-1. Понятие шаблона проектирования, составляющие шаблона. Приведите примеры употребления шаблонов в неверных контекстах

Шаблон проектирования или паттерн (design pattern) в разработке ПО — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Шаблон – не законченный образец, не код; это пример решения задачи, который можно использовать в различных ситуациях.

Примеры неверного употребления:

- злоупотребление Singleton. Глобальные переменные не всегда хорошо
- усложнение программы за счет паттернов
- применение паттерна без оснований (Золотой молоток, применение одного паттерна везде)
- костыли для слабого языка программирования

Нужда в паттернах появляется тогда, когда люди выбирают для своего проекта язык программирования с недостаточным уровнем абстракции. В этом случае, паттерны — это костыль, который придает этому языку суперспособности.

Ещё паттерны могут показаться похожими на алгоритмы. Но у них есть разница. Алгоритм состоит из конкретных шагов, описывающих необходимые действия. Паттерны же лишь описывают подход, но не описывают шаги реализации.

2. Классификация паттернов проектирования, приведите несколько примеров паттернов каждого класса

Разные паттерны решают разные проблемы. Обычно выделяют следующие категории:

Порождающие. Эти паттерны решают проблемы обеспечения гибкости создания объектов. (Синглтон, строитель, фабричный метод)

Структурные. Эти паттерны решают проблемы эффективного построения связей между объектами (Адаптер, мост, декоратор, фасад)

Поведенческие. Эти паттерны решают проблемы эффективного взаимодействия между объектами (Наблюдатель, итератор, хранитель)

3. Функциональные интерфейсы. Понятие функционального интерфейса и использование в программах на языке Джава

Функциональный интерфейс в Java – это интерфейс, который содержит только 1 абстрактный метод. Функциональные интерфейсы — это механизм, который дает нам возможность передавать функции в качестве параметров другим методам.

Основное назначение – использование в лямбда выражениях и method reference.

Пример:

BinaryOperator<T> принимает в качестве параметра два объекта типа T, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа T:

```
public interface BinaryOperator<T> {
    T apply(T t1, T t2);
}
```

Например:

```
import java.util.function.BinaryOperator;

public class LambdaApp {

    public static void main(String[] args) {

        BinaryOperator<Integer> multiply = (x, y) -> x*y;

        System.out.println(multiply.apply(3, 5)); // 15
        System.out.println(multiply.apply(10, -2)); // -20
    }
}
```

4. Определение и использование функциональных интерфейсов. Аннотирование функциональных интерфейсов

Аннотация @FunctionalInterface необязательная. Компилятор проверяет для интерфейсов, помеченных этой аннотацией, что в нем только один абстрактный метод.

Концептуально функциональный интерфейс имеет ровно один абстрактный метод. Поскольку методы по умолчанию имеют реализацию, они не являются абстрактными. Если интерфейс объявляет абстрактный метод, переопределяющий один из общедоступных методов java.lang.Object , это также не учитывается при под счете абстрактных методов интерфейса, поскольку любая реализация интерфейса будет иметь реализацию из java.lang.Object или где-либо еще.

5. Понятие и использование лямбда-выражений в языке Джава. Примеры

Лямбда-выражение или просто лямбда в Java — упрощённая запись анонимного класса, реализующего функциональный интерфейс.

Predicate<String> stringPredicate = (s) -> s.length() > 0;

Lambda-выражения в Java обычно имеют следующий синтаксис (аргументы) -> (тело). Например:

```
(int a, int b) -> {  return a + b; }

() -> System.out.println("Hello World");

(String s) -> { System.out.println(s); }

() -> 42

() -> { return 3.1415 };
```

Можно использовать для инициализации потока Thread, например:

```
new Thread(
    () -> System.out.println("Hello from thread")
).start();
```

Для управления событиями:

```
button.addActionListener(
```

(e) -> { System.out.println("Кнопка нажата.");

Вывод элементов массива:
List list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
list.forEach(n -> System.out.println(n));

6. Паттерн “Стратегия. Пример использования функций м для параметризации поведения объектов

Шаблон «Стратегия» относится к группе поведенческих шаблонов (т.е. решает проблему взаимодействия между объектами).

Шаблон служит для переключения между семейством алгоритмов, когда объект меняет свое поведение, на основании изменения своего внутреннего состояния. Можно использовать в играх, хранении информации и тд.

Например, сделаем интерфейс Стратегия, и реализуем различные конкретные стратегии (добавление, вычитание и умножение).

```
// Класс реализующий конкретную стратегию, должен реализовывать
этот интерфейс
//  Класс  контекста  использует  этот  интерфейс  для  вызова
конкретной стратегии
interface Strategy {
    int execute(int a, int b);
}

// Реализуем алгоритм с использованием интерфейса стратегии
class ConcreteStrategyAdd implements Strategy {

    public int execute(int a, int b) {
                System.out.println("Called  ConcreteStrategyAdd's
execute()");
        return a + b;  // Do an addition with a and b
    }
}
```

```
class ConcreteStrategySubtract implements Strategy {

    public int execute(int a, int b) {
            System.out.println("Called ConcreteStrategySubtract's
execute()");
        return a - b;  // Do a subtraction with a and b
    }
}

class ConcreteStrategyMultiply implements Strategy {

    public int execute(int a, int b) {
            System.out.println("Called ConcreteStrategyMultiply's
execute()");
        return a * b;   // Do a multiplication with a and b
    }
}
```

Теперь создадим контекст, где мы будем устанавливать и выполнять различные стратегии.

```
// Класс контекста использующий интерфейс стратегии
class Context {

    private Strategy strategy;

    // Constructor
    public Context() {
    }

    // Set new strategy
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}
```

Теперь тестирование. Создаем контекст, устанавливаем разные стратегии и смотрим результаты.

```
// Тестовое приложение
class StrategyExample {

    public static void main(String[] args) {

        Context context = new Context();

        context.setStrategy(new ConcreteStrategyAdd());
        int resultA = context.executeStrategy(3,4);
        context.setStrategy(new ConcreteStrategySubtract());
        int resultB = context.executeStrategy(3,4);

        context.setStrategy(new ConcreteStrategyMultiply());
        int resultC = context.executeStrategy(3,4);

        System.out.println("Result A : " + resultA );
        System.out.println("Result B : " + resultB );

        System.out.println("Result C : " + resultC );
    }
}
```

```
//Абстрактный класс или интерфейс определяющий общий интерфейс для всех конкретных стратегий:
public interface SortingStrategy {
    public void sort(int[] arr);
}

//Конкретные классы стратегий, реализующие интерфейс и реализующие свои алгоритмы:
public class BubbleSort implements SortingStrategy {
    public void sort(int[] arr) {
        // реализация алгоритма сортировки пузырьком
    }
}

public class QuickSort implements SortingStrategy {
    public void sort(int[] arr) {
        // реализация алгоритма быстрой сортировки
    }
}

//Класс контекста, который использует экземпляры классов стратегий для выполнения определенных действий:
public class ArraySorter {
    SortingStrategy sortingStrategy;

    public ArraySorter(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void sortArray(int[] arr) {
        sortingStrategy.sort(arr);
    }
```

```
}

// Использование классов:
public static void main(String[] args) {
    int[] arr = {5,2,7,1,9,3};
    ArraySorter sorter = new ArraySorter(new BubbleSort()); // используем стратегию
сортировки пузырьком
    sorter.sortArray(arr);
}
```

7. Понятие чистой функции. Побочные эффекты функций. Чистота функций в ООП. Пример

Чистая функция - функция без побочных эффектов, результат которой зависит только от значений ее параметров.

Побочные эффекты функции - то, что делает функция кроме вычисления результата функции, например, изменение значений полей. (Ввод/вывод также является побочным эффектом) Часто повторный вызов функции с побочным эффектом производит результат, отличный от предыдущего вызова.

Функция с побочным эффектом:

```
class Point {
   int x;
   int y;
   void move(int dx) {
      this.x += dx; // побочный эффект
   }
   int getX() { return x; } // не является чистой, т.к. разные вызовы
                            // могут вернуть разные значения
}
```

Чистая функция:

```
class Point {
   private int x;
   private int y;
   Point(int x, int y) { this.x = x; this.y = y; }
   Point move(int dx) {
      return new Point(this.x + dx, y); // нет побочных эффектов
   }
   int getX() { return x; }
}
```

И move, и getX – чистые функции (в данном случае this мы считаем неявным параметром функции).

8. Неизменяемый класс. Преимущества использования неизменяемых классов. Пример

В ООП чистота функций (методов) достигается с помощью использования неизменяемых (immutable) классов. Все методы неизменяемого класса – чистые, после создания объекта неизменяемого класса его состояние больше не меняется. При операциях с такими объектами создаются новые объекты, а не меняются существующие.

Для того, чтобы гарантировать неизменяемость свойств объекта, следует использовать модификатор **final** для полей, также неизменяемый класс обычно закрыт для наследования, т.к. иначе класс-наследник может добавить изменяемые поля.

Пример описания неизменяемого класса:

```
public final class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Обратите внимание:

Этого недостаточно – все поля класса должны также быть неизменяемыми или недоступными извне:

```
public final class Mutable {
    private final int[] arr;
    public Mutable(int[] arr) { this.arr = arr; }
}
int[] array = {1, 2, 3};
Mutable mut = new Mutable(array);
array[1] = 20; // Состояние mut изменилось!
```

Многие стандартные классы являются неизменяемыми:

1. String
2. Byte/Short/Integer/Long/Character/Boolean/Double
3. java.util.Optional
4. Классы пакета java.time
5. java.nio.file.Path
6. java.nio.charset.Charset
7. java.net.URI

Функциональное программирование

В Java начиная с версии 16 добавлена возможность краткой записи простых неизменяемых классов:

```
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) {
      this.x = x; this.y = y;
   }
   public int x() {
      return x;
   }
   public int y() {
      return y;
   }
}
```

```
public record Point(int x, int y) {
}
```

Кроме того, объявление record определяет методы equals, hashCode и toString с использованием значений полей класса (их можно переопределить вручную).

9. Использование неизменяемых полей класса. Преимущества неизменяемых объектов

Преимущества неизменяемых объектов:

1. Проще понять работу класса:
   1.1. Отсутствует “действие на расстоянии”
   1.2. Результат работы программы не зависит от истории
   1.3. Проще проверять инварианты класса
2. Проще тестировать класс
3. Проще использовать в многопоточной среде
4. Можно использовать в качестве ключей Map.

“Действие на расстоянии”:

```
class Point {
    int x;
    int y;
}
Point p1 = new Point(0, 0);
Point p2 = p1;
...
p2.x = 10; // p1.x меняется тоже!
```

Так как действие на расстоянии по определению не локально, то его невозможно отследить, рассматривая только один изолированный участок кода. Иногда действие на расстоянии полезно, но чаще – нет.

“Зависимость от истории”:

```
class MyClass {
    private ImportantThing thing;
    void init(ImportantThing thing) {
        this.thing = thing;
    }
    void work() {
        thing.doWork();
    }
}
```

```
MyClass obj = new MyClass();
obj.init(new ImportantThing());
// Если мы забудем вызов init, то здесь будет ошибка:
obj.work();
```

Т.е. работоспособность метода зависит от того, какие методы мы вызывали перед этим.

Для неизменяемого класса мы не можем забыть инициализацию из-за модификатора **final:**

```
class MyClass {
  private final ImportantThing thing;
  MyClass(ImportantThing thing) {
    this.thing = thing;
  }
  void work() {
    importantThing.doWork();
  }
}
```

Благодаря отсутствию зависимости от истории вызванных методов неизменяемые классы проще тестировать – не нужно перед вызовом тестируемого метода вызывать методы, которые приводят объект в нужное состояние – он сразу инициализирован в правильном состоянии.

10. Функциональное программирование. Понятие инварианта класса. Примеры

Инвариант класса – условие, которое должно выполняться на протяжении всего срока жизни объекта. В случае изменяемого объекта инвариант нужно проверять в каждом методе, изменяющем состояние (mutator method). Если это забыть сделать, инвариант может быть нарушен. В неизменяемом же классе инвариант достаточно проверять только в конструкторе.

(на скриншоте показан вариант для изменяемого класса – в каждом методе, изменяющем состояние, перед его выполнением проводится проверка инварианта)

```
// Инвариант: a < b
class OrderedPair {
   private int a;
   private int b;

   OrderedPair(int a, int b) {
      check(a < b);
      this.a = a;
      this.b = b;
   }

class OrderedPair {
   ...
   void setA(int newA) {
      check(newA < b);
      this.a = newA;
   }
   void setB(int newB) {
      check(a < newB);
      this.b = newB;
   }
```

11. Особенности многопоточной работы в Джава, использование final для полей данных для обеспечения потокобезопасности

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса. Следует отдельно обговорить фразу «параллельно с другими потоками». Известно, что на одно ядро процессора, в каждый момент времени, приходится одна единица исполнения. То есть одноядерный процессор может обрабатывать команды только последовательно, по одной за раз (в упрощенном случае). Однако запуск нескольких параллельных потоков возможен и в системах с одноядерными процессорами. В этом случае система будет периодически переключаться между потоками, поочередно давая выполняться то одному, то другому потоку. Такая схема называется псевдо-параллелизмом. Система запоминает состояние (контекст) каждого потока, перед тем как переключиться на другой

поток, и восстанавливает его по возвращению к выполнению потока. В контекст потока входят такие параметры, как стек, набор значений регистров процессора, адрес исполняемой команды и прочее… Проще говоря, при псевдопараллельном выполнении потоков процессор мечется между выполнением нескольких потоков, выполняя по очереди часть каждого из них.

```
public class Program        //Класс с методом main().
{
    public static void main(String[] args)
    {
        //Создание потока
        Thread myThready = new Thread(new Runnable()
        {
            public void run() //Этот метод будет выполняться в побочном потоке
            {
                System.out.println("Привет из побочного потока!");
            }
        });
        myThready.start();  //Запуск потока

        System.out.println("Главный поток завершён...");
    }
}
```

В Java процесс завершается тогда, когда завершается последний его поток. Даже если метод main() уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.

В языке Java модификатор final для полей имеет особый смысл при многопоточной работе: к полю с этим модификатором могут безопасно обращаться одновременно несколько потоков и они будут гарантированно видеть одинаковое значение, в то время как для не-final полей без особой синхронизации потоков может быть так, что каждый поток видит своё значение поля.

12. Основное назначение паттерна “Строитель” (Builder) для разработки программ на языке Джава

Одно из неудобств неизменяемых классов – необходимость создавать объект в “готовом” состоянии через конструктор. Для сложных объектов может понадобиться передать в конструктор много параметров, а в Java значений по умолчанию нет.

Паттерн **Builder** позволяет использовать изменяемый объект для задания свойств, а затем создания на их основе неизменяемого объекта. Как правило, изменяемый Builder существует в рамках только одного метода и снаружи не виден, так что это не нарушает функциональной чистоты.

Пример реализации паттерна:

```
class MyServerBuilder {
   private int port = 80;
   private String protocol = “http”;
   private Path rootDir = Path.of(“web”);
   void setPort(int port) { this.port = port; }
   void setProtocol(String protocol) { this.protocol = protocol; }
   void setRootDir(Path rootDir) { this.rootDir = rootDir; }
   MyServer build() {
      return new MyServer(port, protocol, rootDir);
   }
}
```

Использование:

```
MyServerBuilder builder = new MyServerBuilder();
builder.setPort(8080);
MyServer server = builder.build();
```

Вариация этого подхода – Fluent Builder, где каждый setter возвращает this:

```
class MyServerBuilder {
   MyServerBuilder setPort(int port) {
      this.port = port;
      return this;
   }
   ...
}
MyServer server = new MyServerBuilder()
   .setPort(8080)
   .build();
```

13. Стандартные функциональные интерфейсы в Джава и их методы

Стандартные функциональные интерфейсы и их методы:

1. Function: R apply(T arg)
2. Supplier: T get()
3. Consumer: void accept(T arg)
4. Predicate: boolean test(T arg)
5. BiFunction: R apply(T arg1, U arg2)
6. BiConsumer: void accept(T arg1, U arg2)
7. BiPredicate: boolean test(T arg1, U arg2)

Также есть варианты этих интерфейсов для примитивных типов int, long и double.

- Function: представляет функцию перехода от объекта типа T к объекту типа R
- Supplier: не принимает никаких аргументов, но должен возвращать объект типа T
- Consumer: выполняет некоторое действие над объектом типа T, при этом ничего не возвращая
- Predicate: проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T

14. Потоки Stream API в Джава и их использование

Внимание! Thread это поток выполнения, а Stream это поток данных

Поток (Stream) – это представление последовательности элементов, над которым можно производить операции.

Операции бывают нетерминальные (их результат тоже является потоком, и к нему в свою очередь можно также применить операцию) и терминарные (их результат уже не является потоком, и эти операции завершают работу с потоком:

Условие: дана коллекция строк Arrays.asList(«a1», «a2», «a3», «a1»), давайте посмотрим как её можно обрабатывать используя методы filter, findFirst, findAny, skip и count:

Задача	Код примера	Результат
Вернуть количество вхождений объекта «a1»	collection.stream().filter(«a1»::equals).count()	2
Вернуть первый элемент коллекции или 0, если коллекция пуста	collection.stream().findFirst().orElse(«0»)	a1
Вернуть последний элемент коллекции или «empty», если коллекция пуста	collection.stream().skip(collection.size() — 1).findAny().orElse(«empty»)	a1
Найти элемент в коллекции равный «a3» или кинуть ошибку	collection.stream().filter(«a3»::equals).findFirst().get()	a3
Вернуть третий элемент коллекции по порядку	collection.stream().skip(2).findFirst().get()	a3
Вернуть два элемента начиная со второго	collection.stream().skip(1).limit(2).toArray()	[a2, a3]
Выбрать все элементы по шаблону	collection.stream().filter((s) -> s.contains(«1»)).collect(Collectors.toList())	[a1, a1]

15. Статический импорт и его использование для программирования Stream API

Статический импорт выглядит как *import static java.lang.Math.sqrt*;

Оператор import, предваряемый ключевым словом static, можно применять для импорта статических членов класса или интерфейса. Благодаря статическому импорту появляется возможность ссылаться на статические члены непосредственно по именам, не уточняя их именем класса. То есть после такого импорта sqrt нам больше не нужно писать Math.sqrt(16). Пишем просто sqrt(16).

```
import static java.util.stream.Collectors.toSet;
import static java.util.stream.Stream.of;
import static java.util.stream.Stream.concat;
import static java.util.stream.Stream.empty;

public class Test {

    public static void main(String[] args) {
        Set<Integer> set =
        concat(
                concat(
                        of(1, 2, 3, 4),
                        of(5, 6, 7, 8)),
                        empty()
        )
                .collect(toSet());

    }
}
```

16. Назначение метода stream() интерфейса Collection. Примеры

Назначение - создание потока элементов коллекции.

```
Collection<String> collection =
Arrays.asList("a1", "a2", "a3");

Stream<String>   streamFromCollection   =
collection.stream();
```

```
List<String> names = Arrays.asList(“Иван”, “Джон”, “”, “Хуан”);
// Нужно составить строку с приветствием непустых имен:
String greetings = names.stream()
    .filter(name -> !name.isEmpty())
    .map(name -> String.format(“Привет, %s!”, name));
    .collect(joining(“ “));
```

17. Работа со Stream API. Нетерминальные операции потока Stream:, назначение и использование

Нетерминальные операции потока Stream возвращают так же поток, к которому в свою очередь тоже можно применить операцию.

filter	Отфильтровывает записи, возвращает только записи, соответствующие условию	collection.stream().filter(«a1»::equals).count()
skip	Позволяет пропустить N первых элементов	collection.stream().skip(collection.size() — 1).findFirst().orElse(«1»)
distinct	Возвращает стрим без дубликатов (для метода equals)	collection.stream().distinct().collect(Collectors.toList())
map	Преобразует каждый элемент стрима	collection.stream().map((s) -> s + "_1").collect(Collectors.toList())
peek	Возвращает тот же стрим, но применяет функцию к каждому элементу стрима	collection.stream().map(String::toUpperCase).peek((e) -> System.out.print("," + e)). collect(Collectors.toList())
limit	Позволяет ограничить выборку определенным количеством первых элементов	collection.stream().limit(2).collect(Collectors.toList())
sorted	Позволяет сортировать значения либо в натуральном порядке, либо задавая Comparator	collection.stream().sorted().collect(Collectors.toList())
mapToInt, **mapToDouble**, **mapToLong**	Аналог map, но возвращает числовой стрим (то есть стрим из числовых примитивов)	collection.stream().mapToInt((s) -> Integer.parseInt(s)).toArray()

18. Работа со Stream API. Терминальные операции, назначение и использование. Примеры

Терминальные операции потока Stream возвращают уже не поток, а завершают работу с ним. То есть, это последняя операция при работе с потоком.

findAny	Возвращает любой подходящий элемент из стрима (возвращает Optional)	collection.stream().findAny().orElse(«1»)
collect	Представление результатов в виде коллекций и других структур данных	collection.stream().filter((s) -> s.contains(«1»)).collect(Collectors.toList())
count	Возвращает количество элементов в стриме	collection.stream().filter(«a1»::equals).count()

19. Интерфейс Splititerator и его методы

Сплитератор — это интерфейс, который используется для параллельного программирования, последовательная и параллельной обработки данных.

Интерфейс Spliterator предоставляет итератор для Stream API в Java. Он позволяет эффективно обрабатывать коллекции параллельно.

§

Реальный класс Stream работает похоже, но использует не Iterator, а Spliterator:

```
public interface Spliterator<T> {
   boolean tryAdvance(Consumer<T> action);
   Spliterator<T> trySplit();
   long estimateSize();
}
```

Методы:

- *tryAdvance*: полный аналог методов hasNext+next из Iterator; объединены в один, так как так его проще реализовывать (см. пример)
- *estimateSize*: может использоваться для оптимизации; например, в нашем MyStream мы могли бы его использовать для начального размера списка в методе toList
- *trySplit*: используется для возможности параллельного обхода коллекции

Довольно значительная часть особенностей Stream API связана с тем, что в них реализована возможность параллельной обработки элементов в нескольких потоках (threads). Так можно ускорить обработку данных:

```
List<String> list = Arrays.asList("A", "B");
int sumLen = list.stream().parallel().mapToInt(str -> {
   Thread.sleep(1000); return str.length();
}).sum();
// Выполняется 1 секунду, а не 2: см. код
```

Интерфейс Spliterator редко используется напрямую в программах, хотя и является основой для Stream.

Из Spliterator можно создать Stream:

```
Stream<T> StreamSupport.stream(
   Spliterator<T> spliterator,
   boolean parallel)
```

И наоборот, из Stream получить Spliterator можно через метод Stream.spliterator().

При параллельной обработке stream вызывает метод Spliterator.trySplit, который делит последовательность элементов на две примерно равных под-последовательности (если это невозможно, например если в последовательности только один элемент, trySplit возвращает null). Затем каждую под-последовательность Stream обрабатывает в своем собственном thread. Это разбиение может продолжаться и далее, пока не будет достигнуто оптимальное количество threads.

20. Понятие многопоточности и написание многопоточных программ на языке Джава

Смотри билет 11

21. Потоки (threads) в Джава. Создание потоков в Джава программах Подходы к созданию потоков: через наследование и реализацию интерфейса Runnable Thread API. Запуск потока с Runnable

Thread и Stream в джава - совершенно разные потоки. Stream - поток данных, Thread - поток выполнения кода.

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.
Поток в Java представлен в виде экземпляра класса java.lang.Thread
Ну и стоит понимать, что сами по себе Thread нельзя назвать потоками(то есть напрямую они не взаимодействуют) - это некоторый api для потоков которыми управляет jvm.
Стоит добавить что jvm сама создает поток main - главный поток , и несколько служебных

Создание поток в джаве делится на 2 типа
1 - Thread Api через наследование

- Thread thread = new Thread();
- thread.start();

```
public class MyThread extends Thread {
  public void run(){
    System.out.println("MyThread running");
  }
}
```

```
MyThread myThread = new MyThread();
myTread.start();
```

```
Thread thread = new Thread(){
  public void run(){
    System.out.println("Thread Running"); } }
thread.start();
```

2 - Реализация интерфейса Runnable -

- public interface Runnable() { public void run(); }

Класс Java реализует Runnable

```
public class MyRunnable implements Runnable {
   public void run(){
       System.out.println("MyRunnable running");
} }
```

Анонимная реализация Runnable

```
Runnable myRunnable = new Runnable(){
    public void run(){
        System.out.println("Runnable running");
} }
```

Запуск потока при помощи Runnable:

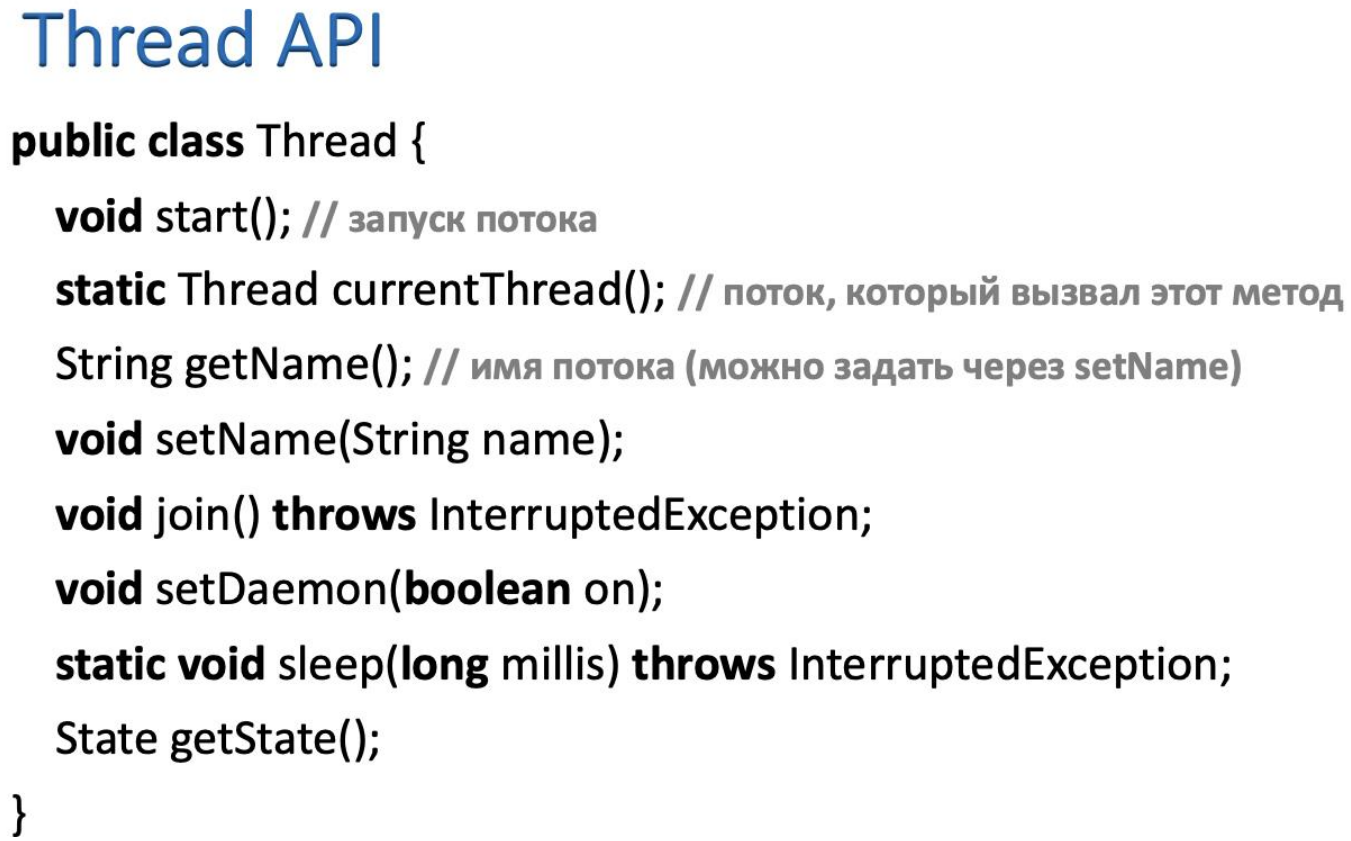
Вообще runnable - это интерфейс (задача которую выполняет поток)
Интерфейс содержит основной метод run() — в нём и находится точка входа и логика исполняемого потока.

```
class MyThread implements Runnable {
    @Override
    public void run() {
        System.out.println("Hello from MyThread!");
    }
}
public class Main {
    public static void main(String[] args) {
        Thread myThread = new Thread(new MyThread());
        myThread.start();
    }
}
```

В коде мы создали метод run, который будет выполняться в отдельном потоке. После чего создали поток, вызвали метод start - который создает поток и запускает метод run

22. Потоки в Джава. Методы Thread API

Про потоки см 21 вопрос.
Основные методы **Thread API** из лекции Мирэа :



23. Синхронизация потоков. Понятие синхронизации. Блок синхронизации

Про потоки см. вопросы 21-22.

Фактически, любая команда в java на самом деле состоит из нескольких и не является атомарной. Таким образом, при обращении нескольких потоков к одним данным результат непредсказуем, т.к. микрокоманды из разных потоков не синхронизированы между собой.

Состояние гонок – это одновременный вызов в потоках исполнения одного и того же метода для того же самого объекта.

Пример работы потоков без синхронизации:

Код первой нити

```
1  System.out.print ("Коле");
2  System.out.print ("");
3  System.out.print ("15");
4  System.out.print ("");
5  System.out.print ("лет");
6  System.out.println ();
```

Код второй нити

```
1  System.out.print ("Лене");
2  System.out.print ("");
3  System.out.print ("21");
4  System.out.print ("");
5  System.out.print ("год");
6  System.out.println ();
```

Ожидаемый вывод на консоль

```
Коле 15 лет
Лене 21 год
```

У нас есть два потока, каждый из которых выполняет код выше. Ниже показан ожидаемый вывод. Но реальный вывод на рисунке ниже.

Реальный вывод на консоль

```
Коле Лене  15 21 лет
год
```

Общим ресурсом потоков в данном случае является консоль.

Мьютекс – специальный объект для синхронизации потоков, управляемый JVM, который есть у каждого объекта и класса.

Монитор – надстройка над мьютексом, позволяющая обеспечить синхронизацию. По сути, он дает право пользования ресурсом только одному потоку.

Синхронизация это процесс, который позволяет выполнять все параллельные потоки в программе синхронно

Для работы с монитором используется несколько технологий, представленных ниже

Ключевое слово Synchronized

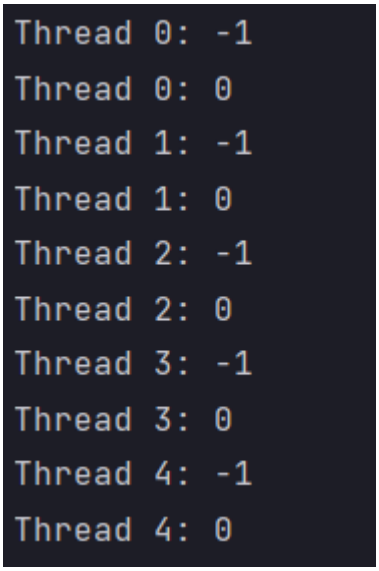
Synchronized - ключевое слово, помечающее определенный блок кода (*блок синхронизации*), при котором монитор захватывает объект, которому данный блок принадлежит. При этом потоки должны сначала захватить мьютекс (право использования) объекта, зайдя внутрь кода помеченного synchronized. В это время другие потоки ждут, пока мьютекс освободится, чтобы захватить его и начать выполнять код.

Таким образом, т.к. захватывается мьютекс всего объекта, в другие synchronized-методы класса остальные потоки заходить также не могут, т.е два потока не могут одновременно выполнять разные synchronized-методы одного класса. Пример приведен ниже - пока один поток не выполнит все синхронизированные методы класса, право другому потоку он не передаст

```
4 usages
class SynchronizedCounter {
    3 usages
    private int c = 0;

    1 usage
    public synchronized void increment() {
        c++;
    }

    1 usage
    public synchronized void decrement() {
        c--;
    }

    2 usages
    public synchronized int value() {
        return c;
    }
}
```

```java
1 usage
class MyThread extends Thread{
    5 usages
    SynchronizedCounter sc;
    3 usages
    String name;

    1 usage
    public MyThread(SynchronizedCounter sc, String name) {
        this.sc = sc;
        this.name=name;
    }

    @Override
public void start(){
        sc.decrement();
        System.out.print(name + ": ");
        System.out.println(sc.value());
        sc.increment();
        System.out.print(name + ": ");
        System.out.println(sc.value());
    }
}

public class Main {
    public static void main(String[] args) {
        SynchronizedCounter sc = new SynchronizedCounter();
        for(int i=0; i<5; i++){
            new MyThread(sc,  name: "Thread "+ i).start();
        }
    }
}
```

Результат выполнения приведен ниже



Имплементация класса Lock

Lock также блокирует определенный ресурс, но управляется вручную и используется для более точечной настройки. Так, например, можно заблокировать ресурс только для чтения или только для записи (ReadLock и WriteLock).

Использование Semaphore

Semaphore – класс, который принимает количество возможных разрешений. Когда количество разрешений заканчивается, следующий Thread, пытающийся его получить, блокируется.

По большому счету это счетчик, который можно увеличивать и уменьшать из разных потоков. Уменьшение до 0 блокирует уменьшающий поток. Состояние, когда счетчик больше нуля называют *сигнальное состояние*, операцию его увеличения – *release* (освобождение) или *signal*, уменьшения – *acquire* (захват) или *wait*.

Пример:

```
private static final Semaphore semaphore = new Semaphore(1);
static void increment() {
try {
semaphore.acquire();
buf++;
semaphore.release();
} catch (InterruptedException e) {
e.printStackTrace();
}
}
```

Потокобезопасные коллекции
По умолчанию джава содержит 2 вида таких коллекций:
– synchonized коллекции (Collections.synchronizedList());

– конкурентные коллекции (ConcurrentHashMap)

24. Использование синхронизации потоков для пары процессов producer/reader (производители/потребители) Проблема производителя-потребителя

Reader - также consumer и потребитель, совместно используют общий буфер фиксированного размера , используемый в качестве очереди. Задача производителя - сгенерировать данные, поместить их в буфер и начать заново. В то же время потребитель потребляет данные (т.е. удаляет их из буфера) по частям. Проблема состоит в том, чтобы убедиться, что производитель не будет пытаться добавить данные в буфер, если он заполнен, и что потребитель не будет пытаться удалить данные из пустого буфера.

Решения

1. (плохое) - ввод переменной *count*

В этой ситуации оба процесса могут попасть в состояние ожидания, если пропадет сигнал активации.

Алгоритм такой ситуации:

1. Процесс **B**, считал count=0 (заблокироваться он еще не успел)
2. Планировщик передал управление процессу **A**
3. Процесс **A**, выполнил все вплоть до wakeup, пытаясь разблокировать процесс **B** (но он не заблокирован, wakeup срабатывает впустую)
4. Планировщик передал управление процессу **B**
5. И он заблокировался, и больше сигнала на разблокировку не получит
6. Процесс **A** в конце концов заполнит буфер и заблокируется, но сигнала на разблокировку не получит.

2. Использование семафоров: https://for-each.dev/lessons/b/-java-producer-consumer-problem

25.

Синхронизация джава реализована на основе мониторов. *Монитор* — это объект, который используется для *взаимоисключающей блокировки*. Взаимоисключающая блокировка позволяет владеть монитором только одному объекту-потоку. Каждый объект-поток имеет собственный, неявно связанный с ним, монитор.
В Java блок synchronized использует мониторы для синхронизации доступа к общим ресурсам или критическим участкам кода.

• Монитор может быть в двух состояниях: свободен (released) и захвачен (acquired)

• При входе в блок synchronized происходит попытка захвата монитора:

• Если монитор свободен, то он становится захвачен

• Если монитор уже захвачен другим потоком, то текущий поток останавливается и ждет, пока другой поток не освободит монитор

• Если монитор захвачен текущим потоком, то он остается захвачен • При выходе из блока synchronized монитор освобождается
• Если при этом другие потоки ждали освобождения этого монитора,то выбирается один из этих потоков, который захватывает монитор и входит в свой блок synchronized

26. Метод synchronized (lock), особенности его использования для потоков

Lock - это монитор. Монитор - это специальный объект, который следит за "состоянием" метода или объекта. Он смотрит, "занят" он или "свободен" в данный момент.

Синхронизация потоков

Блок синхронизации:

```
synchronized (lock) { // lock может быть любым объектом
   // код блока
}
```

1. Блок синхронизации для одного и того же lock может выполнять одновременно только один поток
2. Если поток 2 выполняет блок синхронизации с lock после того, как поток 1 выполнил блок синхронизации с lock, то поток 2 “увидит” все изменения, внесенные потоком 1.

Синхронизация потоков

Пример:

```
synchronized (lock) {
   counter++;
}
```

<table>
<tr><th>Поток 1</th><th>Поток 2</th></tr>
<tr><td colspan="2">counter = 0</td></tr>
<tr><td>counter = counter + 1</td><td rowspan="2">(параллельное выполнение невозможно в блоке synchronized)</td></tr>
<tr><td>counter = 0 + 1</td></tr>
<tr><td colspan="2">counter = 1</td></tr>
<tr><td rowspan="2">(параллельное выполнение невозможно в блоке synchronized)</td><td>counter = counter + 1</td></tr>
<tr><td>counter = 1 + 1</td></tr>
<tr><td colspan="2">counter = 2</td></tr>
</table>

Visibility: Поток 2 гарантированно увидит изменения, внесенные потоком 1

27. Синхронизация потоков. Правила happens-before (hb)

Синхронизация потоков Правила happens-before (hb):
1. В рамках одного потока любая операция happens-before любой операцией следующей за ней в исходном коде
2. Выход из synhronized блока happens-before входа в synhronized блок на том же мониторе
3. Запись volatile поля happens-before чтение того же самого volatile поля
4. Завершение метода run экземпляра класса Thread happens-before выхода из метода join()
5. Вызов метода start() экземпляра класса Thread happensbefore начало метода run() экземпляра того же треда

Связь happens-before транзитивна, т.е. если X happens-before Y, а Y happens-before Z, то X happens-before Z. Если X happens-before Y, то все изменения, внесенные до операции X, будут видны в коде, следующем за операцией Y.

28. Синхронизация потоков. Модификатор полей volatile и его использование

Модификатор полей volatile:

private static volatile boolean ready = **false;**

Правила happens-before гарантируют, что при чтении volatile поля мы читаем последнее записанное значение.

Проще говоря, volatile означает что при обращении нескольких потоков к полю, будет прочитано последнее изменение.
Он гарантирует, что изменения, внесенные в это поле одним потоком, будут видны другим потокам, работающим с этим полем.

- Атомарные операции широко используются в многопоточных приложениях, где несколько потоков должны работать с одними и теми же общими ресурсами. Без использования атомарных операций возникают проблемы несогласованности данных и состояний, что может приводить к ошибкам в работе программы.

29. Синхронизация потоков. Атомарные операции. Механизм Wait/notify

Атомарная операция — это операция, которая выполняется полностью или не выполняется совсем, частичное выполнение невозможно.

Это означает, что операции являются неделимыми, и если они выполняются параллельно несколькими потоками, гарантируется, что каждый поток видит результат выполнения операции в целостности и не сможет подвергнуть его изменениям в процессе выполнения.

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса **Object** определено ряд методов:

- **wait()**: освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
- **notify()**: продолжает работу потока, у которого ранее был вызван метод wait()
- **notifyAll()**: возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

Метод lock.wait() освобождает монитор lock и переводит текущий поток в режим ожидания монитора (состояние WAITING). Так как монитор освобожден, другие потоки могут выполнять блоки “synchronized (lock)”, пока этот поток находится в режиме ожидания (“спит”).

Метод lock.notifyAll() будет все ожидающие этот монитор потоки. Все эти потоки начинают пытаться захватить монитор (это получится не сразу после вызова notifyAll, а только когда вызвавший notifyAll поток выйдет из блока synchronized). После этого все спавшие потоки по очереди захватят монитор и выполнят блок synchronized.

30. Синхронизация потоков с использованием классов пакета java.util.concurrent.locks

В пакете java.util.concurrent.locks есть классы для расширенной поддержки синхронизации. Класс ReentrantLock является аналогом блока synchronized с некоторыми дополнительными функциями. В новых версиях Java рекомендуется использовать его вместо synchronized. Класс ReentrantReadWriteLock позволяет блокировать потоки только в случае, когда идет изменение данных, а в случае чтения данных блокировки потоков не происходит.

31. Запуск и прерывание потоков. Приостановка и прерывание выполнения нити

Для запуска потока нужно использовать команду start:
new JThread("JThread").start();

Для завершения, во-первых, должны соблюдаться правила happens-before. Смотреть 27 вопрос

https://metanit.com/java/tutorial/8.4.php - ссылка на источник

Распространенный способ завершения потока представляет опрос логической переменной. И если она равна, например, false, то поток завершает бесконечный цикл и заканчивает свое выполнение. Пишем метод, далее, когда необходимо сбрасываем флаг, вызываем исключение - поток завершён .

Thread.yield()

Или через метод interrrupt(), используя исключение. При этом сам вызов этого метода НЕ завершает поток, он только устанавливает статус.

Бывает случай, когда Main thread завершается самым последним. Для этого надо применить метод join(). В этом случае текущий поток будет ожидать завершения потока, для которого вызван метод join, а после завершится.

32. Обработка операции прерывания потока

Обработка операции прерывания потока в Java возможна с помощью метода interrupt(), который устанавливает флаг прерывания для потока. Что должен делать поток в ответ на прерывание, решает программист, но обычно поток завершается. Поток отправляет прерывание вызывая метод public void interrupt() класса Thread. Для того чтобы механизм прерывания работал корректно, прерываемый поток должен поддерживать возможность прерывания своей работы.

Как поток должен поддерживать прерывание своей работы? Это зависит от того, что он сейчас делает. Если поток часто вызывает методы, которые могут бросить InterruptedException, то он просто вызывает return при перехвате подобного исключения.

33. Ожидание и присоединение запущенной нити основным потоком управления

Для ожидания завершения выполнения потока и присоединения его к основному потоку управления в Java можно использовать метод join(). Метод join() представляет собой блокирующий вызов, который приостанавливает выполнение текущего потока, пока поток, вызвавший метод join(), не завершит свое

выполнение. То есть, при вызове метода join() основной поток будет ожидать завершения выполнения указанной нити.

```
public class MyThread extends Thread {
  public void run() {
    // выполнение задачи
  }
}
// создание объекта потока и запуск его выполнения
MyThread thread = new MyThread();
thread.start();

try {
  // ожидание завершения выполнения потока и присоединение к основному потоку
  thread.join();
} catch (InterruptedException e) {
  // обработка исключения
}
```

34. Жизненный цикл потока на языке Джава. Состояние потока

Жизненный цикл потока в Java в основном состоит из переходов в различные состояния, начиная с рождения потока (thread birth) и заканчивая его завершением (thread termination).
Поток — это путь выполнения программы, который может войти в одно из пяти состояний в течение своего жизненного цикла:

1. New
2. Runnable
3. Running
4. Blocked
5. Dead

New (новорожденное состояние, newborn state) возникает, когда вы создаете объект Thread в классе Thread. Поток создается и находится в “новорожденном” состоянии. То есть при возникновении потока он входит в новое состояние, но метод start() при этом еще не вызван для экземпляра.
2. Runnable. Это состояние означает, что поток готов к выполнению. Когда метод start() вызывается для нового потока, он становится готовым к запуску. В данном состоянии поток ожидает, пока процессор не станет доступным (CPU time, процессорное время). То есть поток становится в очередь (серию) потоков, ожидающих выполнения.
3. Running (состояние выполнения). Выполнение означает, что процессор выделил временной интервал для выполнения потока. Это состояние, в котором поток выполняет свою фактическую функцию.
4. Blocked (заблокированное состояние). Поток находится в заблокированном состоянии, когда он приостанавливается, спит или ждет некоторое время, чтобы удовлетворить поставленное условие.
5. Dead State. Это состояние возникает, когда метод run() завершает выполнение инструкций. Поток автоматически останавливается или переходит в мертвое состояние (Dead State). Иными словами, когда поток выходит из метода run(), он либо завершается, либо переходит в состояние dead.

35. Многопоточные примитивы и их использование

Многопоточные примитивы в Java представляют собой инструменты, предназначенные для управления и синхронизации параллельных потоков выполнения в многопоточных приложениях. Они обеспечивают безопасное и эффективное взаимодействие между потоками, предотвращают состояния гонки (race conditions) и обеспечивают согласованность данных.

В Java наиболее распространенными многопоточными примитивами являются:

Мьютексы (Monitors): Мьютекс (Mutex) является примитивом синхронизации, который позволяет только одному потоку за раз получить доступ к общему ресурсу. В Java мьютексы реализуются с помощью ключевого слова synchronized и блоков synchronized.

Взаимные исключения (Locks): Взаимные исключения позволяют более гибко управлять доступом к общим ресурсам, чем мьютексы. В Java для реализации взаимного исключения предоставляется интерфейс Lock и его реализации, такие как ReentrantLock.

Условные переменные (Condition Variables): Условные переменные используются для ожидания определенного условия или оповещения других потоков о возникновении определенного события. В Java условные переменные предоставляются интерфейсом Condition, который связан с объектом блокировки.

Счетчики (Countdown Latches): Счетчики используются для ожидания завершения определенного числа операций перед продолжением выполнения потока. В Java CountDownLatch предоставляет эту функциональность, позволяя потокам ждать, пока не будет достигнуто заданное число сигналов.

Барьеры (Barriers): Барьеры позволяют потокам остановиться и ждать, пока все потоки не достигнут определенной точки выполнения, после чего продолжить выполнение. В Java CyclicBarrier и Phaser предоставляют возможность создания барьеров.

Атомарные переменные (Atomic Variables): Атомарные переменные гарантируют атомарные операции чтения и записи без необходимости использования блокировок. В Java атомарные переменные предоставляются классами AtomicInteger, AtomicLong, AtomicBoolean и другими.

36. Интерфейс Executor в Джава и его использование. ExecutorService

Интерфейс Executor в Java представляет собой базовый интерфейс для выполнения задач в асинхронном режиме. Он определяет единственный метод execute(Runnable command), который принимает объект типа Runnable и выполняет его в фоновом потоке.

Executor является основой для более высокоуровневого интерфейса ExecutorService, который предоставляет дополнительные функциональности, такие как пул потоков, позволяющий повторно использовать потоки для выполнения задач.

ExecutorService предоставляет следующие методы для управления выполнением задач:

- submit(Runnable task): Подает задачу на выполнение и возвращает объект Future, который представляет результат выполнения задачи или позволяет отменить её выполнение.
- submit(Callable<T> task): Подает задачу, представленную в виде объекта Callable, на выполнение и возвращает объект Future<T>, который представляет результат выполнения задачи или позволяет отменить её выполнение.
- invokeAny(Collection<? extends Callable<T>> tasks): Подает коллекцию задач типа Callable на выполнение и возвращает результат первой выполненной задачи. Остальные задачи прерываются.
- invokeAll(Collection<? extends Callable<T>> tasks): Подает коллекцию задач типа Callable на выполнение и возвращает список объектов Future<T>, представляющих результаты выполнения всех задач.
- shutdown(): Останавливает прием новых задач и позволяет выполнять оставшиеся задачи, после чего завершает работу пула потоков.

Преимущества использования ExecutorService включают управление жизненным циклом потоков, переиспользование потоков для выполнения нескольких задач, упрощение работы с многопоточностью

и улучшение производительности приложения за счет эффективного распределения задач между потоками.

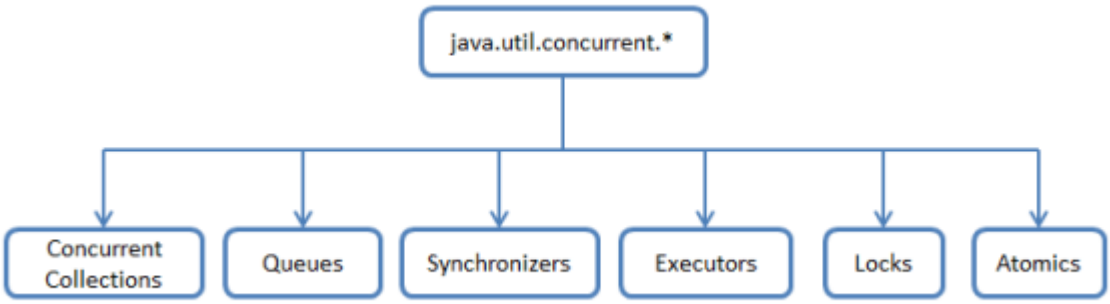
37. Интерфейс Future в Джава. Основное назначение использования его в программах

Еще раз напомним Интерфейс **Runnable** инкапсулирует задачу, выполняющуюся асинхронно. Вы можете воспринимать это как асинхронный метод без параметров и возвращаемого значения. **Callable** подобен **Runnable**, но с возвратом значения. Интерфейс **Callable** является параметризованным типом, с единственным общедоступным методом **call()**.
Интерфейс Future в Java представляет собой механизм для асинхронного выполнения операций и получения их результатов. Этот интерфейс дает возможность запустить задачу в одном потоке и получить результат ее выполнения в другом потоке. Основная цель использования интерфейса Future заключается в том, чтобы не блокировать главный поток выполнения программы (также называемый потоком-пользователем) в ожидании завершения длительной операции.

Интерфейс Future позволяет получить результат, возвращаемый задачей, когда он будет доступен. Для этого нужно выполнить метод get() для объекта Future. Если результат еще не готов, то выполнение главного потока будет приостановлено до готовности результата. Кроме того, Future может отменять задачи и проверять, завершены ли они. Он может быть использован для организации параллельного выполнения нескольких независимых задач.
Пример использования:

```java
import java.util.concurrent.*;

public class FutureExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Future<String> future = executor.submit(() -> {
            // Выполняем асинхронную задачу
            Thread.sleep(2000);
            return "Результат выполнения задачи";
        });

        System.out.println("Выполняется другая работа");

        try {
            // Ожидаем результат выполнения задачи
            String result = future.get();
            System.out.println("Результат: " + result);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }

        executor.shutdown();
    }
}
```

38. Коллекции java.util.concurrent. Состав коллекции. Основные методы



Concurrent Collections — набор коллекций, более эффективно работающих в многопоточной среде нежели стандартные универсальные коллекции из java.util пакета.
Queues — неблокирующие и блокирующие очереди с поддержкой многопоточности.
Synchronizers — вспомогательные утилиты для синхронизации потоков. Представляют собой мощное оружие в «параллельных» вычислениях.
Executors — содержит в себе отличные фрейморки для создания пулов потоков, планирования работы асинхронных задач с получением результатов.

Locks — представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми synchronized, wait, notify, notifyAll.
Atomics — классы с поддержкой атомарных операций над примитивами и ссылками.

39. Потокобезопасные коллекции пакета java.util.concurrent

TransferQueue – блокирующая очередь с подтверждением доставки
BlockingDeque – блокирующая двусторонняя очередь
ConcurrentLinkedQueue – неблокирующая очередь
ConcurrentLinkedDeque – неблокирующая двусторонняя очередь
DelayQueue – блокирующая очередь с задержкой
LinkedBlockingDeque – блокирующая двусторонняя очередь
LinkedTransferQueue – блокирующая очередь с подтверждением

Все эти классы являются потокобезопасными, т.к. спроектированы специально для использования разными потоками одновременно.

40. Реализация асинхронного выполнения в Джава

В Java существует несколько способов реализации асинхронного выполнения задач.

Использование интерфейса Runnable и класса Thread: Этот подход является наиболее простым и низкоуровневым. Вы создаете объекты класса, реализующего интерфейс Runnable, и передаете их в конструкторы объектов класса Thread. Затем вызываете метод start().

Использование класса ExecutorService и пула потоков: Вы создаете экземпляр ExecutorService и подаете задачи на выполнение с помощью метода submit(). Пул потоков автоматически управляет созданием и пере-/использованием потоков.

Использование интерфейса Future и Callable: Интерфейс Callable аналогичен интерфейсу Runnable, но предоставляет возможность возвращать результат выполнения задачи. Вы можете использовать класс FutureTask, реализующий интерфейс RunnableFuture. Future предоставляет возможность получить результат выполнения задачи, отменить задачу или проверить её состояние.

41. Паттерн “Одиночка” (Singleton) и его использование в Джава программах

Паттерн "Одиночка" (Singleton) - это порождающий паттерн проектирования, который гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру. В Java программировании паттерн "Одиночка" широко используется для создания классов, которые должны иметь только один экземпляр в рамках всего приложения, например, для доступа к базе данных или для управления конфигурационными параметрами.

```
public class Singleton {
   private static Singleton instance = new Singleton();

   private Singleton() {}

   public static Singleton getInstance() {
      return instance;
   }
}
```

Здесь мы создаем приватный конструктор, чтобы никто не мог создать экземпляр класса вне метода getInstance(). Метод getInstance() возвращает единственный экземпляр класса, созданный в статическом поле instance. Использование паттерна "Одиночка" может улучшить производительность и уменьшить потребление ресурсов, особенно в случаях, когда создание новых экземпляров класса затратно или нежелательно.

42. Использование статических методов для создания экземпляра объекта вместо конструкторов

Здесь подразумевается фабричный метод

Используется, когда нужно много разных способов создания объекта, и не нужно плодить лишние конструкторы.

Преимущества:
- Статическому методу можно дать осмысленное имя;
- Следуя из первого, возможно наличие методов с одинаковыми параметрами но разной логикой;
- Из статического метода можно вернуть null если что то пошло не так
Конструктор же всегда вернет что-то, или же выбросит исключение
- Можно вернуть тип, отличный от объявленного (например, вернуть класс-наследник)
- Статический метод фабрики может реализовывать логику кэширования объектов и возвращать уже созданные экземпляры, вместо создания новых каждый раз при вызове конструктора.

Статические методы используются вместо конструкторов, чтобы не рефакторить код конструктора, когда нужно значение по умолчанию и тд.

Пример:

```
public static Car createCar(String brand, String model) {
    // Логика проверки или кэширования объектов
    // ...

    return new Car(brand, model);
}
```

43. Паттерн “Строитель” и его использование большом количестве параметров конструктора

Паттерн Builder — это паттерн проектирования, который позволяет поэтапно создавать сложные объекты с помощью четко определенной последовательности действий. Строительство контролируется объектом-распорядителем (director), которому нужно знать только тип создаваемого объекта.

Итак, паттерн проектирования Builder можно разбить на следующие важные компоненты:

- Product (продукт) - Класс, который определяет сложный объект, который мы пытаемся шаг за шагом сконструировать, используя простые объекты.
- Builder (строитель) - абстрактный класс/интерфейс, который определяет все этапы, необходимые для производства сложного объекта-продукта. Как правило, здесь объявляются (абстрактно) все этапы (buildPart), а их реализация относится к классам конкретных строителей (ConcreteBuilder).
- ConcreteBuilder (конкретный строитель) - класс-строитель, который предоставляет фактический код для создания объекта-продукта. У нас может быть несколько разных ConcreteBuilder-

классов, каждый из которых реализует различную разновидность или способ создания объекта-продукта.
- Director (распорядитель) - супервизионный класс, под конролем котрого строитель выполняет скоординированные этапы для создания объекта-продукта. Распорядитель обычно получает на вход строителя с этапами на выполнение в четком порядке для построения объекта-продукта.

Паттерн проектирования Builder решает такие проблемы, как:

- Как класс (тот же самый процесс строительства) может создавать различные представления сложного объекта?
- Как можно упростить класс, занимающийся созданием сложного объекта?

```
// класс - продукт
public class Car {

    private String chassis;
    private String body;
    private String paint;
    private String interior;

    public Car() {
        super();
    }

    public Car(String chassis, String body, String paint, String interior) {
        this();
        this.chassis = chassis;
        this.body = body;
        this.paint = paint;
        this.interior = interior;
    }

// геттеры и сеттеры

public interface CarBuilder {
//шаги для создания автомобился
    // Этап 1
    public CarBuilder fixChassis();

    // Этап 2
    public CarBuilder fixBody();

    // Этап 3
    public CarBuilder paint();

    // Этап 4

    public CarBuilder fixInterior();

    // Выпуск автомобиля

    public Car build();
```

```
}


//тут уже конкретный продукт их можно сделать дофига
public class ClassicCarBuilder implements CarBuilder {

    private String chassis;
    private String body;
    private String paint;
    private String interior;

    public ClassicCarBuilder() {
        super();
    }

    @Override
    public CarBuilder fixChassis() {
        System.out.println("Assembling chassis of the classical model");
        this.chassis = "Classic Chassis";
        return this;
    }

// и переопределяем методы

public class AutomotiveEngineer {

    private CarBuilder builder;

    public AutomotiveEngineer(CarBuilder builder) {
        super();
        this.builder = builder;
        if (this.builder == null) {
            throw new IllegalArgumentException("Automotive Engineer can't work without Car Builder!");
        }
    }

    public Car manufactureCar() {
        return builder.fixChassis().fixBody().paint().fixInterior().build();
    }

}

public class Main {

    public static void main(String[] args) {
        CarBuilder builder = new SportsCarBuilder();
        AutomotiveEngineer engineer = new AutomotiveEngineer(builder);
        Car car = engineer.manufactureCar();
```

44. Понятие dependency injection (внедрение зависимости). Преимущества использования внедрения зависимостей при написании программ на языке Джава. Преимущества этого подхода перед паттерном Одиночка

Внедрение зависимостей — это стиль настройки объекта, при котором поля объекта задаются внешней сущностью. Другими словами, объекты настраиваются внешними объектами. DI — это альтернатива самонастройке объектов

Главный плюс в том, что мы можем рассматривать приложение, как набор сервисов или модулей. Сам по себе код получается чуть компактней и не нужно завязываться на имя класса.

При развитии программ очень часто оказывается, что некоторая сущность, которая ранее присутствовала в 1 экземпляре, требуется в нескольких экземплярах.

Поэтому использованию Singleton предпочитайте dependency injection

45. Преимущество использования try-c-ресурсами по сравнению с использованием try-finally. Интерфейс интерфейса AutoCloseable

При завершении работы с потоком его надо закрыть с помощью метода close(), который определен в интерфейсе Closeable.

При закрытии потока освобождаются все выделенные для него ресурсы, например, файл. В случае, если поток окажется не закрыт, может происходить утечка памяти.

Есть два способа закрытия файла. Первый традиционный заключается в использовании блока try..catch..finally.

Поскольку при открытии или считывании файла может произойти ошибка ввода-вывода, то код считывания помещается в блок try. И чтобы быть уверенным, что поток в любом случае закроется, даже если при работе с ним возникнет ошибка, вызов метода close() помещается в блок finally. И, так как метод close() также в случае ошибки может генерировать исключение IOException, то его вызов также помещается во вложенный блок try..catch

Начиная с Java 7 можно использовать еще один способ, который автоматически вызывает метод close. Этот способ заключается в использовании конструкции try-with-resources (try-c-ресурсами). Данная конструкция работает с объектами, которые реализуют интерфейс AutoCloseable. Так как все классы потоков реализуют интерфейс Closeable, который в свою очередь наследуется от AutoCloseable, то их также можно использовать в данной конструкции

Синтаксис конструкции следующий: try(название_класса имя_переменной = конструктор_класса). Данная конструкция также не исключает использования блоков catch.

После окончания работы в блоке try у ресурса (в данном случае у объекта FileInputStream) автоматически вызывается метод close().

Если нам надо использовать несколько потоков, которые после выполнения надо закрыть, то мы можем указать объекты потоков через точку с запятой:

try(FileInputStream fin=new FileInputStream("C://SomeDir//Hello.txt");

FileOutputStream fos = new FileOutputStream("C://SomeDir//Hello2.txt"))

{

//.....

}

46. Паттерн Декоратор (Decorator) Преимущества его использовании (например композиция по сравнению с наследованием)

Паттерн декоратор (Decorator) – один из наиболее известных и распространенных паттернов проектирования, используемых в Java. Он позволяет динамически добавлять новую функциональность к объектам, не изменяя их исходный код. Это очень удобно, когда нужно добавить дополнительные возможности к уже существующему объекту.
Пример использования паттерна декоратор в Java: Предположим, у нас есть интерфейс `Pizza`, который имеет методы `getDescription()` и `getCost()`. Этот интерфейс представляет собой базовый класс для разных типов пиццы.

```
public interface Pizza {
   String getDescription();
   double getCost();
}
```

Теперь мы хотим добавить новые возможности к объекту `Pizza`, не меняя его исходный код. Для этого мы создадим класс-декоратор `PizzaDecorator`, который также реализует интерфейс `Pizza`.

```
public abstract class PizzaDecorator implements Pizza {
    protected Pizza pizza;

    public PizzaDecorator(Pizza pizza) {
        this.pizza = pizza;
    }


}
```

Класс `PizzaDecorator` имеет ссылку на объект `pizza`, который он декорирует. Он также реализует методы интерфейса `Pizza`.

Теперь мы можем создать конкретные декораторы, которые будут добавлять новые возможности к объекту `Pizza`. Например, мы можем создать декоратор `CheeseDecorator`, который будет добавлять сыр к пицце.

```
public class CheeseDecorator extends PizzaDecorator {
public CheeseDecorator(Pizza pizza) {
        super(pizza);
    }
    public String getDescription() {
        return pizza.getDescription() + ", Cheese";
```

```
    }
    public double getCost() {
        return pizza.getCost() + 2.00;
    }
}
```

Декоратор `CheeseDecorator` добавляет сыр к пицце. Он также имеет ссылку на объект `pizza`, который он декорирует, и реализует методы интерфейса `Pizza`.

Теперь мы можем создать объекты `Pizza` и добавлять к ним декораторы. Например, мы можем создать объект `PepperoniPizza` и добавить к нему декоратор `CheeseDecorator`.

```
Pizza pepperoniPizza = new PepperoniPizza();

pepperoniPizza = new CheeseDecorator(pepperoniPizza);

System.out.println(pepperoniPizza.getDescription() + ": $" +
pepperoniPizza.getCost());
```

47. Преимущества использования списков перед массивами

Массив выигрывает в скорости, однако его предпочтительнее использовать в том случае, когда заранее известно число элементов.

Списки более медленные, однако работать с ними удобнее и проще, дженерики можно применять. Сами по себе являются динамическими. Кроме того возможно изменение подтипа за счет задания через интерфейс, что обеспечивает гибкость (`List<String> list = new ArrayList<String>();`)

Шаблоны проектирования

Списки более универсальны, чем массивы:

- в списки можно добавлять/удалять значения
- массив можно преобразовать в список через Arrays.asList без копирования данных; при вызове list.toArray данные массива копируются
- пустой список Collections.emptyList() не выделяет память, в отличие от “new Type[0]”
- если нужно запретить изменение списка, можно использовать Collections.unmodifiableList; массив всегда изменяемый
- можно создать “new ArrayList<T>()”, но нельзя “new T[n]”, где T – generic тип

online.mirea.ru

48. Основные системы сборки Gradle и Maven. Использование Gradle и основная терминология. Управление зависимостями в Gradle

Основные используемые системы сборки – Maven и Gradle.
Gradle создан в 2007 году, текущая версия – 7.0.
Gradle использует для конфигурации проекта полноценный язык программирования; изначально Groovy, с версии 3.0 также поддерживается Kotlin.

Конфигурация проектов

build		Набор библиотек и приложений. Большие проекты, как правило, состоят из нескольких подпроектов
project	(под)проект	Ваша библиотека или приложение. Кроме исходного кода, для проекта должно быть описание (build.gradle)
task	задача	Задача, которую можно выполнить над проектом. Например, **compileJava** – скомпилировать исходный код на Java
plugin	плагин	Модуль Gradle, предназначенный для выполнения класса задач. Например, **application** – плагин для сборки приложений Java. Плагин добавляет описания задач, которые он умеет выполнять.

• параметры group и version описывают группу и версию, под которой проект будет публиковаться Если вы планируете использование вашего проекта другими разработчиками,

эти параметры нужно указывать
• параметр description – человекочитаемое описание проекта

• sourceCompatibility указывает, какую версию Java использует проек

• options.encoding указывает кодировку исходного кода
Управление зависимостями
Кроме запуска компиляции, Gradle умеет автоматически скачивать нужные для сборки приложения библиотеки. Для этого нужно указать:

• репозиторий, из которого качать библиотеки
(обычно это mavenCentral: https://repo1.maven.org/maven2) • “адрес” библиотеки в виде тройки

• group – группа, в которую входит библиотека • name – имя библиотеки
• version – версия библиотеки

Проекты Maven в первую очередь определяются файлами объектной модели проекта (POM), написанными в XML. Эти файлы POM.xml содержат зависимости проекта, плагины, свойства и данные конфигурации. Maven использует декларативный подход и имеет предопределенный жизненный цикл.

49. Анатомия jar. Сканирование пакетов

50. Реализация REST API с помощью Spring Framework

REST – стиль взаимодействия клиента с сервером. Обычно он подразумевает запросы и ответы в формате JSON, где адрес запроса содержит информацию о том, что хочет сделать клиент.

Итого задача серверной части веб-приложения при обработке запроса:
1. Понять, какой запрос пришел (например, запрос /api/books для получения списка всех книг)
2. Обратиться к базе данных (или другому источнику данных) для получения запрошенных данных в виде обычных Java-объектов (в нашем примере списка книг)
3. Сериализовать Java-объекты в формат JSON и отправить в качестве ответа на запрос

Архитектуры реализуем с помощью MVC:
1. Слой представления отвечает за взаимодействие с клиентом: сериализацию/десериализацию JSON, маршрутизацию (routing) – определение, какое действие было запрошено
2. Слой бизнес-логики отвечает за логику работы приложения (перенаправление запроса к слою хранения данных)
3. Слой хранения данных отвечает за взаимодействие с базой данных: получение данных из БД и сохранение данных в БД

51. Понятие Инверсии управления

В первую очередь это принцип разработки программного обеспечения, который используется во всех фреймворках, который подразумевает предоставление callback-а в качестве реакции на какие либо события извне, вместо того, чтобы реализовывать логику обработки события на месте.

```
@Service
public class MyDatabaseService implements DatabaseService {
    private final DatabaseRepository databaseRepository;

    // Конструктор, который принимает необходимые параметры
    @Autowired
    public MyDatabaseService(DatabaseRepository databaseRepository) {
        this.databaseRepository = databaseRepository;
    }

    // Метод, который использует базу данных для сохранения данных
    @Override
    public void saveData(Object data) {
        databaseRepository.save(data);
    }
}
```

В этом примере Spring автоматически создает экземпляр MyDatabaseService и внедряет зависимость от DatabaseRepository. Это позволяет программисту сосредоточиться на написании бизнес-логики приложения, в то время как контейнер управляет созданием и внедрением объектов, что существенно упрощает и ускоряет процесс разработки.

52. Spring Boot и его использование

Spring Boot – надстройка над spring которая упрощает создание и развертывание приложений. Он предоставляет быстрый и простой способ настройки, конфигурации и запуска приложений, а также включает в себя множество функций, таких как встроенный сервер приложений и обработка ошибок по умолчанию.

Spring boot Предоставляет возможность создавать автономные приложения Spring, которые можно запускать сразу же без заметок, конфигурации XML и написания больших объемов дополнительного кода.

Используйте Spring Boot, когда захотите:

- Простота использования
- Подход на основе рекомендаций*
- Для быстрого создания качественных приложений и сокращения времени разработки.

- Отказ от необходимости писать шаблонный код и настраивать XML.
- Разработка API REST.

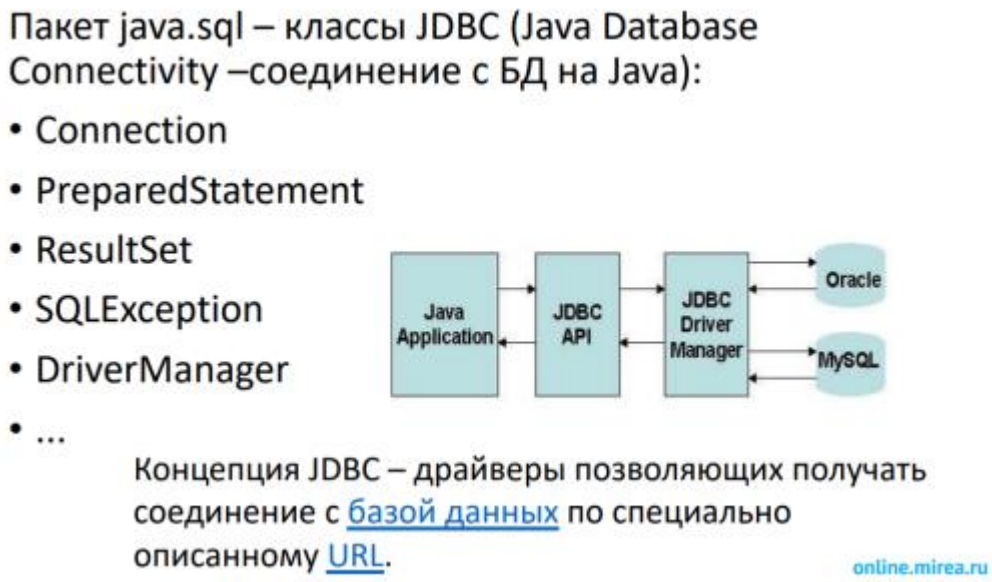
Spring Boot удобен для написания микросервисов.

Spring Boot Берет на себя все рутинные действия по созданию Spring-приложений и ускоряет вашу работу настолько, насколько это возможно.

53. Работа с базами данных в джава приложениях. Встроенные СУБД для Джава приложений.

54. Реляционные СУБД для работы с джава приложениями. Пакет java.sql и его классы

Реляционная база данных — это набор таблиц, между которыми установлены определенные взаимосвязи. Для обслуживания реляционной базы данных и создания запросов к ней система управления базой данных использует язык структурированных запросов SQL — обычное пользовательское приложение, предоставляющее простой интерфейс программирования для взаимодействия с базой данных.



55. Роль интерфейса JDBC для работы с джава приложениями

JDBC создает прослойку между Java приложением и драйвером базы данных. JDBC позволяет устанавливать соединение с источником данных, отправлять запросы и операторы обновления, а также обрабатывать результаты. Проще говоря, JDBC позволяет делать в Java-приложении следующие вещи:

устанавливать соединение с источником данных и отправлять запросы и операторы обновления в источник данных.

56. Основные компоненты JDBC API

DriverManager:

Это класс, использующийся для управления списком Driver (database drivers).

***Driver*:**

Это интерфейс, использующийся для соединения коммуникации с базой данных, управления коммуникации с базой данных.

Connection :

Интерфейс со всеми методами связи с базой данных. Он описывает коммуникационный контекст. Вся связь с базой данных осуществляется только через объект соединения (connection).

***Statement* :**

Это интерфейс, включающий команду SQL отправленный в базу данных для анализа, обобщения, планирования и выполнения.

ResultSet :

ResultSet представляет набор записей, извлеченных из-за выполнения запроса.

57. JDBC URL и его использование

jdbc:<субпротокол>:<имя, связанное с СУБД или протоколом>

PostgreSQL: **jdbc:postgresql://localhost:5432/postgres**

Oracle: **jdbc:oracle:thin:@//localhost:1521/orcl**

H2: **jdbc:h2:~/example_db**

H2 – встроенная СУБД:

```
dependencies {
   runtimeOnly("com.h2database:h2:1.4.200")
}
```

```
Connection c = DriverManager.getConnection("jdbc:h2:~/example_db")
```

58. Работа JDBC драйвера

59. В чем заключается роль DI в Spring. Использования ServiceLoader

ServiceLoader - встроенный DI-фреймворк

ServiceLoader — это механизм, который позволяет внешнему коду «обращаться» к внутренним реализациям. Модули Java позволяют регистрировать сервисы для внутренних реализаций, сохраняя при этом границу.

60. Интерфейс Connection. Пулы соединений

Интерфейс Connection описывает активное соединение с БД.

```
public class BookRepository {
        public List getAllBooks() {
                try (Connection connection = ???) {
                }
        }
}
```

Плохие примеры работы с соединением:

1. Открывать соединение каждый раз в методах BookRepository: Connection connection = DriverManager.getConnection("...") Соединение открывается медленно и приложение не будет справляться с большой нагрузкой
2. Держать Connection в поле BookRepository. Как правило, только один поток может одновременно работать с одним соединением. Хотя соединения обычно потокобезопасны, но это достигается с

помощью синхронизации. Поэтому при большом количестве параллельных запросов к BookRepository только один из них будет работать, остальные будут ждать.

Поэтому используются пулы соединений (по аналогии с пулами потоков). Например, пул с максимальным количеством соединений = 20 позволяет параллельно работать с БД 20 потокам. При этом соединения не закрываются, а по возможности переиспользуются.

Есть несколько реализаций пулов соединений:
• HikariCP
• Apache Commons DBCP
• C3PO

61. Принципы SOLID и их использование на Джава

SOLID — это аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании. Эти принципы позволяют строить на базе ООП масштабируемые и сопровождаемые программные продукты с понятной бизнес-логикой.

S (принцип единой ответственности) - (single responsibility principle) Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.

O (принцип открытости/закрытости) - (open-closed principle) «программные сущности … должны быть открыты для расширения, но закрыты для модификации».

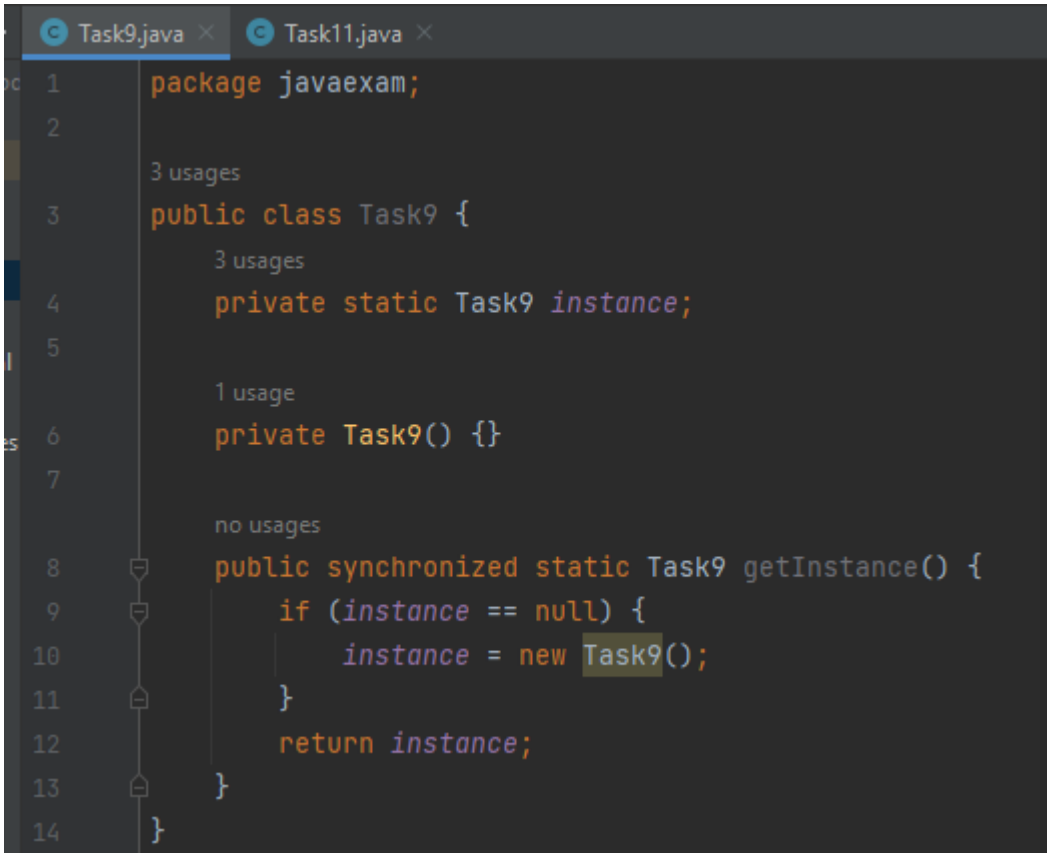
L (принцип подстановки Лисков) - (Liskov substitution principle) «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом».

I (принцип разделение интерфейсов) - (interface segregation principle) «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения».

D (принцип инверсии зависимостей) - (dependency inversion principle) «Зависимость на Абстракциях. Нет зависимости на что-то конкретное»

ПРАКТИКА

Задача 9



Задача 10

```
class Consumer extends Thread{
    BlockingQueue bq;
    public Consumer(BlockingQueue bq) {
        this.bq = bq;
    }
    public void run() {
        for (int i = 0; i < 30; i++) {
            try {
                System.out.println("Consumed: " + bq.take());
            } catch (InterruptedException e) {
                System.err.println(e.getMessage());
            }
        }
        ;
    }
    }

class Producer extends Thread {
    public Producer(BlockingQueue bq) {
        this.bq = bq;
    }

    BlockingQueue bq;
    public void run(){
        java.util.function.Consumer<Integer> write = x -> {
            try {
                System.out.println("Writed: " + x);
```

```
                bq.put(x);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        };
        Supplier<Integer> rand = () -> new Random().nextInt(15);
        Stream<Integer> st = Stream.generate(rand).limit(30);
        st.forEach(write);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        ArrayBlockingQueue<Integer> bq = new
ArrayBlockingQueue<Integer>(5);
        new Producer(bq).start();
        new Consumer(bq).start();
    }
}
```

задача 11

```
package javaexam;

no usages
public class Task11 {
    no usages
    public static void main(String[] args) {
        var count = 0;
        for (var str: args) {
            count += str.length();
        }
        System.out.println(count);
    }
}
```

задача 12

```java
package javaexam;

1 usage
interface Printable {
    no usages
    void print();
}

no usages
public class Task12 {
    no usages
    public static void main(String[] args) {
        Printable p = () -> {
            System.out.println("printed");
        };
    }
}
```

задача 13

```java
package javaexam;

import java.util.function.Predicate;

no usages
public class Task13 {
    no usages
    public static void main(String[] args) {
        Predicate<String> stringPredicate = (s) -> s != null;
    }
}
```

задача 14

```java
package javaexam;

import java.util.function.Predicate;

no usages
public class Task14 {
    no usages
    public static void main(String[] args) {
        Predicate<String> stringPredicate = (s) -> !s.isEmpty();
    }
}
```

задача 15

```java
package javaexam;

import java.util.function.Predicate;

no usages
public class Task15 {
    no usages
    public static void main(String[] args) {
        Predicate<String> stringPredicate = ((Predicate<String>)(s) -> !s.isEmpty()).and((s) -> s != null);
    }
}
```

```java
import java.util.Scanner;
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        String str = new Scanner(System.in).nextLine();
        Predicate<String> notEmpty = s -> s!="";
        Predicate<String> notNull = s -> s!=null;
        System.out.println(notEmpty.and(notNull).test(str));
    }
}
```

задача 16

```java
package javaexam;

import java.util.function.Predicate;

no usages
public class Task16 {
    no usages
    public static void main(String[] args) {
        Predicate<String> stringPredicate = (s) -> (s.startsWith("J") || s.startsWith("N")) && s.endsWith("A");
    }
}
```

задача 17

```java
package javaexam;

import java.util.function.Consumer;

1 usage
class HeavyBox {
    3 usages
    private int weight;

    no usages
    public HeavyBox(int weight) {
        this.weight = weight;
    }

    1 usage
    public int getWeight() {
        return weight;
    }

    no usages
    public void setWeight(int weight) {
        this.weight = weight;
    }
}

no usages
public class Task17 {
    no usages
    public static void main(String[] args) {
        Consumer<HeavyBox> consumer = (box) -> {
            System.out.printf("Отправляем ящик с весом %d\n", box.getWeight());
        };
    }
}
```

Задача 18

```java
public class Main {
    public static void main(String[] args) {
        Function<Integer, String> sign = x -> {
            if(x==0){
                return "Ноль";
            }
            else if(x>0){
```

```
                return "Положительное число";
            }
            return "Отрицательное число";
        };
        Scanner sc = new Scanner(System.in);
        System.out.println(sign.apply(sc.nextInt()));
    }
}
```

Задача 19

```
public class Main {
    public static void main(String[] args) {
        Supplier<Integer> randomNumberReturn = () -> (int)(Math.random() *
11);
        System.out.println(randomNumberReturn.get());
        System.out.println(randomNumberReturn.get());
        System.out.println(randomNumberReturn.get());
        System.out.println(randomNumberReturn.get());
        System.out.println(randomNumberReturn.get());
        System.out.println(randomNumberReturn.get());
    }
}
```

Задача 20

```
@FunctionalInterface
public interface Printable {
    void print();
}
```

```
public class Main {
    private static void printHello() {
        System.out.println("Hello");
    }
    public static void main(String[] args) {
        Printable printHello1 = () -> {
            System.out.println("Hello");
        };
        Printable printHello2 = Main::printHello;
        printHello1.print();
        printHello2.print();
    }
}
```

задача 23 - решение чат гпт

```
class MyThread extends Thread {
    private StringBuilder sb;

    public MyThread(StringBuilder sb) {
        this.sb = sb;
    }
```

```
    @Override
    public void run() {
        synchronized (sb) {
            for (int i = 0; i < 100; i++) {
                System.out.print(sb);
            }
            sb.setCharAt(0, (char) (sb.charAt(0) + 1));
        }
    }
}

class Main {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("a");
        MyThread t1 = new MyThread(sb);
        MyThread t2 = new MyThread(sb);
        MyThread t3 = new MyThread(sb);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

задача 25 полон дом говна

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Philosopher implements Runnable {
    private int id;
    private Lock leftFork;
    private Lock rightFork;

    public Philosopher(int id, Lock leftFork, Lock rightFork) {
        this.id = id;
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }

 private void doAction(String action) throws InterruptedException {
        System.out.println("Philosopher " + id + " " + action);
        Thread.sleep((int) (Math.random() * 100));
    }

    @Override
    public void run() {
        try {
            while (true) {
                // Размышлять
                doAction("is thinking");
                // Взять левую вилку
```

```
                leftFork.lock();
                try {
                    // Взять правую вилку
                    rightFork.lock();
                    try {
                        // Есть
                        doAction("is eating");
                    } finally {
                        // Положить правую вилку на стол
                        rightFork.unlock();
                    }
                } finally {
                    // Положить левую вилку на стол
                    leftFork.unlock();
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

public class DiningPhilosophers {
    private static final int NUM_PHILOSOPHERS = 5;

    public static void main(String[] args) {
        Philosopher[] philosophers = new Philosopher[NUM_PHILOSOPHERS];
        Lock[] forks = new ReentrantLock[NUM_PHILOSOPHERS];

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            forks[i] = new ReentrantLock();
        }

        for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
            Lock leftFork = forks[i];
            Lock rightFork = forks[(i + 1) % NUM_PHILOSOPHERS];

            philosophers[i] = new Philosopher(i, leftFork, rightFork);
            Thread thread = new Thread(philosophers[i]);
            thread.start();
        }
    }
}
```

задача 24 - решение чат гпт

```
class MyThread extends Thread {
   private StringBuilder sb;

   public MyThread(StringBuilder sb) {
       this.sb = sb;
   }
```

```
    @Override
    public void run() {
        synchronized (sb) {
            for (int i = 0; i < 100; i++) {
                System.out.print(sb);
            }
            sb.setCharAt(0, (char) (sb.charAt(0) + 1));
        }
    }
}

class Main {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("a");
        MyThread t1 = new MyThread(sb);
        MyThread t2 = new MyThread(sb);
        MyThread t3 = new MyThread(sb);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

задача 26 - решение чат гпт

```
package q123;

import java.util.ArrayList;
import java.util.List;

public class t26 {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for (int i = 1; i <= 10; i++) {
            list.add(i);
        }

        List<Integer> result = list.stream()
                .filter(s -> list.indexOf(s) % 5 != 4)
                .toList();

        System.out.println(result);
    }
}
```

я у столааа
браааатан давай посидим давай подымим давай поговорим