1. **Лямбда-выражения в Java**

Лямбда-выражения — это блоки кода с параметрами, которые можно вызвать из другого места программы.

Присутствуют начиная с 8 версии. Являются анонимными классами, реализующими метод функционального интерфейса. Имеют доступ только к final (или effectively final) переменным из охватывающей области видимости (для потокобезопасности). Позволяют уменьшить количество кода и повысить его читаемость.

Лямбда-выражения в Java состоят из параметров и стрелки —> отделяющей тело функции.

1. **Неизменяемые классы. Модификатор final**

Неизменяемый класс – это класс, состояние которого не может быть изменено после создания. Здесь состоянием объекта считаются значения, хранимые в экземпляре класса, будь то примитивные типы или ссылочные типы.

Когда ключевое слово final используется с переменной, полем или параметром, это означает, что после передачи ссылки или создания экземпляра его значение не может быть изменено во время выполнения программы. Если переменная без какого-либо значения была объявлена как final, тогда она называется пустой / неинициализированной конечной переменной и может быть инициализирована только через конструктор.

Всякий раз, когда метод объявляется как final, он не может быть переопределен никаким дочерним классом на протяжении всего выполнения программы.

Для класса final означает, что класс не сможет иметь подклассов, т.е. запрещено наследование. Это полезно при создании неизменяемых объектов. К абстрактным классам (с ключевым словом abstract), нельзя применить модификатор final

1. **Побочные эффекты. Функциональное программирование**

Побочные эффекты — любые действия работающей программы, изменяющие среду выполнения. К побочным эффектам относятся: доступ (чтение или запись) к объекту, определённому с модификатором volatile (это один из способов обеспечения согласованного доступа к переменной разными потоками); изменение (запись) объекта; изменение файла.

Функциональное программирование — это программирование, в котором функции являются объектами, и их можно присваивать переменным, передавать в качестве аргументов другим функциям, возвращать в качестве результата от функций.

1. **Потоки (streams). Терминальные и нетерминальные операции**

Нетерминальные потоковые операции – это операции, которые добавляют слушателя в поток.

Они преобразовывают или фильтруют элементы в потоке. При добавлении нетерминальной операции в поток, получаем новый поток в качестве результата. Он представляет собой поток элементов, полученных из исходного с применением нетерминальной операции.

Терминальные потоковые операции – запускают внутреннюю итерацию элементов, вызывают всех слушателей и возвращают результат.

Терминальные операции обычно возвращают одно значение. Как только терминальная операция вызывается в потоке, начинается итерация его и любого из связанных с ним. По завершении итерации возвращается результат.

1. **Потоки (streams). Методы map и filter.**

Метод map преобразует каждый элемент стрима. Например, у нас есть коллекция collection1 = Arrays.asList(«а1», «а2», «а3») и мы добавляем к каждому элементу «\_1» на конце. Collection1.stream().map((s) -> s + “\_1”.collect(Collectors.toList())

Метод filter отфильтровывает записи, возвращает только записи, соответствующие условию. Например, у нас есть коллекция collection = Arrays.asList(«а1», «а2», «а3», «а1») и нам нужно вернуть кол-во вхождений объекта «а1». Collection.stream().filter(«а1»::equals).count()

1. **Потоки (streams). Метод reduce.**

Метод reduce позволяет выполнять агрегатные функции над всей коллекцией (такие как сумма, нахождение минимального или максимального значение и т.п.), он возвращает одно значение для стрима, функция получает два аргумента — значение полученное на прошлых шагах и текущее значение.

Дана коллекция чисел Arrays.asList(1, 2, 3, 4, 2), найдем сумму чисел или вернем ноль. collection.stream().reduce((s1, s2) -> s1 + s2).orElse(0)

1. **Потоки (streams). Метод collect.**

Collect представляет результат в виде коллекций и других структур данных, для этого используются статические методы из Collectors, например преобразование в List будет stream.collect(Collectors.toList()).

Также можно выполнять математические действия. Например, получить сумму нечетных чисел: numbers.stream().collect(Collectors.summingInt(((p) -> p % 2 == 1? p: 0)))

1. **Потоки (threads). Класс Thread и интерфейс Runnable. Потокобезопасность (thread safety).**

Класс Thread – это основной класс, на котором основана многопоточная система Java. Он предоставляет конструкторы и методы для поддержки многопоточности и реализует запускаемый интерфейс. Некоторые методы Thread для управления потоками: getName(), getPriority(), isAlive() – определить, выполняется ли поток, start() – запускаем поток.

Для создания нового потока нужно реализовать интерфейс Runnable. Можно создать поток из любого объекта, реализующего интерфейс Runnable и объявить метод run().

При объявлении нового класса с интерфейсом Runnable, вам нужно использовать конструктор: Thread(Runnable объект\_потока, String имя\_потока). В первом параметре указывается экземпляр класса, он определяет, где начнётся выполнение потока. Во втором параметре передаётся имя потока.

1. **Потоки (threads). Синхронизация потоков: ключевое слово synchronized, захват и освобождение монитора.**

Для решения проблемы с потоками, которые могут внести путаницу, используется синхронизация.

Метод может иметь модификатор syncronized. Когда поток находится внутри синхронизированного метода, все другие потоки, которые пытаются вызвать его в том же экземпляре, должны ожидать.

Также, ключевое слово syncronized можно использовать в качестве оператора.

Каждый объект в java имеет свой монитор. Для достижения эффектов взаимного исключения и синхронизации потоков используют следующие операции.

Monitorenter: захват монитора. В один момент времени монитором может владеть лишь один поток. Если на момент попытки захвата монитор занят, поток, пытающийся его захватить, будет ждать до тех пор, пока он не освободится. При этом, потоков в очереди может быть несколько.

monitorexit: освобождение монитора.

1. **Потоки (threads). Синхронизация потоков: ключевое слово synchronized, отношения happens-before**

Happens-before («выполняется прежде») — отношение порядка между атомарными операциями (Атомарные операции — это операции, которые нельзя разделить. Например, операция присваивания значения переменной — атомарная.). Оно означает, что вторая команда будет видеть изменения первой команды, и что первая команда выполнилась перед второй.

Happens-before возникает:

* Освобождение монитора «выполняется прежде» любого последующего взятия блокировки этого монитора.
* Присвоение значение volatile полю «выполняется прежде» любого последующего чтения значения этого поля.

1. **Потоки (threads). Модификатор volatile**

Ключевое слово volatile используется для изменения значения переменной различными потоками. Он также используется, чтобы сделать классы потокобезопасными.

Это означает, что несколько потоков могут использовать метод и экземпляр классов одновременно без каких-либо проблем. Ключевое слово volatile в Java может использоваться как с примитивным типом, так и с объектами.

1. **Потоки (threads). Классы атомарных значений и их операции**

Наиболее часто используемыми классами атомарных переменных в Java являются AtomicInteger, AtomicLong, AtomicBoolean и AtomicReference. Эти классы представляют собой int, long, boolean, и ссылку на объект соответственно, которые могут быть атомарно обновлены. Основными методами, предоставляемыми этими классами, являются:

Get() – получает значение из памяти, чтобы изменения, внесенные другими потоками, были видны; эквивалентно чтению переменной volatile

Set() – записывает значение в память, чтобы изменение было видно другим потокам; эквивалентно записи переменной volatile

lazySet() – в конечном итоге записывает значение в память, возможно, переупорядоченное с последующими соответствующими операциями с памятью. Один из вариантов использования - аннулирование ссылок ради сбора мусора, к которому больше никогда не будет доступа.

1. **Потоки (threads). Использование методов wait/notify для взаимодействия между потоками.**

wait: перемещение текущего потока в так называемый wait set монитора и ожидание того, как произойдёт notify. Выход из метода wait может оказаться и ложным. После того, как поток, владеющий монитором, сделал wait, монитором может завладеть любой другой поток.

notify(all): пробуждается один (или все) потоки, которые сейчас находятся в wait set монитора. Чтобы получить управление, пробуждённый поток должен успешно захватить монитор.

1. **Потоки (threads). Классы блокировок java.util.concurrent.locks**

ХЗ

1. **Потоки (threads). Понятие взаимной блокировки (deadlock) и способы борьбы с ней**

Взаимная блокировка – ситуация в многозадачной среде при которой несколько процессов находятся в состоянии ожидания ресурсов, занятых друг другом, и ни один из них не может продолжать свое выполнение.

Достаточно согласованного порядка захвата ресурсов, между потоками, при котором все потоки захватывают ресурсы в одном и том же порядке.

1. **Потоки (threads). Статус прерывания потока и InterruptedException**

Распространенный способ завершения потока представляет опрос логической переменной. Если она равна false, то поток завершает бесконечный цикл и заканчивает свое выполнение.

Еще один способ вызова завершения или прерывания потока представляет метод interrupt(). Вызов этого метода устанавливает у потока статус, что он прерван. Сам метод возвращает true, если поток может быть прерван, в ином случае возвращается false.

При этом сам вызов этого метода НЕ завершает поток, он только устанавливает статус: в частности, метод isInterrupted() класса Thread будет возвращать значение true. Мы можем проверить значение, возвращаемое данным методом и прозвести некоторые действия.

InterruptedException сигнализирует о том, что поток просит завершить его работу. При этом вас не просят немедленно завершить свою работу. Вас просят корректно завершить работу. На это может понадобится некоторое время. Прерывание потока осуществляется при помощи метода Thread.interrupt().

1. **Пулы потоков. Механизмы Executor и Future. Стандартная реализация пулов потоков ThreadPoolExecutor.**

Пул потоков — это набор объектов Runnable и постоянно работающих потоков. Коллекция объектов Runnable называется рабочей очередью. Постоянно запущенные потоки проверяют рабочий запрос на наличие новой работы, и если новая работа должна быть выполнена, то из рабочей очереди будет запущен объект Runnable.

Executor содержит множество функций для эффективного управления несколькими потоками. Не нужно каждый раз создавать новые потоки с помощью Executor — он позволяет использовать уже созданные потоки, когда это необходимо. В результате экономится как память Java-приложения, так и время. В данном фреймворке есть три интерфейса: Executor, ExecutorService и ScheduledExecutorService.

Executor — простой интерфейс, содержащий метод execute() для запуска задачи, заданной запускаемым объектом Runnable.

ExecutorService представляет собой суб-интерфейс Executor, который добавляет функциональность для управления жизненным циклом потоков. Он также включает в себя метод submit(), который аналогичен методу execute(), но более универсален. Перегруженные версии метода submit() могут принимать как выполняемый (Runnable), так и вызываемый (Callable) объект.

ScheduledExecutorService, в свою очередь, — это суб-интерфейс ExecutorService. Он добавляет функциональность, которая позволяет планировать выполнение задач в коде.

Future хранит результат асинхронного вычисления. Можно запустить вычисление, предоставив кому-либо объект Future, и забыть о нем. Владелец объекта Future может получить результат, когда он будет готов.

ThreadPoolExecutor — пул потоков, который содержит фиксированное количество потоков. Также этот пул можно создать с использованием конструктора через ключевое слово new.

1. **Потокобезопасные коллекции. LinkedBlockingQueue, ConcurrentHashMap. Методы Collections.synchronizedCollection.**
2. **Понятие шаблона проектирования, составляющие шаблона. Приведите примеры употребления шаблонов в неверных контекстах.**

Шаблон проектирования или паттерн - повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста. Они не влияют на структуру программной системы в целом и сохраняют независимость от реализации языка программирования.

1. **Классификация паттернов проектирования, приведите несколько примеров паттернов каждого класса.**

Есть три основных типа шаблонов проектирования: структурный, порождающий и поведенческий.

Структурные шаблоны, в общем случае, имеют дело с отношениями между объектами, облегчая их совместную работу. Например, адаптер, мост, фасад.

Порождающие шаблоны обеспечивают механизмы инстанцирования, облегчая создание объектов способом, который наиболее соответствует ситуации. Например, фабрика, строитель и абстрактная фабрика.

Поведенческие шаблоны используются в коммуникации между объектами, делая её более лёгкой и гибкой. Например, итератор, посредник и наблюдатель.

1. **Опишите паттерн синглтон, пример реализации**
2. **Опишите паттерн фабрика, пример реализации**
3. **Опишите паттерн стратегия, пример реализации**
4. **Опишите паттерн строитель, пример реализации**
5. **Опишите паттерн абстрактная фабрика, пример реализации**
6. **Опишите паттерн адаптер, его отличие от декоратора, пример реализации**
7. **Опишите паттерн декоратор, его отличие от адаптера, пример реализации**
8. **Опишите паттерн фасад, пример реализации**
9. **Что такое итератор, интерфейсы связанные с итераторами**
10. **Принципы SOLID: принцип единственной ответственности (Single responsibility principle)**
11. **Принципы SOLID: принцип открытости-закрытости (Open/Closed principle)**
12. **Принципы SOLID: принцип Барбары Лисков (Liskov substitution principle)**
13. **Принципы SOLID: принцип инверсии зависимостей (Dependency inversion principle)**