Альфа-Банк - Матрица компетенций Java

**СОДЕРЖАНИЕ**

[ОБЩЕЕ 2](#_Toc182061576)

[INTERN 2](#_Toc182061577)

[1.1 Что такое алгоритмы, для чего они используются, примеры 2](#_Toc182061578)

[1.2 Что такое классы и интерфейсы, чем отличаются друг от друга 3](#_Toc182061579)

[1.3 Основные концепции программирования (условия, циклы, функции, рекурсия, типы данных) 4](#_Toc182061580)

[1.4 Что такое тестирование ПО, отличие белого и черного ящиков 5](#_Toc182061581)

[1.5 Принципы работы клиент-серверных приложений 6](#_Toc182061582)

[JUNIOR 8](#_Toc182061583)

[1.6 Нотация Big O, оценка сложности алгоритма 8](#_Toc182061584)

[1.7 Структуры данных и основные алгоритмы для работы с ними (стеки, очереди, хеш таблицы, деревья, сортировки, обходы) 10](#_Toc182061585)

[1.8 Принципы ООП 13](#_Toc182061586)

[1.9 Пирамида тестирования, unit-тесты, моки 15](#_Toc182061587)

[1.10 Основные паттерны ООП 16](#_Toc182061588)

# ОБЩЕЕ

## INTERN

### 1.1 Что такое алгоритмы, для чего они используются, примеры

*Алгоритмы* — это последовательные шаги или инструкции, предназначенные для решения задач или выполнения определенных действий. Алгоритм можно рассматривать как чётко структурированный план, который нужно выполнить, чтобы достичь цели.

*Для чего используются алгоритмы:*

1. Программирование: алгоритмы используются для обработки данных, выполнения вычислений, работы с массивами и базами данных
2. Математика: алгоритмы помогают решать математические задачи, например, нахождение наибольшего общего делителя (НОД) или сортировка чисел
3. Кибербезопасность: алгоритмы шифрования применяются для защиты данных
4. Машинное обучение: алгоритмы используются для анализа данных и создания моделей, которые могут прогнозировать или классифицировать информацию

*Примеры алгоритмов:*

1. *Алгоритм сортировки пузырьком (Bubble Sort)*. Это простейший способ упорядочить элементы массива. Он последовательно сравнивает соседние элементы и меняет их местами, если они не в нужном порядке.

fun bubbleSort(arr: IntArray) {

for (i in arr.indices) {

for (j in 0 until arr.size - i - 1) {

if (arr[j] > arr[j + 1]) {

val temp = arr[j]

arr[j] = arr[j + 1]

arr[j + 1] = temp

}

}

}

}

1. *Алгоритм бинарного поиска (Binary Search)*. Это метод для поиска элемента в отсортированном массиве, который уменьшает диапазон поиска в два раза на каждом шаге.

fun binarySearch(arr: IntArray, key: Int): Int {

var low = 0

var high = arr.size - 1

while (low <= high) {

val mid = (low + high) / 2

when {

arr[mid] == key -> return mid

arr[mid] < key -> low = mid + 1

else -> high = mid - 1

}

}

return -1 // Элемент не найден

}

### 1.2 Что такое классы и интерфейсы, чем отличаются друг от друга

*Классы и интерфейсы* — это два ключевых понятия объектно-ориентированного программирования (ООП), которые используются для организации и структурирования кода.

*Класс* — это шаблон или «чертеж» для создания объектов. Он объединяет данные (поля) и поведение (методы) в одну структуру. В классе можно реализовать как данные, так и их поведение.

*Интерфейс* — это контракт, который определяет набор методов и свойств, но не предоставляет их реализацию. Реализация этих методов должна быть предоставлена классами, которые реализуют интерфейс. Интерфейсы нужны для задания поведения, которое должно быть у всех объектов, реализующих этот интерфейс, но без указания, как именно это поведение должно быть реализовано.

*Отличия между классами и интерфейсами:*

1. Содержимое:
   1. Класс может содержать реализацию методов и хранить данные (в виде полей или свойств)
   2. Интерфейс не содержит реализаций методов (в некоторых языках, таких как Kotlin и Java 8, интерфейсы могут иметь методы с реализацией по умолчанию), он только описывает поведение, которое должно быть реализовано в классах
2. Наследование:
   1. Класс может наследоваться от одного класса (так как в большинстве языков программирования используется одиночное наследование)
   2. Интерфейс может быть реализован несколькими классами, и класс может реализовать несколько интерфейсов
3. Использование:
   1. Класс используется для создания объектов и описания их структуры
   2. Интерфейс используется для задания контракта поведения, который будет реализован разными классами
4. Реализация методов:
   1. В классе методы могут быть сразу реализованы
   2. В интерфейсе методы не реализуются, но классы, которые реализуют интерфейс, должны предоставить реализацию этих методов.

### 1.3 Основные концепции программирования (условия, циклы, функции, рекурсия, типы данных)

1. *Условия (ветвления)*. Условия используются для выполнения определенных блоков кода в зависимости от выполнения логического выражения. Операторы: if, else, else if, switch
2. *Циклы.* Циклы позволяют повторять выполнение блока кода несколько раз, пока выполняется определенное условие. Типы циклов: for, while, do-while
3. *Функции (методы)*. Функции — это блоки кода, которые выполняют определенные действия и могут возвращать результат. Они помогают структурировать код и избегать его повторения
4. *Рекурсия* — это техника, при которой функция вызывает саму себя. Это полезно для решения задач, которые могут быть разбиты на подзадачи того же типа, например вычисление факториала
5. *Типы данных*. Типы данных определяют, какие значения могут храниться в переменных и какие операции можно выполнять с этими значениями. В Java типы данных делятся на:
   1. Примитивные типы: int, double, char, boolean и др.
   2. Ссылочные типы: объекты классов, массивы и другие структуры данных.

### 1.4 Что такое тестирование ПО, отличие белого и черного ящиков

*Тестирование программного обеспечения (ПО)* — это процесс проверки работы программы для выявления дефектов и ошибок, а также для того, чтобы убедиться, что она соответствует требованиям и ожиданиям пользователей.

Тестирование ПО помогает:

* Обеспечить качество продукта;
* Найти и исправить ошибки до выпуска продукта в эксплуатацию;
* Проверить соответствие продукта требованиям.

*Основные виды тестирования*:

1. Функциональное тестирование — проверка того, что система работает в соответствии с заданными функциями;
2. Нефункциональное тестирование — проверка аспектов, таких как производительность, безопасность, удобство использования и масштабируемость.

Тестирование *черного ящика*:

* Тестер не знает внутренней структуры или кода системы
* Проверяется только функциональность системы на основе входных и выходных данных
* Подход: "ввод данных — получение результата"
* Пример: тестирование пользовательского интерфейса, проверка реакций системы на определённые входные данные
* Используется для функционального тестирования

Тестирование *белого ящика*:

* Тестер знает внутреннюю структуру системы и может видеть код
* Тестирование проводится с учётом логики работы программы, ветвлений, циклов и т.д
* Цель — проверка работы всех внутренних частей программы, таких как функции, циклы и структуры данных
* Пример: модульное тестирование кода на уровне методов или классов
* Используется для проверки логики и работы отдельных компонентов.

Основное различие между этими подходами в том, что при тестировании черного ящика проверяется внешнее поведение системы, а при тестировании белого ящика — её внутренняя реализация.

### 1.5 Принципы работы клиент-серверных приложений

*Клиент-серверная архитектура* — это модель взаимодействия между двумя сущностями: клиентом и сервером, которая лежит в основе работы большинства современных приложений. В этой архитектуре одна сторона (клиент) делает запросы, а другая (сервер) обрабатывает их и отправляет ответы.

*Основные принципы работы клиент-серверных приложений:*

1. Разделение ответственности:
   1. Клиент — это приложение или устройство, которое инициирует запросы и использует данные или услуги, предоставляемые сервером. Клиент может быть браузером, мобильным приложением, терминалом и т.д.
   2. Сервер — это программа или устройство, которое обрабатывает запросы клиентов и предоставляет ресурсы, данные или вычислительные мощности. Сервер может работать с базой данных, выполнять бизнес-логику или управлять сессиями
2. Обмен запросами и ответами:
   1. Клиент отправляет запрос на сервер (например, HTTP-запрос в веб-приложениях), обычно с некоторыми параметрами
   2. Сервер принимает запрос, обрабатывает его (например, обращается к базе данных, выполняет вычисления) и возвращает ответ клиенту (например, HTML-страницу, JSON-данные)
3. Протоколы взаимодействия:
   1. Для общения между клиентом и сервером используется набор правил — протокол. Одним из самых популярных протоколов является HTTP/HTTPS для веб-приложений, но также могут использоваться FTP, WebSocket, SMTP и другие
   2. Протокол определяет формат запросов, ответов и порядок обмена сообщениями между сторонами
4. Асинхронность:
   1. Клиент не всегда получает ответ сразу — сервер может обрабатывать запросы какое-то время, в результате чего работа клиента не должна блокироваться. Это достигается с помощью асинхронного взаимодействия
   2. Пример асинхронности — веб-браузеры, которые могут продолжать выполнять другие задачи, пока сервер не вернет ответ (например, загрузку данных через Ajax-запросы)
5. Состояние сессии:
   1. Взаимодействие может быть сессионным (например, для отслеживания действий пользователя) или бессессионным. HTTP по своей природе является бессессионным протоколом, однако сессии могут реализовываться с помощью куки, токенов или других механизмов для идентификации пользователя
6. Масштабируемость:
   1. Серверы могут обрабатывать множество запросов одновременно, что требует поддержания балансировки нагрузки и масштабируемости для обработки большого числа клиентов. Это может включать использование кластеров серверов или облачных решений
7. Безопасность:
   1. Для защиты данных и предотвращения атак важно использовать безопасные соединения и механизмы авторизации/аутентификации. HTTPS обеспечивает шифрование данных при передаче, а различные схемы аутентификации (OAuth, JWT) позволяют безопасно управлять доступом

*Пример клиент-серверного взаимодействия*:

1. Пользователь через браузер (клиент) вводит URL-адрес
2. Браузер отправляет HTTP-запрос к серверу
3. Сервер обрабатывает запрос (например, обращается к базе данных)
4. Сервер отправляет HTML-страницу в виде ответа
5. Браузер отображает полученные данные пользователю

## JUNIOR

### 1.6 Нотация Big O, оценка сложности алгоритма

Нотация Big O используется для описания оценки времени выполнения или использования памяти алгоритма в зависимости от размера входных данных. Она помогает понять, как производительность алгоритма масштабируется при увеличении объёма данных. Важным аспектом Big O является то, что она показывает асимптотическое поведение алгоритма, то есть как его время работы изменяется при увеличении размера входных данных до бесконечности.

Основные классы сложности Big O:

1. ***O(1)*** — Константная сложность. Алгоритм выполняется за фиксированное время, которое не зависит от размера входных данных. Пример: доступ к элементу массива по индексу
2. ***O(log n)*** — Логарифмическая сложность. Время выполнения алгоритма увеличивается логарифмически с ростом данных. Такие алгоритмы часто встречаются в бинарном поиске, где на каждом шаге количество рассматриваемых элементов уменьшается вдвое.
3. ***O(n)*** — Линейная сложность. Время выполнения увеличивается линейно по отношению к количеству входных данных. Пример: простой цикл по всем элементам массива.
4. ***O(n \* log n)*** — Линейно-логарифмическая сложность. Часто встречается в алгоритмах сортировки (например, быстрая сортировка, сортировка слиянием). Такие алгоритмы комбинируют линейную проверку и логарифмическое разбиение данных.
5. ***O(n²)*** — Квадратичная сложность. Время выполнения увеличивается пропорционально квадрату количества входных данных. Пример: вложенные циклы для сравнения всех элементов друг с другом, как в сортировке пузырьком.
6. ***O(2n)*** — Экспоненциальная сложность. Время выполнения удваивается при каждом добавлении единицы к количеству входных данных. Такие алгоритмы встречаются в задачах перебора всех возможных комбинаций (например, решение задачи коммивояжера методом полного перебора).
7. ***O(n!)*** — Факториальная сложность. Время выполнения растёт очень быстро при увеличении входных данных, например, в задаче перебора всех возможных перестановок.

Важные моменты:

* Нотация Big O показывает наихудший сценарий работы алгоритма.
* Не учитываются константы и меньшие степени при оценке сложности. Например, если алгоритм имеет сложность O(3n+5) то его сложность оценивается как O(n), так как константы 3 и 5 не играют значительной роли при больших n.

### 1.7 Структуры данных и основные алгоритмы для работы с ними (стеки, очереди, хеш таблицы, деревья, сортировки, обходы)

***Стек (Stack)*** — это структура данных, где элементы добавляются и удаляются с одного конца, называемого вершиной. Принцип работы стека — «последним пришел — первым вышел» (LIFO, Last In First Out).

Основные операции:

* push — добавление элемента на вершину стека
* pop — удаление элемента с вершины
* peek — просмотр элемента на вершине без удаления
* isEmpty — проверка, пуст ли стек

Применение:

* Рекурсия и управление вызовами функций
* Обратный порядок операций (например, в обратной польской нотации)
* Отмена операций (в текстовых редакторах)

***Очередь (Queue)*** — это структура данных, работающая по принципу «первым пришел — первым вышел» (FIFO, First In First Out). Элементы добавляются в конец очереди и извлекаются с начала.

Основные операции:

* enqueue — добавление элемента в конец очереди
* dequeue — удаление элемента из начала очереди
* peek — просмотр первого элемента
* isEmpty — проверка, пуста ли очередь

Применение:

* Управление задачами в операционных системах
* Моделирование реальных очередей (например, в банках)
* Алгоритмы поиска в графах (BFS)

***Дек (Deque, двусторонняя очередь)*** — это обобщение очереди, где элементы могут добавляться и удаляться как с начала, так и с конца.

Основные операции:

* addFirst / addLast — добавление элемента в начало/конец
* removeFirst / removeLast — удаление элемента с начала/конца
* peekFirst / peekLast — просмотр элемента на начале/конце

Применение:

* Реализация очередей с приоритетом
* Моделирование двусторонних структур данных (например, двусторонние буферы)

***Хеш-таблица (Hash Table)*** — структура данных, которая позволяет хранить пары «ключ-значение». По ключу вычисляется индекс в массиве с помощью хеш-функции, что обеспечивает быстрый доступ к данным.

Основные операции:

* put(key, value) — добавление пары "ключ-значение"
* get(key) — получение значения по ключу
* remove(key) — удаление элемента по ключу
* containsKey(key) — проверка наличия ключа

Применение:

* Реализация ассоциативных массивов (словари)
* Кеширование данных
* Быстрый доступ к данным при минимальных временных затратах

***Деревья (Trees)*** — это иерархическая структура данных, где один элемент (корень) связан с несколькими подэлементами (потомками). Наиболее распространенным типом деревьев является бинарное дерево, где каждый узел имеет не более двух потомков.

Виды деревьев:

* Бинарное дерево — каждое звено имеет до двух потомков.
* Бинарное дерево поиска (BST) — для каждого узла левое поддерево содержит элементы меньше значения узла, а правое — больше.
* AVL-дерево — сбалансированное бинарное дерево поиска, где для каждого узла разница высот левого и правого поддеревьев не более 1.

Основные операции:

* Вставка, удаление и поиск элемента
* Обходы дерева (прямой, обратный, симметричный)

Применение:

* Организация данных в файловых системах
* Реализация динамических наборов данных
* Парсинг выражений (например, математических)

***Алгоритмы сортировки***

Основные алгоритмы:

1. Пузырьковая сортировка (Bubble Sort) — последовательное сравнение и обмен соседних элементов, "всплывание" большего элемента к концу массива
2. Сортировка выбором (Selection Sort) — на каждом шаге выбирается наименьший элемент из неотсортированной части массива и ставится на правильное место
3. Сортировка вставками (Insertion Sort) — элементы добавляются в отсортированную часть массива один за другим
4. Быстрая сортировка (Quick Sort) — рекурсивное разбиение массива на две части относительно опорного элемента и сортировка каждой из них.
5. Сортировка слиянием (Merge Sort) — деление массива на две части и их рекурсивная сортировка с последующим слиянием отсортированных частей.

Применение:

* Упорядочивание данных для последующей обработки
* Оптимизация поиска (бинарный поиск требует предварительной сортировки данных)
* Управление приоритетами (например, в планировании задач)

***Обходы деревьев*** — это процесс посещения всех узлов дерева в определённом порядке. Существуют три основных типа обходов бинарных деревьев:

1. Прямой (pre-order): посещение узла -> левое поддерево -> правое поддерево
2. Симметричный (in-order): левое поддерево -> посещение узла -> правое поддерево. Часто используется для получения отсортированных данных из бинарного дерева поиска
3. Обратный (post-order): левое поддерево -> правое поддерево -> посещение узла

Применение обходов:

* Прямой обход используется для копирования дерева
* Симметричный обход полезен для получения отсортированного списка из бинарного дерева поиска
* Обратный обход помогает удалять дерево снизу вверх, например, при удалении узлов дерева

### 1.8 Принципы ООП

***Объектно-ориентированное программирование (ООП)*** — это подход к разработке программного обеспечения, который базируется на концепции объектов и классов. Основные принципы ООП включают *инкапсуляцию, наследование, полиморфизм и абстракцию*. Давайте рассмотрим каждый из них с примерами.

***1. Инкапсуляция.*** Инкапсуляция означает, что данные и методы, которые работают с этими данными, объединяются в один объект. Это помогает скрыть детали реализации объекта от внешнего мира и предоставляет только те методы, которые позволяют безопасно работать с его состоянием.

Например, банковский аккаунт скрывает баланс и методы для его изменения (например, метод пополнения или списания), чтобы пользователь не мог напрямую изменять баланс.

***2. Наследование.*** Наследование позволяет создавать новый класс на основе уже существующего. Новый класс (наследник) получает все свойства и методы базового класса, а также может добавлять свои собственные или изменять унаследованные. Этот принцип помогает избежать дублирования кода и создает иерархию классов.

В приложении для учета сотрудников можно создать базовый класс «Сотрудник», содержащий общие характеристики, такие как имя, возраст и должность. На его основе можно создать классы «Инженер» и «Менеджер», добавляя уникальные свойства и методы, специфичные для каждой роли.

***3. Полиморфизм.*** Полиморфизм позволяет использовать один и тот же метод или операцию для объектов разных классов, предоставляя каждому классу свою собственную реализацию. Благодаря этому разные объекты могут реагировать на одно и то же действие по-разному, что облегчает расширение и сопровождение кода.

В системе обработки платежей разные виды платежных методов, такие как кредитные карты, электронные кошельки и банковские переводы, могут реализовать один и тот же метод обработатьПлатеж(), но логика обработки для каждого метода будет своя.

***4. Абстракция.*** Абстракция — это выделение значимых характеристик объекта, которые важны для решения текущей задачи, при сокрытии незначительных деталей. Она позволяет сосредоточиться на том, что делает объект, а не на том, как он это делает.

В автомобиле есть педаль газа. Водителю достаточно знать, что нажатие на нее приводит к ускорению машины. Технические детали (как работает двигатель или коробка передач) скрыты — они абстрагированы от пользователя, поскольку для него важно только действие (ускорение) и его результат.

### 1.9 Пирамида тестирования, unit-тесты, моки

***Пирамида тестирования*** — это концепция, помогающая распределить тесты в проекте так, чтобы поддерживать баланс между скоростью выполнения тестов, их надежностью и объемом покрытия кода. Она состоит из нескольких уровней: *юнит-тестов (unit tests), интеграционных тестов и end-to-end тестов (или UI-тестов)*. Каждый из этих уровней играет свою роль в проверке качества системы. Давайте рассмотрим каждый из них.

***Уровни пирамиды тестирования***

1. ***Юнит-тесты (Unit tests).*** Юнит-тесты находятся на основании пирамиды и составляют её основную часть. Эти тесты проверяют отдельные модули или функции в изоляции от остальной системы, что делает их быстрыми и дешевыми в выполнении. Юнит-тесты позволяют разработчику проверять корректность логики небольших частей кода (обычно отдельных классов или методов), не зависящих от внешних сервисов и данных. Например: проверка функции расчета налога для конкретного типа товара или проверки алгоритма, отвечающего за валидацию данных в форме;
2. ***Интеграционные тесты (Integration tests).*** Интеграционные тесты проверяют взаимодействие между несколькими модулями, а также их работу с внешними зависимостями — базами данных, API и сервисами. Хотя они медленнее и сложнее в поддержке, чем юнит-тесты, их цель — убедиться, что компоненты корректно взаимодействуют друг с другом. Например: тестирование взаимодействия системы с базой данных: выполнение запросов на создание и получение данных;
3. ***End-to-End (e2e) тесты или UI-тесты.*** Эти тесты находятся на вершине пирамиды, так как они наиболее затратны и медленны. E2E тесты проверяют систему целиком, симулируя реальное поведение пользователя. Они проверяют функциональность от начала до конца, включая работу фронтенда, бэкенда, баз данных и других систем, имитируя сценарии использования. Например: проверка всей последовательности оформления заказа на сайте, от выбора товара до завершения покупки.

***Роль моков (mocks) в тестировании***

Моки (mock objects) используются, в основном, на уровне юнит-тестирования, чтобы изолировать тестируемый код от внешних зависимостей. Моки заменяют реальный объект заглушкой, которая позволяет имитировать поведение реального компонента. Это важно, когда тестируемый компонент взаимодействует с внешними системами, такими как базы данных, сторонние сервисы или API, поскольку мок позволяет избежать фактического обращения к этим зависимостям.

Если юнит-тест должен проверить, как компонент реагирует на ответ от API, то вместо реального API можно использовать мок, который имитирует возвращаемые значения. Это позволит контролировать тестовую среду и проверить работу функции в разных сценариях (успешный ответ, ошибка и т.д.).

***Как использовать пирамиду тестирования***

Следуя пирамиде, в проекте предпочтение отдается большому количеству юнит-тестов, средней доле интеграционных тестов и небольшому числу e2e-тестов. Такое распределение снижает затраты на тестирование, позволяет быстрее выявлять ошибки и упрощает отладку кода.

### 1.10 Основные паттерны ООП