collect(Collectors.asList()) produces [50,60]
  - List<Integer> tempList = numberList.stream().takeWhile(j → j < 80).collect(Collectors.asList()) produces [80, 90, 40, 30]
- Example
  - List<Integer> nullList = null
  - Stream.ofNullable(nullList).collect(Collectors.toList()) // []
  - Stream.ofNullable(numbersList).collect(Collectors.toList()) // [50, 60, 80, 90, 40, 30]

Collectors

- someStream.collect(Collectors.toList())
- someStream.toArray(EntryType[]::new)
- Examples
  - EntryType[] myArray = myStream.toArray(EntryType[]::new);
  - EntryType[] myArray = myStream.toArray(n → buildEmptyArray(n)); // Lambda is an IntFunction that takes an int (size) as an argument and
    // returns an empty array that can be filled in.
- Example
  - anyStream.collect(toList())
  - anyStream.collect(joining(delimiter)).toString()
  - AnyStream.collect(toCollection(CollectionType::new)
  - List<Employee> emps = ids.stream().map(EmployeeSamples::findGoogler).collect(Collectors.toList())
  - Ids.stream.map(EmployeeSamples::findGoogler).filter(e → e != null).map(Employee::getLastName).collect(Collectors.joining(“, “).toString()
- StringJoiner class builds delimiter separated String with optional prefix and suffix
- Examples
  - SomeStream.collect(Collectors.toSet())
  - SomeStream.collect(toCollection(TreeSet::new))
  - SomeStream.collect(toCollection(Stack::new))
  - Googleers.stream().map(Employee:getFirstName).collect(Collectors.toCollection(TreeSet::new))

Collectors

- PartioningBy: Building Maps
  - Provide a predicate and create two list of entries (true, false)
  - Map<Bolean<List<Employee>> oldTimerMap = employeeStream().collect(partitionBy(e→ e.getEmployeeId() < 10).get(true)
  - Map<Bolean<List<Employee>> oldTimerMap = employeeStream().collect(partitionBy(e→ e.getEmployeeId() < 10).get(false)
- Grouping By
  - Provide a function and it builds a map where each output value of the function maps to a List of entries that gave the value
  - Map<Department, List<<Employee>> deptTable = employeeStream().collect(Collectors.groupingByj(Employee::getDepartment));

Reduce Method and Related Reduction Operations

- Takes a Stream<T> and combine or compare the entries to produce a single value of Type T
- You start with a seed (identity) value, combine this value with the first entry of the Stream
  - Combine the result with the second entry of the Stream and so
    - reduce is particularly with map or filter
    - Works properly with parallel streams if operator is associative and has no side effects
  - reduce(start, binaryOperator) → Takes starter value and Binary Operator and returns result direclty
  - reduce(binaryOperator) → Takes binaryOperator with no starter, It starts by combining first 2 values with each, Returns as Optional
- Examples
  - nums.strream().reduce(Double.MIN_VALUE, Double::max);
  - nums.stream()..reduce(1, (n1, n2) → n1 * n2)
  - letters.stream().reduce("", StringConcat)
    - The "" is start (identity value). It is combined with the first entry in the Stream
  - Example
    - List <String> = Arrays.asList("a", "b", "c", "d");
    - String concat = letters.stream().reduce("", String::concat); → will print abcd
  - Example: letters.stream().reduce("", (s1,s2) → s2 + s1); // Reverse the order of the S1 and S2 in the concatentation

Reduce Method and Related Reduction Operations

- Example
  - googlers.stream().mapToInt(Employee::getSalary).sum()
  - googlers.stream().map(Employee::getSalary).reduce(0, Integer::sum)
- Example
  - googlers.stream().mapToInt(Employee::getSalary).min().orElse(Integer.MAX_VALUE)   // use min()
  - googlers.stream().map(Employee:getSalary).min (n1, n2) → n1 - n2).orElse(Integer.MAX_VALUE)   // min(comparator)
  - googlers.stream().map(Employee:getSalary).reduce(Integer.MAX_VALUE), Integer::min)   // Use map then reduce

Parallel Streams

- SequentialStream → stream()
- parallelStream → parallelStream()
  - Often easier than explicit threads
  - Can used with minimal changes to serial code
  - Use Fork/Join
  - Are beneficial when you never wait for I/O
  - give more performance
  - Does this by using more hardware cores, so do not use on Single Core Computers
    - Have no benefit on single core computer
  - Examples
    - anyStream.parellal()
    - anyStream.pareallelStream() → shortcut for anyList.stream().parallel()
    - Int sum = IntStream.of(num).parallel().sum();
    - Explanation
      - Use fork/join framework internally (see separate lecture)
      - have one thread per code
      - Are beneficial even when waitng for I/O
      - minimal changes to serial code
- Take a look at the following
  - Duration.between(startTime,endTime)
  - Instance.now()

Parallel Streams

- Will you always get same answer in Parallel (do some research)
  - sorted, min, max
    - No
  - findFirst
    - No. Use findAny if you do not care
  - map, filter
    - No, but you do not care about what the stream looks like in intermediate stages. You only care about the intermediate operation
  - allMatch, anyMatch, noneMatch
    - Yes
  - sum and average are the same if IntStream and LongStream
    - sum and average could be different if you DoubleStream. Reordering the additions could yield slightly different results due to round off err
  - reduce (and sum and average)
    - It depends
    - reduce is the same
      - if no side effect on global data are preformed.
      - The combining associative (where reodering the operations does not matter)
  - Binary Operator
    - should be stateless
      - Guaranteed if an explicit lambda is used
    - Operator does not modify global data
      - Do not modify data that is not passed into the lamdba

Infinite Streams

- Infinite Streams
  - Creating
    - Streams.generte(value generator)
      - The Supplier is invoked each time the system needs a Stream element
    - Stream.iterate(initial value, valueTransformer)
  - Usage
    - The values are not not calculated
    - To avoid unterminated processing you must eventually use a size limiting operation like limit or findFirst
  - Example List<Employee> emps = Stream.generate(() → randomEmployee()).limit(someRuntimeValue).collect(Collectors.toList()

Higher Order Functions

- Example
  - round = Math::rint
  - transform(nums, round.compose(Math:sqrt))
- Example
  - tranform(nums, Math::rint.compose(Math::sqrt))
- First one works and the second one does not
  - Math:rint does not have at type until it is assigned to a variable or passed to a method
  - Any interface that has a SAM methods that can take two doubles could be a target for Math::rint
  - But other interfaces do not have compose method
- Method from Consumer
  - andThen --f1.andThen(f2) produces a Consumer that first passes argument to f1 and then passes argument to f2
  - Differences between andThen of Consumer and of function
    - With andThen from Consumer, the argument is passes to the accept method of f1, then that same argument is passed to the accept method of f2
    - With andThen from Function, the argument is passed to the apply method of f1, then the result of apply is passed to the apply method of f2
- The Comparator has a thenComparing so you can compare multiple sorting criteria

Parallel Streams

- Parallel reduce: No Global Data
  - binary operator itself should be stateless
    - Guaranteed if an explicit lambda is used, but not guaranteed if you directly build an instance of a class that implements BinaryOperator or if you use a method reference that refers to a stateful class
  - operator does not modify glboal data
    - The body of a lambda is allowed to mutate instance variables of the surrounding class or call setter methods that modify instance variables. There is no guarantee that parallel reduce will be safe.
- Parallel reduce: Associative Operation → (a op b) op c == a op (b op c)
  - Division or Subtraction are not associative
  - Addition or multiplication of ints ,b double may have a slightly different result if doules is used

Predefined Functional Interface
Predefined Function Interface – Function(andThen & compose)

- For Single Input/Single Output the the Function<T,U> form the java.util.function class
    - has one function function.apply
- andThen
    - Method 1 is input to Method 2
- compose
    - Method 2 is input to method 1
    - Example
        - public String static void main(String args[]) {
            - Function<String, String> function = s→ s.toUpperCase();
            - Function<String, String> function2 = s → s + “ World”;
            -
            - System.out.println(“Output of function1 : “ + function1.apply(“Hello”))
            - System.out.println(“andThen Output : “ + function1.andThen(function2).apply(“Hello”); // To hello apply function 1 then function 2
            - System.out.println(“andThen Output : “ + function1.compose(function2).apply(“Hello”); // To hello apply function 2 then function 1
            -
            -
        -