1. Многопоточность. Класс Thread, интерфейс Runnable. Модификатор synchronized

Поток — отдельная выполняемая последовательность в программе. Другим способом реализации многозадачности являются процессы. Основное отличие процессов от потоков состоит в том, что процесс обладает собственным адресным пространством и обменивается данными с другими процессами при помощи межпроцессного взаимодействия. При этом потоки одного процесса имеют общее адресное пространство и могут обращаться к одной области памяти, используя для обмена данными общие переменные. Переключения контекста потока происходит намного быстрее, чем для процесса, поэтому создание и уничтожение потоков требуют меньше времени и системных ресурсов.
Java предусматривает две возможности реализации потоков:
Класс реализует интерфейс java.lang.Runnable с определением метода run(), затем экземпляр этого класса передается в конструктор класса Thread;
Класс строится как потомок класса java.lang.Thread с переопределением метода run(), затем создается экземпляр этого класса.
Состояния потока с точки зрения виртуальной машины определены во вложенном классе Thread.State: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED.
Ключевое слово synchronized в Java используется для обеспечения потокобезопасности (thread safety). Оно гарантирует, что только один поток может выполнять определенный блок кода или метод в конкретный момент времени. Поток захватывает монитор объекта и другие потоки, пытающиеся войти в этот же синхронизированный блок или метод, блокируются до тех пор, пока первый поток не освободит монитор.

2. Методы wait(), notify() класса Object, интерфейсы Lock и Condition.

Для того, чтобы потоки могли извещать друг друга о наступлении некоторого события, например, о том, что данные подготовлены для считывания, можно использовать методы wait(), notify(), notifyAll() класса Object. Методы wait(), notify() и notifyAll() работают с использованием внутреннего монитора объекта. При вызове этих методов поток должен обладать монитором, поэтому эти методы должны всегда располагаться внутри синхронизированных блоков или методов. Метод wait() помещает поток в список ожидания объекта, после чего освобождает монитор и переходит в состояние WAITING или TIMED_WAITING. При этом метод wait() остается в незавершенном состоянии. Другие потоки могут захватить свободный монитор и начать выполнять синхронизированный блок. Как только некоторый поток совершил действие, которое ожидают другие потоки, он вызывает метод notify() или notifyAll() и выходит из синхронизированного блока, освобождая монитор. Методы notify() и notifyAll() выводят из состояния ожидания либо один, либо все потоки, которые находились в списке ожидания данного объекта. Эти потоки пытаются захватить монитор, получив который, они могут завершить выполнение метода wait() и продолжить работу.
Интерфейс Lock предназначен для реализации поведения, подобного синхронизированным методам, но с расширенными возможностями, включая неблокирующий захват блокировки, блокировку с прерыванием и блокировку с таймаутом. Методы:
Lock() — получить блокировку. Если блокировка свободна, то она захватывается текущим потоком. Если блокировка захвачена другим потоком, то текущий поток блокируется и «засыпает» до освобождения блокировки.
unlock() — освободить блокировку.
lockInterruptibly() throws InterruptedException — получить блокировку с возможностью отменить захват блокировки прерыванием потока.
tryLock() — получить блокировку, если она свободна.
Интерфейс Lock предназначен для реализации поведения, подобного синхронизированным методам, но с расширенными возможностями, включая неблокирующий захват блокировки, блокировку с прерыванием и блокировку с таймаутом. Методы:
lock() — получить блокировку. Если блокировка свободна, то она захватывается текущим потоком. Если блокировка захвачена другим потоком, то текущий поток блокируется и «засыпает» до освобождения блокировки. unlock() — освободить блокировку.
tryLock() — получить блокировку, если она свободна.
Condition newCondition() — возвращает условие, связанное с данной блокировкой
Интерфейс Condition позволяет осуществлять блокировку с ожиданием условия, подобно методам wait-notify, но опять же с расширенными возможностями, например, с возможностью иметь несколько условий для одной блокировки. Методы:
await() — заставляет текущий поток ожидать сигнала или прерывания. await(time, unit) — заставляет текущий поток ожидать сигнала, прерывания либо окончания таймаута.

3. Коллекции из пакета java.util.concurrent.

В отличие от синхронизированных коллекций, которые можно получить с помощью специальных методов класса Collections и которые блокируют доступ ко всей коллекции при чтении или записи, потокобезопасные коллекции блокируют коллекцию частично, тем самым увеличивая производительность при параллельных операциях.
ConcurrentHashMap — разрешает любое количество параллельных операций чтения и ограниченное количество параллельных операций записи (достигается изменением распределения значений в хеш-таблице).
CopyOnWriteArrayList — реализация динамического массива, при которой любая модифицирующая операция выполняется с использованием копирования массива.

4. Модификатор volatile. Атомарные типы данных и операции

Атомарные операции в Java — это операции, которые выполняются как единое неделимое действие. Это означает, что они гарантированно завершаются без прерывания другими потоками. Атомарность важна в многопоточной среде, чтобы избежать состояния гонки (race condition) и обеспечить потокобезопасность.
Модификатор volatile применяется для переменных и означает, что:
1. Переменная, объявленная volatile, не кэшируется потоком (что для обычных переменных может происходить для оптимизации), а всегда читается или записывается напрямую в память.
2. При операциях чтения-записи из нескольких потоков гарантируется, что операция записи для volatile-переменной будет завершена до операции чтения.
Операции чтения-записи для volatile-переменной всегда атомарны.
Хотя использование переменных с модификатором volatile позволяет решить проблему атомарности чтения-записи, другие операции (инкремент, декремент, сложение) остаются неатомарными. Для упрощения работы в таких случаях существует пакет java.util.concurrent.atomic. Классы, входящие в данный пакет, позволяют достичь более высокой производительности, чем использование синхронизированных блоков для обеспечения атомарности операций над различными типами данных.
AtomicInteger, AtomicLong, AtomicBoolean, AtomicReference — классы, реализующие атомарный доступ и операции для соответствующих типов:
V get() — получить значение;
set(V) — установить значение;
V getAndSet(V) — установить значение и вернуть старое.

5. Классы-сихронизаторы из пакета java.util.concurrent

Semaphore — семафор позволяет ограничить количество потоков, имеющих доступ к ресурсу. При создании семафора указывается количество разрешений.
Exchanger<V> — класс для обмена данными двух потоков. Метод V exchange(V) ожидает, когда второй поток вызовет такой же метод, после чего отдает свое значение, получая взамен значение от второго потока.
ountDownLatch — триггер с обратным отсчетом. При создании объекта данного класса указывается начальное значение счетчика. Вызов метода await() приводит к блокировке потока до тех пор, пока необходимое количество раз не будет вызван метод countDown().
CyclicBarrier — циклический барьер. При создании барьера указывается его размер. Вызов метода await() приводит к блокировке потока до тех пор, пока количество потоков, ожидающих дальнейшего продвижения, не сравняется с размером барьера. После этого потоки разблокируются.

6. Пулы потоков

FixedThreadPool — пул с фиксированным количеством потоков и общей неограниченной очередью.
WorkStealingPool — пул с изменяемым количеством потоков и несколькими очередями, обычно количество потоков равно количеству доступных процессоров.
CachedThreadPool — пул с кэшированием, по возможности повторно используются уже имеющиеся потоки. При этом потоки уничтожаются, если не использовались в течение минуты, и создаются новые по мере необходимости.
ScheduledThreadPool — пул с возможностью запуска задач с задержкой или периодических задач.
SingleThreadExecutor — однопоточный исполнитель с неограниченной очередью. В случае необходимости создается новый поток.
Кроме этого, можно использовать механизм fork-join с помощью класса ForkJoinPool и класса ForkJoinTask. Основным принципом использования механизма fork-join является выполнение задачи потоком самостоятельно, если она достаточно мала, в противном случае задача делится на две подзадачи, которые передаются на исполнение другим потокам, которые в свою очередь выполняют аналогичную операцию.

7. Интерфейсы Executor, ExecutorService, Callable, Future

Базовый интерфейс Executor выполняет предоставленную ему задачу, реализующую интерфейс Runnable. Он обеспечивает разделение добавления задачи и механизма запуска задачи на выполнение. Содержит метод execute(Runnable), который асинхронно запускает задачу на выполнение.
Executor содержит методы:
Future<T> submit(Callable<T> task) — запускает асинхронно на выполнение задачу, возвращает объект Future для получения результата.
List<Future<T» invokeAll(Collcection<Callable> tasks) — запускает на выполнение несколько задач, возвращая список объектов Future для получения их результатов после завершения всех задач.
Интерфейс Callable<V> предоставляет функциональность, аналогичную Runnable, но, в отличие от Runnable, имеет возвращаемое значение и может выбрасывать исключение. Содержит метод V call().
Интерфейс Future<V> позволяет получить результат работы задачи в будущем. Содержит методы:
V get() — блокирует текущий поток до завершения операции и возвращает значение
V get(long, TimeUnit) — блокирует поток до завершения операции или таймаута
boolean cancel(boolean) — пытается отменить задачу. Если задача еще не началась, возвращает true и успешно ее отменяет. Если задача уже завершилась, возвращает false. Если задача выполняется, то при параметре true пытается ее прервать, при параметре false разрешает ей завершиться самостоятельно.
boolean isDone() — возвращает true, если задача завершилась любым способом (нормально, в результате отмены или исключения).
boolean isCancelled() — возвращает true, если задача была отменена.

8. Интерфейсы Statement, PreparedStatement, ResultSet, RowSet

В многопоточной среде очередь играет ключевую роль для организации взаимодействия между потоками. Она позволяет потокам безопасно обмениваться данными, избегая состояния гонки (race condition) и других проблем синхронизации. В Java для работы с очередями в многопоточной среде используются специальные реализации интерфейса Queue из пакета java.util.concurrent. BlockingQueue — это подинтерфейс Queue, который предоставляет блокирующие операции для добавления и извлечения элементов. Он специально разработан для многопоточной среды.
Statement: выполнение простых SQL-запросов без параметров. Запрос передается в виде строки, каждый запрос компилируется заново при выполнении, уязвим к SQL-инъекциям, если данные подставляются напрямую.
PreparedStatement: выполнение параметризованных SQL-запросов. Наследует интерфейс Statement, запрос компилируется один раз и может выполняться многократно с разными параметрами, параметры задаются через плейсхолдеры (?), что предотвращает SQL-инъекции, поддерживает типизацию данных (методы setInt(), setString() и т.д.).
ResultSet — это интерфейс из пакета java.sql, который представляет результат выполнения SQL-запроса (например, SELECT). Он позволяет последовательно обходить строки результата запроса. Курсор указывает на текущую строку в ResultSet. По умолчанию курсор движется только вперед (TYPE_FORWARD_ONLY), но можно настроить двунаправленный курсор. Поддерживает доступ к данным по столбцам через индекс или имя.

```java
import java.sql.*;

public class ResultSetExample {
    public static void main(String[] args) {
        String url = "jdbc:postgresql://localhost:5432/mydb";
        String user = "postgres";
        String password = "password";

        try (Connection conn = DriverManager.getConnection(url, user, password);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery("SELECT id, name FROM users")) {

            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                System.out.println("ID: " + id + ", Name: " + name);
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

RowSet — это расширение интерфейса ResultSet, которое предоставляет более гибкий способ работы с данными из базы данных. В отличие от ResultSet, RowSet может работать автономно (disconnected) и поддерживает сериализацию. Автономные RowSet (например, CachedRowSet) позволяют работать с данными после закрытия соединения с базой данных.