Вопросы для подготовки:

- Какие нововведения, появились в Java 8 и JDK 8?

* Методы интерфейсов по умолчанию;
* Лямбда-выражения;
* Функциональные интерфейсы;
* Ссылки на методы и конструкторы;
* Повторяемые аннотации;
* Аннотации на типы данных;
* Рефлексия для параметров методов;
* *Stream API* для работы с коллекциями;
* Параллельная сортировка массивов;
* Новое API для работы с датами и временем;
* Новый движок JavaScript *Nashorn*;
* Добавлено несколько новых классов для потокобезопасной работы;
* Добавлен новый API для Calendar и Locale;
* Добавлена поддержка *Unicode 6.2.0*;
* Добавлен стандартный класс для работы с *Base64*;
* Добавлена поддержка беззнаковой арифметики;
* Улучшена производительность конструктора java.lang.String(byte[], \*) и метода java.lang.String.getBytes();
* Новая реализация AccessController.doPrivileged, позволяющая устанавливать подмножество привилегий без необходимости проверки всех остальных уровней доступа;
* *Password-based* алгоритмы стали более устойчивыми;
* Добавлена поддержка *SSL/TLS Server Name Indication (NSI)* в *JSSE Server*;
* Улучшено хранилище ключей (KeyStore);
* Добавлен алгоритм *SHA-224*;
* Удален мост *JDBC - ODBC*;
* Удален *PermGen*, изменен способ хранения мета-данных классов;
* Возможность создания профилей для платформы Java SE, которые включают в себя не всю платформу целиком, а некоторую ее часть;
* Инструментарий
  + Добавлена утилита jjs для использования *JavaScript Nashorn*;
  + Команда java может запускать *JavaFX* приложения;
  + Добавлена утилита jdeps для анализа *.class*-файлов.

- Что такое lambda? Какова структура и особенности использования лямбда-выражения?

Лямбда представляет собой набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы.

Основу лямбда-выражения составляет лямбда-оператор, который представляет стрелку ->. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая, собственно, представляет тело лямбда-выражения, где выполняются все действия.

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

interface Operationable {

int calculate(int x, int y);

}

public static void main(String[] args) {

Operationable operation = (x, y) -> x + y;

int result = operation.calculate(10, 20);

System.out.println(result); //30

}

По факту лямбда-выражения являются в некотором роде сокращенной формой внутренних анонимных классов, которые ранее применялись в Java.

Отложенное выполнение (deferred execution) лямбда-выражения- определяется один раз в одном месте программы, вызываются при необходимости, любое количество раз и в произвольном месте программы.

Параметры лямбда-выражения должны соответствовать по типу параметрам метода функционального интерфейса:

operation = (int x, int y) -> x + y;

//При написании самого лямбда-выражения тип параметров разрешается не указывать:

(x, y) -> x + y;

//Если метод не принимает никаких параметров, то пишутся пустые скобки, например,

() -> 30 + 20;

//Если метод принимает только один параметр, то скобки можно опустить:

n -> n \* n;

Конечные лямбда-выражения не обязаны возвращать какое-либо значение.

interface Printable {

void print(String s);

}

public static void main(String[] args) {

Printable printer = s -> System.out.println(s);

printer.print("Hello, world");

}

Блочные лямбда-выражения обрамляются фигурными скобками. В блочных лямбда-выражениях можно использовать внутренние вложенные блоки, циклы, конструкции if, switch, создавать переменные и т.д. Если блочное лямбда-выражение должно возвращать значение, то явным образом применяется оператор return:

Operationable operation = (int x, int y) -> {

if (y == 0) {

return 0;

}

else {

return x / y;

}

};

- Как отсортировать список строк с помощью лямбда-выражения?

public static List<String> sort(List<String> list){

Collections.sort(list, (a, b) -> a.compareTo(b));

return list;

}

- Что такое ссылка на метод?

Если существующий в классе метод уже делает все, что необходимо, то можно воспользоваться механизмом method reference (ссылка на метод) для непосредственной передачи этого метода. Такая ссылка передается в виде:

имя\_класса::имя\_статического\_метода для статического метода;

объект\_класса::имя\_метода для метода экземпляра;

название\_класса::new для конструктора.

Результат будет в точности таким же, как в случае определения лямбда-выражения, которое вызывает этот метод.

private interface Measurable {

public int length(String string);

}

public static void main(String[] args) {

Measurable a = String::length;

System.out.println(a.length("abc"));

}

Ссылки на методы потенциально более эффективны, чем использование лямбда-выражений. Кроме того, они предоставляют компилятору более качественную информацию о типе и при возможности выбора между использованием ссылки на существующий метод и использованием лямбда-выражения, следует всегда предпочитать использование ссылки на метод.

- Что такое функциональные интерфейсы?

**Функциональный интерфейс** - это интерфейс, который определяет только один абстрактный метод.

Чтобы точно определить интерфейс как функциональный, добавлена аннотация @FunctionalInterface, работающая по принципу @Override. Она обозначит замысел и не даст определить второй абстрактный метод в интерфейсе.

Интерфейс может включать сколько угодно default методов и при этом оставаться функциональным, потому что default методы - не абстрактные.

- Какие функциональные интерфейсы вы знаете?

Predicate

Predicate — функциональный интерфейс для проверки соблюдения некоторого условия. Если условие соблюдается, возвращает true, иначе — false:

Consumer

Consumer (с англ. — “потребитель”) — функциональный интерфейс, который принимает в качестве входного аргумента объект типа T, совершает некоторые действия, но при этом ничего не возвращает:

Supplier

Supplier (с англ. — поставщик) — функциональный интерфейс, который не принимает никаких аргументов, но возвращает некоторый объект типа T:

Function

Function — этот функциональный интерфейс принимает аргумент T и приводит его к объекту типа R, который и возвращается как результат:

UnaryOperator

UnaryOperator — функциональный интерфейс, принимает в качестве параметра объект типа T, выполняет над ним некоторые операции и возвращает результат операций в виде объекта того же типа T:

- Что такое StringJoiner?

Класс StringJoiner используется, чтобы создать последовательность строк, разделенных разделителем с возможностью присоединить к полученной строке префикс и суффикс:

StringJoiner joiner = new StringJoiner(".", "prefix-", "-suffix");

for (String s : "Hello the brave world".split(" ")) {

joiner.add(s);

}

System.out.println(joiner); //prefix-Hello.the.brave.world-suffix

- Что такое default методы в интрефейсе?

Java 8 позволяет добавлять неабстрактные реализации методов в интерфейс, используя ключевое слово default:

interface Example {

int process(int a);

default void show() {

System.out.println("default show()");

}

}

* Если класс реализует интерфейс, он может, но не обязан, реализовать методы по-умолчанию, уже реализованные в интерфейсе. Класс наследует реализацию по умолчанию.
* Если некий класс реализует несколько интерфейсов, которые имеют одинаковый метод по умолчанию, то класс должен реализовать метод с совпадающей сигнатурой самостоятельно. Ситуация аналогична, если один интерфейс имеет метод по умолчанию, а в другом этот же метод является абстрактным - никакой реализации по умолчанию классом не наследуется.
* Метод по умолчанию не может переопределить метод класса java.lang.Object.
* Помогают реализовывать интерфейсы без страха нарушить работу других классов.
* Позволяют избежать создания служебных классов, так как все необходимые методы могут быть представлены в самих интерфейсах.
* Дают свободу классам выбрать метод, который нужно переопределить.
* Одной из основных причин внедрения методов по умолчанию является возможность коллекций в Java 8 использовать лямбда-выражения.

- Что такое Optional?

Опциональное значение Optional — это контейнер для объекта, который может содержать или не содержать значение null. Такая обёртка является удобным средством предотвращения NullPointerException, т.к. имеет некоторые функции высшего порядка, избавляющие от добавления повторяющихся if null/notNull проверок

- Что такое Stream?

Интерфейс java.util.Stream представляет собой последовательность элементов, над которой можно производить различные операции.

Операции над стримами бывают или *промежуточными (intermediate)* или *конечными (terminal)*. Конечные операции возвращают результат определенного типа, а промежуточные операции возвращают тот же стрим. Таким образом вы можете строить цепочки из несколько операций над одним и тем же стримом.

У стрима может быть сколько угодно вызовов промежуточных операций и последним вызов конечной операции. При этом все промежуточные операции выполняются лениво и пока не будет вызвана конечная операция никаких действий на самом деле не происходит (похоже на создание объекта Thread или Runnable, без вызова start()).

Стримы создаются на основе каких-либо источников, например классов из java.util.Collection.

Ассоциативные массивы (maps), например, HashMap, не поддерживаются.

Операции над стримами могут выполняться как последовательно, так и параллельно.

Потоки не могут быть использованы повторно. Как только была вызвана какая-нибудь конечная операция, поток закрывается.

Кроме универсальных объектных существуют особые виды стримов для работы с примитивными типами данных int, long и double: IntStream, LongStream и DoubleStream. Эти примитивные стримы работают так же, как и обычные объектные, но со следующими отличиями:

* используют специализированные лямбда-выражения, например, IntFunction или IntPredicate вместо Function и Predicate;
* поддерживают дополнительные конечные операции sum(), average(), mapToObj().

- Какие существуют способы создания стрима?

Из коллекции:

Stream<String> fromCollection = Arrays.asList("x", "y", "z").stream();

Из набора значений:

Stream<String> fromValues = Stream.of("x", "y", "z");

Из массива:

Stream<String> fromArray = Arrays.stream(new String[]{"x", "y", "z"});

Из файла (каждая строка в файле будет отдельным элементом в стриме):

Stream<String> fromFile = Files.lines(Paths.get("input.txt"));

Из строки:

IntStream fromString = "0123456789".chars();

С помощью Stream.builder():

Stream<String> fromBuilder = Stream.builder().add("z").add("y").add("z").build();

С помощью Stream.iterate() (бесконечный):

Stream<Integer> fromIterate = Stream.iterate(1, n -> n + 1);

С помощью Stream.generate() (бесконечный):

Stream<String> fromGenerate = Stream.generate(() -> "0");

- В чем разница между Collection и Stream?

Коллекции позволяют работать с элементами по-отдельности, тогда как стримы так делать не позволяют, но вместо этого предоставляют возможность выполнять функции над данными как над одним целым.

Также стоит отметить важность самой концепции сущностей: Collection - это прежде всего воплощение Структуры Данных. Например, Set не просто хранит в себе элементы, он реализует идею множества с уникальными элементами, тогда как Stream, это прежде всего абстракция необходимая для реализации конвейера вычислений, собственно, поэтому, результатом работы конвейера являются те или иные Структуры Данных или же результаты проверок/поиска и т.п.

- Для чего нужен метод collect() в стримах?

Метод collect() является конечной операцией, которая используется для представления результата в виде коллекции или какой-либо другой структуры данных.

collect() принимает на вход Collector<Тип\_источника, Тип\_аккумулятора, Тип\_результата>, который содержит четыре этапа: *supplier* - инициализация аккумулятора, *accumulator* - обработка каждого элемента, *combiner* - соединение двух аккумуляторов при параллельном выполнении, *[finisher]* - необязательный метод последней обработки аккумулятора. В Java 8 в классе Collectors реализовано несколько распространённых коллекторов:

* toList(), toCollection(), toSet() - представляют стрим в виде списка, коллекции или множества;
* toConcurrentMap(), toMap() - позволяют преобразовать стрим в Map;
* averagingInt(), averagingDouble(), averagingLong() - возвращают среднее значение;
* summingInt(), summingDouble(), summingLong() - возвращает сумму;
* summarizingInt(), summarizingDouble(), summarizingLong() - возвращают SummaryStatistics с разными агрегатными значениями;
* partitioningBy() - разделяет коллекцию на две части по соответствию условию и возвращает их как Map<Boolean, List>;
* groupingBy() - разделяет коллекцию на несколько частей и возвращает Map<N, List<T>>;
* mapping() - дополнительные преобразования значений для сложных Collector-ов.

- Какие терминальные методы работы со стримами вы знаете?

* findFirst() возвращает первый элемент;
* findAny() возвращает любой подходящий элемент;
* collect() представление результатов в виде коллекций и других структур данных;
* count() возвращает количество элементов;
* anyMatch() возвращает true, если условие выполняется хотя бы для одного элемента;
* noneMatch() возвращает true, если условие не выполняется ни для одного элемента;
* allMatch() возвращает true, если условие выполняется для всех элементов;
* min() возвращает минимальный элемент, используя в качестве условия Comparator;
* max() возвращает максимальный элемент, используя в качестве условия Comparator;
* forEach() применяет функцию к каждому объекту (порядок при параллельном выполнении не гарантируется);
* forEachOrdered() применяет функцию к каждому объекту с сохранением порядка элементов;
* toArray() возвращает массив значений;
* reduce()позволяет выполнять агрегатные функции и возвращать один результат.

Для числовых стримов дополнительно доступны:

* sum() возвращает сумму всех чисел;
* average() возвращает среднее арифметическое всех чисел.

- Какие промежуточные методы работы со стримами вы знаете?

* filter() отфильтровывает записи, возвращая только записи, соответствующие условию;
* skip() позволяет пропустить определённое количество элементов в начале;
* distinct() возвращает стрим без дубликатов (для метода equals());
* map() преобразует каждый элемент;
* peek() возвращает тот же стрим, применяя к каждому элементу функцию;
* limit() позволяет ограничить выборку определенным количеством первых элементов;
* sorted() позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator;
* mapToInt(), mapToDouble(), mapToLong() - аналоги map() возвращающие стрим числовых примитивов;
* flatMap(), flatMapToInt(), flatMapToDouble(), flatMapToLong() - похожи на map(), но могут создавать из одного элемента несколько.

Для числовых стримов дополнительно доступен метод mapToObj(), который преобразует числовой стрим обратно в объектный.

- Для чего в стримах применяются методы forEach() и forEachOrdered()?

* forEach() применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется;
* forEachOrdered() применяет функцию к каждому объекту стрима с сохранением порядка элементов.

- Для чего в стримах предназначены методы map() и mapToInt(), mapToDouble(), mapToLong()?

Метод map() является промежуточной операцией, которая заданным образом преобразует каждый элемент стрима.

mapToInt(), mapToDouble(), mapToLong() - аналоги map(), возвращающие соответствующий числовой стрим (то есть стрим из числовых примитивов):

- Что делает метод filter() в стримах?

Метод filter() является промежуточной операцией принимающей предикат, который фильтрует все элементы, возвращая только те, что соответствуют условию.

- Что делает метод limit() в стримах?

Метод limit() является промежуточной операцией, которая позволяет ограничить выборку определенным количеством первых элементов.

- Что делает метод sorted() в стримах?

Метод sorted() является промежуточной операцией, которая позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator.

Порядок элементов в исходной коллекции остается нетронутым - sorted() всего лишь создает его отсортированное представление.

- Для чего в стримах предназначены методы flatMap(), flatMapToInt(), flatMapToDouble(), flatMapToLong()?

Метод flatMap() похож на map, но может создавать из одного элемента несколько. Таким образом, каждый объект будет преобразован в ноль, один или несколько других объектов, поддерживаемых потоком. Наиболее очевидный способ применения этой операции — преобразование элементов контейнера при помощи функций, которые возвращают контейнеры.

Stream

.of("H e l l o", "w o r l d !")

.flatMap((p) -> Arrays.stream(p.split(" ")))

.toArray(String[]::new);//["H", "e", "l", "l", "o", "w", "o", "r", "l", "d", "!"]

flatMapToInt(), flatMapToDouble(), flatMapToLong() - это аналоги flatMap(), возвращающие соответствующий числовой стрим.

- Что такое ForkJoinPool?

- Что вы знаете про Date Time API из Java 8?

- Какой класс появился в Java 8 для кодирования/декодирования данных?

Base64 - потокобезопасный класс, который реализует кодировщик и декодировщик данных, используя схему кодирования base64 согласно RFC 4648 и RFC 2045.

Base64 содержит 6 основных методов:

getEncoder()/getDecoder() - возвращает кодировщик/декодировщик base64, соответствующий стандарту RFC 4648; getUrlEncoder()/getUrlDecoder() - возвращает URL-safe кодировщик/декодировщик base64, соответствующий стандарту RFC 4648; getMimeEncoder()/getMimeDecoder() - возвращает MIME кодировщик/декодировщик, соответствующий стандарту RFC 2045.

- Как создать Base64 кодировщик и декодировщик?

String b64 = Base64.getEncoder().encodeToString("input".getBytes("utf-8")); //aW5wdXQ==

// Decode

new String(Base64.getDecoder().decode("aW5wdXQ=="), "utf-8"); //input