Table of contents

О курсе

Данный комплекс предлагает углубленное изучения Java и широкое овладение стандартными технологиями языка, а также платформой Jakarta EE. В нём раскрыты такие важные темы как написание чистого кода, оптимизация алгоритмов, тестирование и сборка приложений, работа с файловой системой и многопоточностью, а также разработка приложений корпоративного уровня.

Комплекс представляет собой 3 логически целостных модуля, расположенных в определённом порядке. Каждый модуль содержит от 2 до 3 лекций и ссылки на видеоуроки. Лекции представляют собой структурированный текстовый материал со вставками изображений для лучшей визуализации, а также полезные ссылки, перейдя по которым учащиеся смогут подробнее ознакомиться с материалом. После каждой лекции расположен тест для контроля знаний, приведены примера кода, демонстрирующие общепринятые практики разработки, а также присутствует статистика, оценивающая прогресс выполнения курса для мотивации учащихся.

Видеоуроки раскрывают обширные темы и позволяют учащимся лучше усвоить материал, а также перенять опыт работы со средой разработки, который не сможет предоставить ни один другой формат. Они длятся в среднем 1 академический час и при необходимости разбиты на 2 части для более комфортного просмотра. Также в каждом модуле присутствует практическое задание, призванное закрепить навыки, полученные во время просмотра лекций и видеоуроков. Они последовательно ставят задачи, приближенные к реальным, перед учащимися, и, в конечном итоге, приводят к разработке приложения корпоративного уровня с написанием тестов и документации.

Особенностью практической части данного комплекса является то, что она состоит не только из пунктов, описывающих последовательность действий для выполнения задачи, но также имеет определённую цель, область применения и техническое задание, что приближает практическую часть к реальным задачам, стоящим перед разработчиком. В каждом файле README с постановкой задачи находятся варианты работы.

Модуль 1

Алгоритмы и сортировки

> **Алгоритм**
>
> Последовательность шагов или инструкций, необходимых для решения определенной задачи или выполнения конкретной операции. Он представляет собой набор логически упорядоченных шагов, которые приводят к определённому результату за конечное время.

Нередко при разработке информационной системы, особенно неопытными разработчиками, понятие «конечного времени» пропадает из виду, уступая другим критериям. Безопасность, масштабируемость и доступность безусловно важны, но иногда именно проблемы в работе низкоуровневых алгоритмов приводят к массированным сбоям системы и недостаткам её функционирования. И если какие-то миллисекунды в работе данного программного средства могут сперва показаться мелочью, то при пересчете на упущенную выгоду корпоративной информационной системы у её разработчиков может случиться легкий шок, а то и полноценный приступ паники.

Среди множества алгоритмов одними из наиболее известных и распространенных, особенно в обучающих целях, являются алгоритмы сортировки. Именно на их примерах будут строиться дальнейшие статьи в рамках данного модуля.

Алгоритмы сортировки (Алгоритмы сортировки)

Коллекции в Java

Коллекции часто используются для работы с данными в Java. Они предоставляют различные подходы к хранению и доступу к объектам, что позволяет разработчикам выбирать наиболее подходящую структуру данных для конкретной

задачи. Коллекции могут использоваться для реализации алгоритмов, структур данных, управления данными и многих других сценариев.

В Java коллекции представлены наследниками интерфейса `Collection`, а также отдельно стоящего интерфейса `Map`. Данная статья не ставит задачу перечислить абсолютно все коллекции и методы для работы с ними, поэтому здесь будут кратко описаны интерфейсы `List, Map, Set, Queue` и основные используемые коллекции, а также принцип их работы. Видеоурок: введение в Collection и List (https://www.youtube.com/watch?v=EIWD7DUqJI8&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=1)

Как уже было сказано, базовым элементом для всех коллекций является интерфейс `Collection`, присутствующий в составе JDK с версии 1.2. С приходом Generics в Java 1.5 данный интерфейс был переработан, также в версии Java 8 он получил методы для работы с лямбда-выражениями. `Collection` определяет методы, которые должны быть реализованы во всех коллекциях, такие как `add(), remove(), contains()`.

Коллекции в Java (Коллекции Java)

Алгоритмы сортировки

Простые алгоритмы сортировки

Одним из основных критериев производительности алгоритма является его временная сложность (O), выражающая зависимость количества операций или времени, необходимых для выполнения алгоритма в зависимости от размера входных данных. Оптимальной сложностью является O(1). При данном значении алгоритм всегда затрачивает одинаковое количество операций и времени вне зависимости от размера входных данных. Такой сложностью обладает, например, доступ к элементу массива. При O(n) время выполнения алгоритма пропорционально размеру входных данных. Данной сложность, называющейся ещё линейной, будет обладать просмотр всех элементов массива. Также зачастую используются O(log(n)), O(n(log(n))), $O(n^2)$, $O(n^3)$. Видеоурок: Больше о Big O нотации и эффективности алгоритмов (https://www.youtube.com/watch?v=_LS5MdZeAD4&list=PLqj7-hRTFl_oDMBjl_EstsFcDAwt-Arhs&index=11)

Классификация алгоритмов сортировки достаточно обширна и по этой причине тут будут приведены лишь основные критерии. Кроме вышеупомянутой временной сложности алгоритма, часто называемой его эффективностью, алгоритмы сортировки также обладают стабильностью (или устойчивостью), затратам дополнительной памяти, чувствительностью к начальному состоянию и ещё нескольким критериям, которые, по мнению автора, не столь важны для образовательной цели и не будут упомянуты здесь.

Под стабильностью алгоритма понимается свойство, гарантирующее сохранение исходного порядка элементов с одинаковыми значениями после сортировки. Затраты дополнительной памяти определяют количество памяти, необходимое для выполнения сортировки, помимо исходного массива данных, включая временные структуры или массивы для обработки элементов. Наконец, чувствительность к начальному состоянию отражает степень влияния изначального состояния (частичной отсортированности) элементов на эффективность работы конкретного алгоритма сортировки.

Кроме того, по принципу действия среди алгоритмов сортировки можно выделить рекурсивные и нерекурсивные алгоритмы. Первые из них используют принцип рекурсии, где задача сортировки разбивается на более мелкие части, которые решаются с использованием того же алгоритма. Примерами рекурсивных сортировок являются быстрая сортировка и сортировка слиянием. Нерекурсивные сортировки не используют принцип рекурсии и обычно реализуются через циклы или итеративные процессы. Сортировка пузырьком и сортировка вставками являются их распространенными примерами.

В данной статье будут рассмотрены следующие алгоритмы сортировки: пузырьковая сортировка, быстрая сортировка, сортировка слиянием и сортировка деревом.

Пузырьковая сортировка проста в реализации, но обладает достаточно низкой эффективностью. Она работает путем прохода по массиву, меняя местами соседние элементы, если они находятся в неправильном порядке. Пузырьковая сортировка имеет квадратичную временную сложность $O(n^2)$ в худшем и среднем случае и линейную O(n) в лучшем случае. Потребление дополнительной памяти составляет O1 благодаря отсутствию необходимости создания дополнительных массивов. Алгоритм быстрой сортировки выбирает опорный элемент, разделяет массив на части, меньшие и большие опорного, и рекурсивно сортирует эти части. В худшем случае эффективность быстрой сортировки также может быть $O(n^2)$, но в среднем и лучшем случае – Olog(n). Наихудшая оценка достигается при неудачном выборе опорного элемента. Для быстрой сортировки затраты дополнительной памяти характеризуются Olog(n). Данная сортировка не является устойчивой, но может стать такой при внесении в критерий сравнения двух элементов не только их значение, но и порядок в массиве.

При сортировке слиянием массив разделяется на две части и рекурсивно сортируется, после чего отсортированные части объединяются в один массив. Данная рекурсивная сортировка достаточно сильно похожа на быструю, однако имеет всегда одинаковую эффективность – O(log(n)), за что платит большим потреблением дополнительной памяти – O(n) из-за необходимости создания дополнительных массивов.

Сортировка деревом за счет дополнительной памяти быстро решает вопрос с добавлением очередного элемента в отсортированную часть массива. Причём в роли отсортированной части массива выступает бинарное дерево. Дерево формируется буквально на лету при переборе элементов. Основная затратная операция – вставка элемента на свое

место в отсортированной части массива здесь решена. Проблема здесь кроется в том, что шанс получения несбалансированного дерева из входящего набора элементов велик, что приводит к временной сложности $O(n^2)$ в худшем случае. В основном же, эффективность данного алгоритма составляет O(log(n) , а затраты на дополнительную память – O(n).

Сортировка пузырьком

Сортировка пузырьком – одна из самых очевидных и простых. Её часто используют в качестве примера при изучении алгоритмов. Во время сортировки пузырьком для n элементов массива необходимо сделать n – 1 проходов по нему, во время каждого из которых соседние элементы сравниваются и меняются местами при необходимости. То есть, если на вход подаётся массив из 5 элементов: [2, 8, 3, 6, 1], недостаточно сделать одну итерацию по циклу, поменяв местами значения примерно так: [2, 3, 6, 1, 8]. В случае нахождения наименьшего элемента на последней позиции для того, чтобы гарантированно отсортировать массив, необходимо проитерироваться (n – 1) * (n – 1) раз, что и приводит к квадратичной сложности.

Пример сортировки пузырьком:

```
public void bubbleSort(int[] array) {
        for (int i = 0; i < array.length - 1; i++) {
            for (int j = 0; j < array.length - 1; j++) {
                if (array[j] > array[j + 1]) {
                    int swap = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = swap;
                }
            }
        }
    }
```

При квадратичном возрастании количества элементов время работы данного алгоритма будет значительно возрастать, что, несмотря на константные затраты дополнительной памяти (O1), делает её неэффективным для прикладного применения.

Быстрая сортировка

Быстрая сортировка является достаточно распространённой и неплохо показывает себя на практике. Её основная идея в рекурсивной сортировке левой и правой от разделительного элемента частей массива. Таким образом, важным этапом при её разработке является оптимальный выбор разделительного (или опорного) элемента. Основных подходов при выборе разделительного элемента существует несколько: выбор первого, среднего, последнего, случайного или медианного элемента. Также существуют несколько методик более сложных и эффективных, о них можно почитать, например, здесь: ссылка. Сигнатура метода выбора разделительного элемента обычно выглядит следующим образом:

int partition (int begin, int end, int [] array) {}

Так как быстрая сортировка не использует дополнительных массивов и работает со ссылкой на исходный массив, в метод также передаются индексы начального и конечного элемента для сортировки. Возвращаемое значение данного метода представляет индекс опорного элемента, слева от которого по окончанию работы метода будут находиться элементы, меньшие, либо равные опорному, а справа – большие опорного. Вариант реализации данного метода с выбором последнего элемента в качестве разделителя (опорного, он же – pivot) представлена ниже:

```
int partition (int begin, int end, int [] array) {
    int pivot = array[end];
    int i = begin - 1;

    for (int j = begin; j < end; j++) {
        if (array[j] <= pivot) {
            i++;
            int temp = array[i];
```

```
            array[i] = array[j];
            array[j] = temp;
        }
    }

    i++;
    array[end] = array[i];
    array[i] = pivot;
    return i;

}
```

Важно понять значение переменных `i` и `j`. `j`, как нетрудно догадаться, увеличивается с каждой итерацией, перебирая элементы массива. Что же представляет `i`, почему оно на единицу меньше индекса начального элемента массива, и почему ее значение увеличивается только при нахождении меньшего либо равного элемента, а также после цикла?

Когда цикл встречает элемент, меньший либо равный опорному, `i` увеличивается и после этого происходит обмен двух элементов. С учетом того, что изначально `j` на единицу впереди `i`, если первые несколько элементов массива меньше либо равны опорному, они меняются местами сами с собой. `i` попросту догоняет `j` на каждой итерации благодаря соблюдению условия `if (array[j] <= pivot)`.

Когда цикл встречает элемент, больший опорного, он идет дальше, увеличивая значение `j`, но оставляя `i` на том же месте. Так, получается, что `i` всегда занимает позицию перед элементом, большим опорного, фактически отслеживая подобные элементы. Наконец, после того, как снова встречается элемент меньший, либо равный опорному, `i` увеличивается, указывая на позицию отслеживаемого элемента, и происходит обмен. Больший элемент уходит направо, а меньший – налево. Что же происходит в конце, ведь опорный элемент (последний в данном случае) занимает позицию справа, хотя логично, что он должен встать на место разделителя?

Именно с этой целью после окончания цикла значение `i` увеличивается вне зависимости от значения каких-либо других переменных. Происходит смена мест опорного элемента, стоящего последним, и последнего элемента, на

котором остановился рост `i`. Получается, что даже при отсутствии при итерации элементов, выполняющих проверку `if (array[j] <= pivot)`, больший опорного элементы все равно поменяется с ним местами, расставив все на корректные позиции – элементы меньшие, чем опорный, и большие, чем опорный. В конце метода возвращается индекс разделительного элемента. Теперь, о самом методе, выполняющим функции быстрой сортировки. Он может выглядеть, например, следующим образом:

```
public void quickSort(int begin, int end, int[] array) {
    if (begin < end) {
        int partitionIndex = partition(begin, end, array);
        quickSort(begin, partitionIndex - 1, array);
        quickSort(partitionIndex + 1, end, array);
    }
}
```

Здесь проверка в начале тела метода служит для того, чтобы проверить необходимость дальнейших действий. Она нужна не столько для начала работы метода при его первичном вызове, сколько для того, чтобы рекурсивные вызовы внутри (вызываемые без проверки значения аргументов) не выполнялись при длине отрезка массива, предназначенного для сортировки, (в данный отрезок не включен опорный элемент), меньше двух элементов. К примеру, если значение `begin` = 0, а значение `end` = 1, то отрезок, состоящий из двух элементов с индексами 0 и 1 может нуждаться в упорядочении. Это логично, ведь несмотря на то, что данные элементы стоят слева от опорного и, следовательно, имеют значение меньше либо равное значению опорного, между собой они могут не соблюдать порядок (к примеру, [2, 1, 3 (опорный), 5, ...]). В то же время при вызове метода для отрезка с длиной 1, а следовательно, равными значениями переменных `begin` и `end`, тело метода не будет выполняться.

Далее, все достаточно просто, вызов метода `partition` (описанного выше) упорядочивает элементы, находя опорный, перемещая элементы с большим значением направо от него, а с меньшим, либо равным, – налево, и возвращая индекс опорного элемента. После окончания его работы сортировка вызывается для левого и правого, по отношению к нахождению опорного элемента, отрезков массива.

Сортировка слиянием

Сортировка слиянием – еще один из распространенных алгоритмов сортировки, который обеспечивает эффективную устойчивую сортировку массивов. Основная идея этого метода заключается в разделении массива пополам, рекурсивной сортировке каждой половины и последующем их объединении (слиянии) в упорядоченный массив.

Одним из ключевых шагов в данной сортировке является слияния двух упорядоченных массивов. Этот этап требует дополнительной памяти, так как он создает временный массив для слияния элементов, однако он же обеспечивает устойчивость сортировки, сохраняя порядок одинаковых элементов. Например, метод слияния может выглядеть следующим образом:

```
public void merge(int[] array, int[] tempArray, int leftIndex, int middleIndex, int rightIndex) {

    for (int i = leftIndex; i <= rightIndex; i++) {
        tempArray[i] = array[i];
    }

    int i = leftIndex;
    int j = middleIndex + 1;
    int resultArrayIndex = i;

    while (i <= middleIndex && j <= rightIndex) {
        if (tempArray[i] <= tempArray[j]) {
            array[resultArrayIndex] = tempArray[i];
            i++;
        } else {
            array[resultArrayIndex] = tempArray[j];
            j++;
        }
```

```
        resultArrayIndex++;
    }

    while (i <= middleIndex) {
        array[resultArrayIndex] = tempArray[i];
        resultArrayIndex++;
        i++;
    }

}
```

Для лучшего понимая того, что происходит в методе, стоит взглянуть на сортировку целиком. Ниже представлены методы sort и `mergeSort`. В данном примере `sort` является точкой входа в сортировку. Внутри него создаётся и заполняется данными копия оригинального массива, после чего вызывается метод `mergeSort`. Данный метод работает рекурсивно, постоянно разбивая массив до размера одного элемента.

В данной реализации вместо настоящего создания массивов с все меньшим количеством элементов в целях экономии памяти и повышения эффективности, данный метод работает с индексами элементов, где `leftIndex` представляет собой индекс крайнего левого элемента для этой итерации разбиения массива, `rightIndex` аналогично представляет собой индекс крайнего правого элемента, а `middleIndex` – индекс середины массива. В тех случаях, когда значение `leftIndex` равняется значению `middleIndex`, работа идет с массивом, состоящим из двух элементов.

```
public void mergeSort(int[] array) {
    int[] tempArray = new int[array.length];
    for (int i = 0; i < array.length; i++) {
        tempArray[i] = array[i];
    }
    mergeSort(array, tempArray, 0, array.length - 1);
}
```

```
public void mergeSort(int[] array, int[] tempArray, int leftIndex, int rightIndex) {
    if (leftIndex < rightIndex) {
        int middleIndex = (leftIndex + rightIndex) / 2;
        mergeSort(array, tempArray, leftIndex, middleIndex);
        mergeSort(array, tempArray, middleIndex + 1, rightIndex);
        merge(array, tempArray, leftIndex, middleIndex, rightIndex);
    }
}
```

После попадания массива в метод `mergeSort`, куда также передается копия массива и индексы начального и конечного элемента для разбиения и последующей сортировки, происходит следующее:

- Проверяется размер массива для сортировки и разбиения. Если левый индекс больше либо равен правому, массив состоит из одного элемента и работа текущего вызова `mergeSort` не имеет смысла, работу нужно передавать дальше – либо работе `mergeSort` для следующей части массива, либо методу merge для их объединения. Если длина массива – 2 и больше, ищется индекс центрального элемента. После этого метод рекурсивно вызывает сам себя, передавая «координаты» нового массива. Так, получая на вход массив [3,7,2] метод разобьет его на массивы [3,7] и [2]. Массив [3,7] (`leftIndex` = 0, `rightIndex`, он же `middleIndex` для прошлой итерации, = 1) в свою очередь будет разбит на массивы [3] и [7]. Здесь важно держать в голове то, что работа идет не с настоящими мини-массивами, а лишь с одним, где мы оперируем индексами начала и конца аналогичных срезов массивов. Про tempArray пока можно забыть, он передается лишь для хранения ссылки на себя, и работа с ним происходит в методе `merge`;
- После разбиения массива [3,7], у которого `middleIndex` = 0, на массивы [3] и [7] аргументы метода будут выглядеть следующим образом. Для [3] `leftIndex` и `rightIndex` будут равны 0 и 0 соответственно. Здесь значение `rightIndex` получено при вызове метода из переданного `middleIndex`. Для [7] `leftIndex` и `rightIndex` будут равны 1 и 1. Здесь, `leftIndex` получен при вызове метода из `middleIndex`, к которому прибавлен 1. Очень важно понять, откуда взялись

эти цифры, потому что в дальнейшем многое будет на них опираться. Итак, равные значения `leftIndex` и `rightIndex` для обоих «мини-массивов» приводят к тому, что проверка `if (leftIndex < rightIndex)` не выполняется и действие данных рекурсивных вызовов оканчивается. Очередь переходит к методу `merge`;

- Метод `merge` принимает тот же массив. Кроме того, он принимает ссылку на массив-копию и индексы элементов, которые будут объединены в оригинальный массив. В примере, описанном выше, метод `merge` будет впервые вызван со значения аргументов `array = [3,7,2]`, `tempArray = [3,7,2]`, `leftIndex = 0`, `middleIndex = 0`, `rightIndex = 1`. Работа метода происходит после деления в методе `mergeSort` на массивы [3] и [7]. Здесь в самом начале освежается состояние массива-копии, так как в дальнейшем порядок хранения значений в нем влияет не только на устойчивость данного алгоритма сортировки, но и в целом на его корректность. Причем, достаточно освежить только ту часть, с которой сейчас будет работать метод – от `leftIndex` до `rightIndex` включительно. Следующим шагом является введение переменных `i` и `j`. Их задача – контролировать итерацию выбранной части массива (среза), а их ограничения – концы двух отрезков: `middleIndex` включительно для левой, а для правой, начинающейся со следующего после `middleIndex` элемента, им является `rightIndex`, также включительно. Переменная `resultArrayIndex` служит для хранения индекса массива, на который будет помещен следующий элемент из среза массива;

- После этого начинается магия: цикл `while` перебирает элементы из обоих частей массива и сравнивает их. Меньший попадает на свою позицию (`resultArrayIndex`) в оригинальный массив, заменяя стоящий там до него элемент. Счетчик для той части среза (левой или правой) увеличивается вместе с `resultArrayIndex`. Причем, вполне возможна ситуация, когда все элементы из левой части оказываются меньше даже первого элемента из правой и цикл заканчивает свою работу со значением i, не проходящим проверку. Кажется, что в таком случае логично после этого цикла сделать ещё один цикл `while` только для правого счетчика и аналогичный ему для левого, но не все так просто. Здесь важно помнить, что несмотря на то, что между собой элементы левой и правой части не отсортированы, внутри своего отрезка они гарантировано расположены по возрастанию. Таким образом, если все элементы из левой части меньше первого элемента из правой (где все элементы расположены по возрастанию), элементы из левой части перезапишут значения, после которых в правильном порядке пойдут элементы из правой части среза.

- Так, когда метод `merge` срабатывает впервые, он работает с отрезками, состоящими из одного элемента. Он сравнивает 3 и 7 и так, как 3 меньше, заменяет 3 в оригинальном массиве само на себя, после чего заканчивает свою работу. 7 остается на том же месте, что и была. По итогу срезы массивов [3] и [7] объединены в [3,7]. В конце работы метода `merge` стек вызовов методов «раскручивается» обратно, вызывая метод `merge` на срезы, предшествующие [3] и [7] (оригинальный массив – [3,7,2]). Так, метод `merge` вызывается на [3,7] и [2], где следуя изложенным выше принципам 2 занимает первую позицию. После замены вместе с `resultArrayIndex` увеличивается и значение `j`, тем самым, завершив работу первого цикла `while` (`rightIndex` = 2, `j` становится равным 3). Тут и пригождается второй цикл `while`. После замены array состоит из [2,7,2], `i` остается равным 0, `middleIndex` – 1, `resultArrayIndex` также равен 1. Копия массива, обновленная в начале метода `merge` содержит элементы [3,7,2]. Начиная с элемента с индексом `i` (0) и до тех пор, пока выполняется условие `while` (i <= `middleIndex`), элементы из tempArray будут замещать элементы array, начиная с позиции `resultArrayIndex` (1). Так, спустя две итерации array будет выглядеть следующим образом: [2,3,7].

Сортировка бинарным деревом поиска

В основе данной сортировки, как и следует из названия, лежит бинарное дерево поиска. Первый элемент становится корнем, дальнейшие элементы со значением больше движутся направо от него, со значением меньше либо равные ему – налево. Для того, чтобы элементы ссылались на своих потомков, имеет смысл применить тип данных подобный Node:

```
class Node{
  int val;
  Node leftNode;
  Node rightNode;
}
```

Для формирования дерева по очереди рекурсивно размещаем элементы:

```
private void addNote(Node node, int value) {
    if (value > node.value) {
        if (node.rightNode == null) node.rightNode = new Node(value);
        else addNote(node.rightNode, value);
    } else if (node.leftNode == null) node.leftNode = new Node(value);
    else addNote(node.leftNode, value);
}
```

После чего симметрично обходим дерево в следующем порядке:

- Левое поддерево
- Текущий элемент
- Правое поддерево

```
private int inorderTraversal(Node node, int[] array, int index) {
    if (node != null) {
        index = inorderTraversal(node.leftNode, array, index);
        array[index++] = node.value;
        index = inorderTraversal(node.rightNode, array, index);
    }
    return index;
}
```

Контроль знаний

1. Какими критериями характеризуются алгоритмы?

2. Какая сортировка более требовательна к памяти: слиянием или быстрая?

3. Какую сложность имеет бинарный поиск?

Коллекции Java

Коллекции, реализующие интерфейс List

Интерфейс List в Java представляет собой упорядоченную коллекцию, позволяющую доступ как по индексу, так и по значению, и допускающую наличие повторяющихся элементов. Существует несколько реализаций List, основными из которых являются **ArrayList** и **LinkedList**.

`ArrayList` – это динамический массив, позволяющий хранить null-элементы. Основная структура основана на обычном массиве, поэтому иногда его описывают как "resizable array". Как и у обычного массива, сложность операций получения элементов из ArrayList - (O(1)). Это реализуется благодаря тому, что область хранения элементов массива непрерывна, а размер элемента фиксирован (ссылки занимают 8 байт для 64-битной архитектуры и 4 байта для 32-битной). Таким образом, зная начальную ячейку массива, достаточно прибавить к ней (n * M) байт, где n - индекс элемента массива, начиная с 0, M - количество памяти в байтах, занимаемое хранимым типом данных. Видеоурок: Внутреннее строение ArrayList (https://www.youtube.com/watch?v=O7wF-chC-ms&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=2)

Операция добавления в конец списка имеет сложность (O(1)), добавление в начало занимает (O(n)) из-за необходимости перемещения всех последующих элементов. При добавлении элементов также периодически приходится расширять массив, когда он переполняется, и эта операция формально стоит O(n), хотя и используется очень эффективная функция System.arraycopy. Но так как расширение производится в полтора раза(int newCapacity = oldCapacity + (oldCapacity >>1);), то общее количество таких расширений - логарифм, то есть пренебрежимо мало по сравнению с O(n), поэтому в среднем сложность O(1).

`LinkedList` – еще одна реализация List, которая принимает любые данные, включая null-значения. Ее особенность заключается в двунаправленной структуре связанного списка, где каждый элемент содержит ссылки на предыдущий и следующий элементы. Операции в начале или конце списка выполняются за постоянное время (O(1)), благодаря возможности перехода к следующему / предыдущему элементу с начала или конца списка. По той же причине доступ к элементам по индексу или значению из середины списка занимают линейное время (O(n)). Несмотря на то, что при

определенных случаях LinkedList может быть эффективнее ArrayList, большое количество оптимизаций и общий механизм работы делают ArrayList значительно более универсальным и распространенным. Видеоурок: Подробнее про LinkedList (https://www.youtube.com/watch?v=zEBMKVycvik&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=7)

Также интерфейс List имеет реализации `Vector` и `Stack`, являющиеся устаревшими и не рекомендованные для применения. `Vector`представляет собой динамический массив, являясь при этом синхронизированной коллекцией, что позволяет использовать ее в многопоточной среде. Это отрицательно сказывается на производительности при его применении, а в случае необходимости использовать многопоточные коллекции стоит обратиться к реализациям из пакета **java.concurrent**. `Stack`, расширяющий Vector, был введен для реализации стека LIFO (Last in - first out). Хотя он частично синхронизирован, он стал менее предпочтительным после появления интерфейса Deque в Java 1.6. Видеоурок: Класс Vector (https://www.youtube.com/watch?v=XpGR6zEhhSc&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=11) Видеоурок: Класс Stack (https://www.youtube.com/watch?v=cvJyy1jMOv0&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=12)

Коллекции, реализующие интерфейс Map. Хэширование

Интерфейс `Map` находится в составе JDK с версии 1.2 и предоставляет базовые методы для работы с данными вида «ключ — значение». Также как и `Collection`, он был дополнен дженериками в версии Java 1.5, а в версии Java 8 появились дополнительные методы для работы с лямбдами. Данный интерфейс реализуют такие коллекции как `HashMap`, `HashTable`, `LinkedHashMap`, `TreeMap`, `WeakHashMap` и другие.

Одной из самых распространенных коллекций, реализующих `Map` является `HashMap`. В своей основе она содержит массив `Bucket`, хранящий внутри себя связанный список `Node`. Перед рассмотрением `Bucket`, стоит взглянуть на `Node`.

`Node` имеет следующий вид:

```
class Node<K,V> {
    int hash;
    K key;
```

```
    V value;
    Node<K,V> next;
}
```

Данный класс параметризован значениями `K` и `V`, представляющими собой ключ и значение. Он также имеет ссылку на следующий узел в пределах одной корзины (о ней чуть позже). Поле hash хранит хеш-код текущего элемента, полученный в результате хеширования ключа.

Хеширование - это процесс преобразования входных данных произвольной длины в фиксированный набор битов определенной длины. Для корректного хеширования тип данных, используемый в качестве ключа для экземпляра HashMap должен переопределить метод `hashCode()`, определенный в `Object`. Также важно соблюдать контракт между методами `equals()` и `hashCode`, а именно:

1. Два объекта, для которых `equals()` возвращает false могут иметь одинакоый хеш. Это явление называется *коллизией*.
2. Два объекта, для которых `equals()` возвращает true обязаны иметь одинаковый хеш-код.
3. Два объекта, для которых `hashCode()` возвращает разный хеш-код не могут быть равны.

Подробнее об этом можно почитать здесь: Контракты equals и hashCode или как оно все там (https://javarush.com/groups/posts/1989-kontraktih-equals-i-hashcode-ili-kak-ono-vsje-tam) Видеоурок: Методы equals и hashCode (https://www.youtube.com/watch?v=NQdwRwbPVCs&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=14)

Алгоритм добавления элемента в HashMap выглядит следующим образом:

1. При вызове метода `.put(O obj)` для аргумента `obj` вычисляется хеш-код и в зависимости от него для элемента подбирается соответствующий `Bucket`. Посколько диапазон значений хеш-кода элементов огромен и массив такого размера создать невозможно/нерационально, хеш-код элемента сжимается до предела количества `Bucket` у

данной `HashMap`. Сложность данной операции - O(1), так как и вычисление хеш-кода и получение `Bucket` (хранящихся в массиве) константны.

2. Создается объект `Node`, хранящий `Key` и `Value` элемента.
3. Если текущий `Bucket` свободен, новосозданный `Node` помещается в него.
4. Если в данном `Bucket` уже есть элемент, значения `equals()` и `hashCode` сравниваются.
5. Если элементы равны - текущее значение `Value` заменяется новым.
6. Если это разные объекты - новый `Node` занимает свою позицию в списке после прошлого элемента и указывает в своем поле `next` ссылку на следующий элемент.

Когда элементов в связанном списке для одного из `Bucket` становится много и сложность итерации по ним O(n) становится дорогой, происходит переход к сбалансированным деревьям, улучшающим эффективность алгоритма до O(log(n)).

Процедура получения элемента по ключу выглядит следующим образом:

1. При вызове метода `.get(K key)` для аргумента `key` вычисляется хеш-код и дальнейшие поиск происходит в соответствующей корзине:
2. Список элементов `Bucket` итерируется, сравнивая значения `equals()` и `hashCode` для двух ключей.
3. При нахождении равных ключей, возвращается значение `Value`.
4. В противном случае возвращается `null`.

Таким образом, обладая сложностью получения добавления элемента в среднем O(1), (O(n) в худшем до перебалансировки и O(log(n)) после неё), HashMap позволяет эффективно хранить элементы в формате ключ-

значение. Видеоурок: Класс HashMap в деталях (https://www.youtube.com/watch?v=qlVF2RErvEU&list=PLqj7-hRTFl_oDMBjl_EstsFcDAwt-Arhs&index=15)

`HashTable` - реализация `Map`, похожая на `HashMap` с аналогичными процедурами получения и добавления элементов. Однако, `HashTable` - синхронизированная потокобезопасная коллекция, что отрицательно сказывается на производительности. По аналогии с `Vector` использование `HashTable` не рекомендуется. Взамен стоит рассмотреть реализации из пакета **java.concurrent**, например, **ConcurrentHashMap**. Видеоурок: Класс HashTable (https://www.youtube.com/watch?v=9hoq5gRb9hE&list=PLqj7-hRTFl_oDMBjl_EstsFcDAwt-Arhs&index=19)

`TreeMap` реализует интерфейс `SortedMap` и использует структуру красно-черного дерева для хранения элементов. В отличие от `HashMap` и `HashTable`, где элементы не упорядочены, `TreeMap` хранит элементы в отсортированном порядке на основе ключей.

При добавлении элемента в `TreeMap`, он автоматически сортируется по ключу. Это делает операции вставки и поиска сложностью O(log(n)) в среднем случае, что делает `TreeMap` эффективным для поиска, удаления и обхода элементов в отсортированном порядке. Однако, операции добавления и поиска могут быть медленнее, чем в `HashMap` и `HashTable`, из-за необходимости поддержания сбалансированности дерева. Видеоурок: Класс Vector (https://www.youtube.com/watch?v=u-ilAwbJWYc&list=PLqj7-hRTFl_oDMBjl_EstsFcDAwt-Arhs&index=17)

Одной из интересных коллекций, реализующей данный интерфейс является также `LinkedHashMap`. Её особенность в том, что каждый вставленный элемент имеет ссылку на предыдущий и на следующий элемент, образуя, тем самым, двунаправленный связанный список. При помощи boolean-параметра `accessOrder`, передаваемого в конструктор, можно указывать, каким образом будет осуществляться доступ к элементам при использовании итератора: по порядку последнего доступа (при значении true) или в том порядке, в котором элементы были вставлены (при значении false - по умолчанию). Однако, наличие дополнительных ссылок на элементы и необходимость дополнительных операций при вставке и получении элементов приводят к большим затратам памяти и меньшей эффективности данной коллекции, что несколько ограничивает область её применения. Видеоурок: Класс Vector (https://www.youtube.com/watch?v=lH-Stxa8zQ8&list=PLqj7-hRTFl_oDMBjl_EstsFcDAwt-Arhs&index=18)

Коллекции, реализующие интерфейс Set

Интерфейс `Set`, присутствующий в JDK с версии 1.2, представляет собой неупорядоченную коллекцию, которая не может содержать дублирующиеся данные. Она является программной моделью математического понятия «множество». Интерфейс Set расширяет интерфейс Collection и добавляет в него несколько методов, таких как методы проверки наличия элемента в множестве и методы добавления и удаления элементов из множества. Основные реализации Set - `HashSet` и наследующий его `LinkedHashSet`, а также `TreeSet`, имеющий иерархию предков NavigableSet -> SortedSet -> Set.

`HashSet` является несложным для понимания, если разобраться в принципе работы HashMap. По сути, HashSet базируется на HashMap и использует внутри себя объект HashMap, а также объект-пустышку (new Object()). В качестве ключа используется добавляемый элемент, а в качестве значения - объект-пустышка. HashSet не гарантирует порядок хранения элементов внутри себя. Данная коллекция сопоставима с HashMap по эффективности и основным их отличием является хранение уникальных элементов у HashSet. Видеоурок: Интерфейс Set и класс HashSet (https://www.youtube.com/watch?v=Gn5b3WDbSUE&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=20)

`TreeSet` имеет в своей основе структуру данных "Красно-черное дерево". Он не хранит порядок добавления элементов, но после каждой вставки и удаления элемента автоматически перебалансирует элементы внутри, поддерживая их в отсортированном порядке. Для элементов, добавляемых в TreeSet реализация методов equals и hashCode не является критической, так как благодаря своей структуре, TreeSet для получения элемента осуществляет бинарный поиск. Видеоурок: Класс TreeSet (https://www.youtube.com/watch?v=P-fcrskPf8U&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=21)

`LinkedHashSet` отличается от HashSet только тем, что в основе лежит LinkedHashMap вместо HashMap. Благодаря этому отличию порядок элементов при обходе коллекции является идентичным порядку добавления элементов. За это LinkedHashSet, как и LinkedHashMap платит меньшей эффективностью и большим затратам памяти. Видеоурок: Класс LinkedHashSet (https://www.youtube.com/watch?v=45HObmaFxHc&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=22)

Таким образом, для типа данных, хранимого внутри экземпляра TreeSet важным является реализация интерфейса Comparable и его метода `compareTo(T o)`, позволяющего сравнивать объект с другим объектом и возвращать значение, соответствующее результату (отрицательное число; ноль; или положительное число, в случае если объект,

на котором вызван метод, меньше; равен; или больше передаваемого как аргумент метода объекта). Альтернативой реализации данного интерфейса является передача в конструктор TreeSet объекта типа Comparator, применяемого обычно для реализации неестественной (unnatural) или внешней сортировки. Comparator представляет собой функциональный интерфейс, так что в конструктор TreeSet его можно передать как лямбда-выражение, представляющее реализацию анонимного класса с переопределенным методом `compare(Object o1, Object o2)`.
Пример:

```
Set<String> s = new TreeSet<>(new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
        });
```

Аналогичен:

```
Set<String> s = new TreeSet<>((o1, o2) -> o1.length() - o2.length());
```

В свою очередь аналогичен:

```
Set<String> treeSet = new TreeSet<>(Comparator.comparing(String::length));
```

Коллекции, реализующие интерфейс Queue

Интерфейс `Queue` появился в JDK в версии 1.5 и представляет собой реализацию очереди FIFO. Помимо методов, определённых в интерфейсе Collection, он определяет дополнительные методы для извлечения и добавления элементов в очередь. Большая часть его реализаций находится в пакете java.concurrent и служит для работы в многопоточной среде. Его две основные реализации вне многопоточной среды: `PriorityQueue` и `ArrayDeque`.

`PriorityQueue` — является единственной прямой реализацией интерфейса Queue (была добавлена, как и интерфейс Queue, в Java 1.5), не считая класса LinkedList, который так же реализует этот интерфейс, но был реализован намного раньше. Особенностью данной очереди является возможность управления порядком элементов. По-умолчанию, элементы сортируются с использованием естественной сортировки (natural ordering), но это поведение может быть переопределено при помощи объекта Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов. Видеоурок: Класс PriorityQueue (https://www.youtube.com/watch?v=B_W9P-6Esbk&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=24)

Операции добавления и удаления элементов в PriorityQueue имеют временную сложность в худшем случае O(log n), где n - количество элементов в очереди, что является достаточно эффективным. При добавлении элемента в PriorityQueue, внутренняя структура данных поддерживает так называемую "пирамидальную" (heap) структуру, где каждый элемент имеет приоритет относительно своих потомков. Это обеспечивает быстрое извлечение элемента с наивысшим приоритетом. Операции доступа к элементам, такие как получение первого элемента (с наивысшим приоритетом), выполняются за константное время O(1). Это обеспечивает эффективность применения данной очереди в тех случаях, где важно быстрое получение приоритетных элементов (например, определённые реализации кэша типа MRU - Most Recently Used), хотя тут важно отметить, что производительность данной коллекции всё ещё далека от желаемой для реализации подобных задач.

`ArrayDeque` — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Интерфейс Deque и реализация ArrayDeque были добавлены в Java 1.6. Эта коллекция представляет собой реализацию с использованием массивов, подобно ArrayList, но не позволяет обращаться к элементам по индексу и хранение null. Как заявлено в документации, коллекция работает быстрее чем Stack, если используется как LIFO коллекция, а также быстрее чем LinkedList, если используется как FIFO. Видеоурок: Введение в Deque и класс ArrayDeque (https://www.youtube.com/watch?v=XhAIbdOAyO8&list=PLqj7-hRTFI_oDMBjI_EstsFcDAwt-Arhs&index=25)

Основной особенностью ArrayDeque является его реализация с использованием динамического массива, подобного ArrayList. Однако, в отличие от ArrayList, ArrayDeque не позволяет обращаться к элементам по индексу, предоставляя вместо этого методы для добавления и удаления элементов как в начале, так и в конце коллекции. Операции

добавления и удаления элементов в ArrayDeque выполняются за константное время (O(1)), что делает его эффективным выбором для динамических структур данных. Как и у ArrayList динамическое изменение размера массива занимает некоторое время, но это лишь незначительно отражается на эффективности методов коллекции.

Дополнительный материал

Подробнее о коллекциях Java как о структурах данных можно почитать здесь: Справочник по Java Collections Framework (https://habr.com/ru/articles/237043/)

Контроль знаний

1. Охарактеризовать интерфейсы List, Set, Queue, Map.
2. Какая из коллекций покажет лучший результат при частых вставках и удалениях из начала и конца массива и редком чтении из середины: ArrayList или LinkedList?
3. Что будет, если включить в расчет хеш-кода свойства, которые могут изменить свое значение?
4. Что такое коллизия?
5. Какова временная эффективность HashMap в лучшем, среднем и худшем случае?
6. Какие коллекции реализуют стек по принципу LIFO, а какие - по принципу FIFO?

Модуль 2

Тестирование и сборка приложений

Maven - это мощный инструмент для управления проектами на языке Java. Он предоставляет средства для автоматизации сборки, тестирования и управления зависимостями проекта. Maven определяет структуру проекта и позволяет настроить различные этапы сборки проекта, такие как компиляция, тестирование и упаковка в JAR-файлы.

JUnit 5 — это одна из наиболее популярных библиотек для тестирования Java-приложений. Она предоставляет набор инструментов для написания и запуска тестов, а также для проверки ожидаемых результатов. JUnit 5 предоставляет более гибкую модель для написания тестов, чем его предшественники, и включает в себя множество новых возможностей.

Mockito — это фреймворк для создания заглушек (mock objects) в Java-приложениях. Он позволяет создавать объекты, которые имитируют поведение реальных объектов, что делает тестирование более простым и эффективным. Mockito хорошо интегрирован с JUnit и может быть использован вместе с ним для написания комплексных тестов.

Тестирование и сборка приложений (Тестирование и сборка приложений)

Работа с файловой системой

Работа с файловой системой (Работа с файловой системой)

Тестирование и сборка приложений

В современной разработке программного обеспечения тестирование играет важную роль. Оно позволяет проверить корректность работы приложения и обнаружить ошибки до того, как они достигнут конечного пользователя. В этом модуле речь будет идти о том, как использовать библиотеку JUnit 5 и фреймворк Mockito для написания тестов Java-приложений.

Однако, перед тем, как перейти к теме использования классов сторонних библиотек в проекте, необходимо разобраться с тем, как именно импортировать подобные ресурсы в локальный проект.

Управление зависимостями

Одна из ключевых функций Maven - управление зависимостями. Для добавления зависимости в проект нужно указать ее координаты (groupId, artifactId и version) в файле pom.xml. Пример:

```
<dependencies>
    <dependency>
        <groupId>org.example</groupId>
        <artifactId>example-library</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>
```

Maven автоматически загрузит эту зависимость из удаленного репозитория и добавит ее в проект.

Жизненный цикл сборки

Жизненный цикл (ЖЦ) сборки в Maven состоит из нескольких этапов, таких как `compile`, `test`, `package`, `install` и `deploy`. Каждый этап выполняет определенные действия, например, компиляцию исходного кода, выполнение тестов,

упаковку проекта и.т.д. Maven автоматически выполняет эти этапы в порядке, определенном в стандартном жизненном цикле сборки.

Этапы жизненного цикла сборки Maven `validate`: Проверка корректности проекта, включая проверку структуры каталогов и наличие необходимых файлов. `compile`: Компиляция исходного кода проекта. `test`: Выполнение тестов проекта. `package`: Упаковка скомпилированного кода и ресурсов в JAR, WAR или другой тип артефакта. `verify`: Проверка качества кода, например, выполнение статического анализа кода или проверка на соответствие стандартам. `install`: Установка собранного артефакта в локальный репозиторий Maven. `deploy`: Развертывание собранного артефакта в удаленный репозиторий или хранилище.

Кроме стандартных этапов жизненного цикла сборки Maven предоставляет возможность создания собственных циклы сборки, определив свои наборы фаз. Подробнее об этом можно почитать здесь: Как создать плагин Maven (https://www.baeldung.com/maven-plugin)

Для запуска цикла сборки Maven необходимо исполнить команду `mvn` в командной строке. Например, чтобы выполнить все этапы цикла сборки, можно использовать команду:

`mvn clean install`

Плагины

Плагины Maven - это инструменты, которые расширяют функциональность среды сборки Maven. Они предоставляют возможность выполнения различных задач во время сборки проекта, таких как выполнение тестов, генерация документации, анализ кода и многие другие. Для использования плагинов Maven необходимо добавить соответствующую конфигурацию в файл pom.xml проекта. Каждый плагин определяется с помощью его groupId, artifactId и version, а также списка целей (goals), которые он может выполнять. Пример:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
```

```
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

В этом примере используется плагин maven-compiler-plugin, который определяет конфигурацию компилятора Java для проекта.

Распространенные плагины Maven:

- `maven-compiler-plugin`: Плагин для компиляции исходного кода Java.
- `maven-surefire-plugin`: Плагин для выполнения тестов JUnit.
- `maven-jar-plugin`: Плагин для упаковки проекта в JAR-файл.
- `maven-install-plugin`: Плагин для установки собранного артефакта в локальный репозиторий Maven.
- `maven-deploy-plugin`: Плагин для развертывания собранного артефакта в удаленный репозиторий.

Surefire

Плагин Maven Surefire используется для запуска и выполнения тестов JUnit в проектах Java. Он автоматически находит и запускает тестовые классы в проекте, выводя результаты выполнения тестов в консоль и генерирует отчеты. Его

использование позволяет запускать тесты выборочно. Пример использования данного плагина для выборочного запуска тестового класса с передачей аргумента выбранной модели для сортировки:

```
<properties>
        //Other Maven properties
        <model>Stub</model>
        <sort>Bubble</sort>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.1.2</version>
                <configuration>
                    <includes>
                        <include>${sort}SortTest</include>
                    </includes>
                    <systemPropertyVariables>
                        <model>${model}</model>
                    </systemPropertyVariables>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

В properties указаны переменные среды, которые будут доступны в Java-коде при помощи метода `System.getEnv("propertyName")`. В конфигурации плагина в `systemPropertyVariables` указаны системные переменные, которые будут доступны в Java-коде при помощи метода `System.getProperty("propertyName")`. Системные переменные подходят для их использования при выборе тестовых классов на этапе тестирования приложения. Эта логика описывается в теге `<includes>` конфигурации плагина Surefire. В данном случае к значению переменной `sort`, переданной при запуске задачи `test` сборщика Maven командой: `mvn test -Dsort=SomeSortValue`, добавляется SortTest, на которое оканчиваются все названия тестовых классов в папке `test/java`.

Настроить исключение определенных тестов из выполнения можно следующим образом:

```
<configuration>
    <excludes>
        <exclude>**/IntegrationTest.java</exclude>
    </excludes>
</configuration>
```

Кроме того, Surefire генерирует отчеты о выполнении тестов в каталог target/surefire-reports.

Добавление JUnit 5 в проект

Для добавления библиотеки JUnit 5 в проект используется следующий тег в `pom.xml`:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
</dependency>
```

Использование JUnit 5 для написания тестов

Ниже приведен пример простого теста с использованием JUnit 5:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertDoesNotThrow;

public class SortTest {

    @Test
    public void shouldNotCauseNullPointerExceptionWhenGivenNullArray() {
        // given
        AbstractModel<?>[] nullArray = null;

        //when
        sortingService.sort(nullArray);

        // assert
        Assertions.assertDoesNotThrow(() -> sortingService.sort(nullArray));
    }

}
```

В этом примере используется класс Assertions, предоставляющий функционал методов-"утверждений". Метод assertDoesNotThrow принимает экземпляр функционального интерфейса `ThrowingSupplier` и проверяет отсутствие выбрасывания исключений результатом выполнения переданного в тело метода `T get() throws Throwable` выражения.

Проверка равенства/неравенства значений

Еще одно применение класса Assertions - проверка равенства/неравенства значений. Пример:

```
@Test
    public void shouldCompareArrayElementsInCorrectOrder() {
        //given
        AbstractModel modelToCompare = factory.createModel(model, 1);

        //assert
        Assertions.assertNotEquals(0, modelToCompare.compareToOtherModel(model));
        Assertions.assertEquals((modelToCompare.compareToOtherModel(model) > 0), isOrderAscending);
    }
```

Здесь две модели сначала проверяются на их разность для того, чтобы можно было их уверенно сравнивать. В тело методов assert***Equals и assert***NotEquals (*** может быть Array, Link и.т.п. методы, переопределяющие условия сравнения значений, например со сравнения простых ссылок на массивы, на перебор и сравнение их значений) передаются так называемые `expected` и `actual`, причем их не стоит путать местами. Так, результат `modelToCompare.compareToOtherModel(model)` ожидаемо должен быть 0.

После проверки на неравенство тестируется порядок сравнения. Свойство `isOrderAscending` обозначает порядок сравнения объектов, при его значении true ожидается, что model (имеющая по умолчанию самое низкое значение), переданная как параметр метода `compareToOtherModel` будет ниже при сравнении и результате вызова данного метода для "большей" `modelToCompare` вернет значение, большее, чем 0.

Добавление Mockito в проект

Для добавления зависимости на Mockito стоит добавить следующий код в файл `pom.xml`:

```
<dependency>
    <groupId>org.mockito</groupId>
```

```
    <artifactId>mockito-core</artifactId>
    <version>4.2.0</version>
    <scope>test</scope>
</dependency>
```

Создание заглушек

При тестировании кода (прежде всего unit-тестировании, но не только) тестируемому элементу часто требуется предоставить экземпляры классов, которыми он должен пользоваться при работе. При этом часто они не должны быть полнофункциональными — наоборот, от них требуется вести себя жестко заданным образом, так, чтобы их поведение было простым и предсказуемым. Они и называются заглушками.

Ниже приведен пример тестового класса, использующего Mockito для создания заглушки:

```
import static org.mockito.Mockito.*;

import org.junit.jupiter.api.Test;

public class MyServiceTest {

    @Test
    public void testDoSomething() {
        // Создание заглушки для зависимости
        Dependency dependency = mock(Dependency.class);

        // Настройка поведения заглушки
        when(dependency.someMethod()).thenReturn("mocked result");

        // Создание объекта класса, который мы тестируем
```

```
        MyService myService = new MyService(dependency);

        // Вызов метода, который мы тестируем
        String result = myService.doSomething();

        // Проверка результата
        assertEquals("mocked result", result);
    }

}
```

Проверка void-методов

Иногда может потребоваться создать заглушку для метода, который возвращает void. Для этого в Mockito используется метод doNothing(). Ниже приведен пример:

```
@Test
public void testDoSomethingVoidMethod() {
Dependency dependency = mock(Dependency.class);
doNothing().when(dependency).voidMethod();

    // вызов метода
    myService.doSomethingWithVoidMethod();

    // проверка, что метод был вызван
    verify(dependency).voidMethod();

}
```

Проверка факта вызовов методов

Mockito можно использовать для проверки вызовов методов объектов. Например:

```
@Test
    //проверка отсутствия вызовов метода
    public void shouldNotCallInnerMethodsWhenGivenEmptyArray() {
        // given
        AbstractModel<?>[] emptyArray = new AbstractModel[]{};

        // when
        sortingService.sort(emptyArray);

        // then
        verify(sortingService, never()).mergeSort(any(), any(), anyInt(), anyInt());
        verify(sortingService, never()).merge(any(), any(), anyInt(), anyInt(), anyInt());

    }

    //проверка наличия вызова методов
    @Test
    public void shouldCallInnerMethodsAtLeastFewTimesWhenGivenNonEmptyArray() {

        // given
        AbstractModel<?>[] array = factory.createModels(10, model);

        // when
        sortingService.sort(array);
```

```
        // then
        verify(sortingService, atLeast(3)).mergeSort(any(), any(), anyInt(), anyInt());
        verify(sortingService, atLeast(3)).merge(any(), any(), anyInt(), anyInt(), anyInt());
    }
```

Создание заглушек с аргументами

Способ передачи аргументов в метод thenReturn() при создании заглушек:

```
@Test
public void testMethodWithArguments() {
    InnerService innerService = mock(InnerService.class);
    when(innerService.methodWithArgument(eq("input"))).thenReturn("output");

    //внутри вызывает метод внутреннего сервиса (сервиса-зависимости), поведение которого было
переопределено выше
    String result = mainService.doSomethingWithArgument("input");

    assertEquals("output", result);
}
```

Исключения и тестирование их вызовов

```
@Test
public void testMethodThrowsException() {
    Dependency dependency = mock(Dependency.class);
    when(dependency.methodThatThrowsException()).thenThrow(new RuntimeException());
```

```
    assertThrows(RuntimeException.class, () -> {
        myService.doSomethingThatThrowsException();
    });

}
```

Контроль знаний

1. Охарактеризовать жизненный цикл Maven.

2. Какой класс из библиотеки JUnit 5 используется для "предположений" (методы doesNotThrow, assertTrue и.т.д.)?

3. В чем отличие Mock от Spy?

4. Для чего нужны юнит-тесты?

Работа с файловой системой

Файловая система

Способ организации и хранения файлов на диске компьютера. В Java файловая система представляется в виде иерархии файлов и каталогов, где каждый файл или каталог имеет свое имя и расположение.

Ресурсы файловой системы

Файлы, каталоги, символические ссылки и другие объекты, с которыми можно взаимодействовать в Java приложениях.

Отличительной чертой многих языков программирования является работа с файлами и потоками. В Java основной функционал работы с потоками сосредоточен в классах из пакетов java.io и java.nio.

Ключевым понятием здесь является понятие потока. Хотя понятие "поток" в программировании довольно перегружено и может обозначать множество различных концепций. В данном случае применительно к работе с файлами и вводом-выводом речь будет идти о потоке (stream), как об абстракции, которая используется для чтения или записи информации (файлов, сокетов, текста консоли и т.д.).

Поток связан с реальным физическим устройством с помощью системы ввода-вывода Java. Так, может быть определен поток, связанный с файлом и через который может идти чтение или запись файла. Это также может быть поток, связанный с сетевым сокетом, с помощью которого можно получить или отправить данные в сети. Все эти задачи: чтение и запись различных файлов, обмен информацией по сети, ввод-вывод в консоли решаются в Java с помощью потоков.

Объект, из которого можно считать данные, называется потоком ввода. В свою очередь, объект, в который можно записывать данные, — потоком вывода. Например, если надо считать содержание файла, то применяется поток ввода, а если надо записать в файл - то поток вывода. В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: `InputStream` (представляющий потоки ввода) и `OutputStream` (представляющий потоки вывода). Поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы `Reader` (для чтения потоков символов) и `Writer` (для записи потоков символов).

Все остальные классы, работающие с потоками, являются наследниками этих абстрактных классов. Основные классы потоков:

InputStream	OutputStream	Reader	Writer
FileInputStream	FileOutputStream	FileReader	FileWriter
BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
DataInputStream	DataOutputStream		
ObjectInputStream	ObjectOutputStream		

java-file-streams

Класс File

Класс `File`, определенный в пакете java.io, не работает напрямую с потоками. Его задачей является управление информацией о файлах и каталогах. Хотя на уровне операционной системы файлы и каталоги отличаются, в Java они описываются одним классом `File`.

Создание объекта `File` Для создания объекта `File` можно использовать несколько конструкторов, принимающих имя файла или путь к файлу в виде строки или объекта `Path`. Примеры:

```
// Создание объекта File для файла "example.txt"
File file1 = new File("example.txt");

// Создание объекта File для каталога "myFolder"
File file2 = new File("myFolder");

// Создание объекта File из объекта Path
Path path = Paths.get("example.txt");
File file3 = path.toFile();
```

Основные методы класса `File`:

- boolean createNewFile(): создает новый файл по пути, который передан в конструктор. В случае удачного создания возвращает true, иначе false
- boolean exists(): проверяет, существует ли по указанному в конструкторе пути файл или каталог. И если файл или каталог существует, то возвращает true, иначе возвращает false
- boolean isDirectory(): возвращает значение true, если по указанному пути располагается каталог
- boolean delete(): удаляет каталог или файл по пути, который передан в конструктор. При удачном удалении возвращает true.
- getName(): Возвращает имя файла или каталога.
- getPath(): Возвращает путь к файлу или каталогу.

- isDirectory(): Проверяет, является ли объект файлом или каталогом.
- isFile(): Проверяет, является ли объект файлом.
- mkdir(): Создает каталог.
- long lastModified(): возвращает время последнего изменения файла или каталога. Значение представляет количество миллисекунд, прошедших с начала эпохи Unix

Примеры использования

```
// Проверка существования файла
File file = new File("example.txt");
if (file.exists()) {
System.out.println("Файл существует");
} else {
System.out.println("Файл не существует");
}

// Создание каталога
File directory = new File("myFolder");
if (directory.mkdir()) {
System.out.println("Каталог создан");
} else {
System.out.println("Ошибка при создании каталога");
}

// Удаление файла или каталога
File toDelete = new File("toBeDeleted.txt");
```

```
if (toDelete.delete()) {
System.out.println("Файл удален");
} else {
System.out.println("Ошибка при удалении файла");
}
```

Чтение и запись файлов

Для считывания данных из файла предназначен класс `FileInputStream`, который является наследником класса `InputStream` и поэтому реализует все его методы. Для создания объекта `FileInputStream` можно использовать ряд конструкторов. Одна из версий конструктора в качестве параметра принимает путь к считываемому файлу:

`FileInputStream(String fileName) throws FileNotFoundException`

Если файл не может быть открыт, например, по указанному пути такого файла не существует, генерируется исключение `FileNotFoundException`.

Пример обработки `IOException` в блоке кода, использующем `FileInputStream`:

```
import java.io.*;

public class Program {

    public static void main(String[] args) {

        try(FileInputStream fin=new FileInputStream("notes.txt"))
        {
            int i;
            while((i=fin.read())!=-1){
```

```
                System.out.print((char)i);
            }
        }
        catch(IOException ex){
            System.out.println(ex.getMessage());
        }
    }
}
```

В примере выше экземпляр `FileInputStream` создается прямо в скобках блока `try`. Такая конструкция называется **try-with-resources** и ее работа заключается в автоматической генерации блока `finally` с закрыванием потока, созданного в блоке try. Для использования пользовательских классов внутри этой конструкции необходимо наследование интерфейса `AutoCloseable`.

Подробнее о **try-with-resources** и `AutoCloseable` можно почитать здесь: Оператор try-with-resources (https://javarush.com/quests/lectures/questsyntaxpro.level15.lecture00)

Класс `FileOutputStream` предназначен для записи байтов в файл. Он является производным от класса `OutputStream`, поэтому наследует всю его функциональность. Через конструктор класса `FileOutputStream` задается файл, в который производится запись. Класс поддерживает несколько конструкторов:

- `FileOutputStream(String filePath)`
- `FileOutputStream(File fileObj)`
- `FileOutputStream(String filePath, boolean append)`
- `FileOutputStream(File fileObj, boolean append)`

Файл задается либо через строковый путь, либо через объект `File`. Второй параметр - append задает способ записи: если он равен true, то данные дозаписываются в конец файла, а при false - файл полностью перезаписывается

Использование класса Reader

```
import java.io.*;

public class Program {

    public static void main(String[] args) {

        String text = "Hello world!"; // строка для записи
        try(FileOutputStream fos=new FileOutputStream("notes.txt"))
        {
            // перевод строки в байты
            byte[] buffer = text.getBytes();

            fos.write(buffer, 0, buffer.length);
            System.out.println("The file has been written");
        }
        catch(IOException ex){

            System.out.println(ex.getMessage());
        }
    }
}
```

Работа с байтовыми и символьными потоками

В Java существуют два основных вида потоков: байтовые и символьные. Байтовые потоки (`InputStream` и `OutputStream`) предназначены для работы с байтовыми данными, такими как изображения, аудиофайлы и другие двоичные данные.

Символьные потоки (`Reader` и `Writer`) предназначены для работы с текстовыми данными, где символы представлены в кодировке Unicode. Они предоставляют удобные методы для работы с текстовыми данными, такими как чтение и запись строк, а также автоматическое преобразование символов из и в Unicode. Они также обеспечивают локализацию и поддержку различных кодировок.

Различия между байтовыми и символьными потоками Основное различие между байтовыми и символьными потоками заключается в том, как они обрабатывают данные. Байтовые потоки работают с байтами напрямую, в то время как символьные потоки используют символы и автоматически преобразуют их в байты с помощью указанной кодировки.

Примеры использования:

Байтовые потоки

```
try (InputStream inputStream = new FileInputStream("image.jpg");
  OutputStream outputStream = new FileOutputStream("copy.jpg")) {
  byte[] buffer = new byte[1024];
  int bytesRead;
  while ((bytesRead = inputStream.read(buffer)) != -1) {
    outputStream.write(buffer, 0, bytesRead);
  }
} catch (IOException e) {
  e.printStackTrace();
}
```

Символьные потоки

```
try (Reader reader = new FileReader("text.txt"); Writer writer = new FileWriter("copy.txt")) {
  char[] buffer = new char[1024];
  int charsRead;
  while ((charsRead = reader.read(buffer)) != -1) {
    writer.write(buffer, 0, charsRead);
  }
} catch (IOException e) {
  e.printStackTrace();
}
```

Работа с классом Files и интерфейсом Path

Класс `Path` и интерфейс `Files` в пакете java.nio.file предоставляют удобные средства для работы с путями к файлам и каталогам в Java. В этой части мы рассмотрим основные методы этих классов и интерфейсов, а также рассмотрим примеры их использования.

Методы интерфейса `Path`:

- String getFileName(): возвращает имя файла или директории, на который указывает путь
- boolean isAbsolute(): возвращает true, если путь абсолютный, иначе false
- boolean exists(): возвращает true, если файл или директория по указанному пути существует, иначе false
- boolean isRegularFile(): возвращает true, если путь указывает на обычный файл, иначе false
- boolean isDirectory(): возвращает true, если путь указывает на директорию, иначе false
- Path getParent(): возвращает родительскую директорию пути

- Path toAbsolutePath(): возвращает абсолютный путь
- Path resolve(Path other): объединяет текущий путь с другим путем и возвращает новый путь
- Path relativize(Path other): возвращает относительный путь от текущего пути до другого пути

Пример использования интерфейса `Path`:

```
// Создание объекта Path из строки
Path path = Paths.get("folder", "file.txt");

// Создание объекта Path из URI
Path uriPath = Paths.get(new URI("file:///folder/file.txt"));

// Получение абсолютного пути
Path absolutePath = path.toAbsolutePath();
```

`Files` содержит статические методы для выполнения операций над файлами и каталогами в файловой системе. Он предоставляет методы для чтения, записи, копирования, удаления файлов и многих других операций.

Основные методы интерфейса `Files`:

- static boolean exists(Path path, LinkOption... options): проверяет, существует ли файл или каталог по заданному пути
- static boolean isDirectory(Path path, LinkOption... options): проверяет, является ли путь каталогом
- static Path createDirectory(Path dir, FileAttribute<?>... attrs): создает каталог по заданному пути
- static long/Path copy(Path source, Path target, CopyOption... options): копирует файл или каталог
- static void delete(Path path): удаляет файл или каталог

- static void deleteIfExists(Path path): удаляет файл или каталог, если он существует

Примеры использования методов класса `Files` с применением интерфейса `Path`:

```
// Проверка существования файла
Path path = Paths.get("folder", "file.txt");
boolean exists = Files.exists(path);

// Создание каталога
Path directory = Paths.get("newFolder");
Files.createDirectory(directory);

// Копирование файла
Path sourcePath = Paths.get("sourceFile.txt");
Path targetPath = Paths.get("targetFolder", "targetFile.txt");
Files.copy(sourcePath, targetPath, StandardCopyOption.REPLACE_EXISTING);

// Удаление файла
Path fileToDelete = Paths.get("fileToDelete.txt");
Files.delete(fileToDelete);
```

Использование буферизованных потоков

Для оптимизации операций ввода-вывода используются буферизуемые потоки. Эти потоки добавляют к стандартным специальный буфер в памяти, с помощью которого повышается производительность при чтении и записи потоков.

Класс `BufferedInputStream` накапливает вводимые данные в специальном буфере без постоянного обращения к устройству ввода. Он определяет два конструктора:

- `BufferedInputStream(InputStream inputStream)`
- `BufferedInputStream(InputStream inputStream, int bufSize)` Первый параметр - это поток ввода, с которого данные будут считываться в буфер. Второй параметр - размер буфера.

Пример использования `ByteArrayInputStream` для буферного считывания данных из потока:

```
import java.io.*;

public class Program {

    public static void main(String[] args) {

        String text = "Hello world!";
        byte[] buffer = text.getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(buffer);

        try(BufferedInputStream bis = new BufferedInputStream(in)){

            int c;
            while((c=bis.read())!=-1){

                System.out.print((char)c);
            }
        }
        catch(Exception e){

            System.out.println(e.getMessage());
        }
```

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

}
```

Класс `BufferedInputStream` в конструкторе принимает объект `InputStream`. В данном случае таким объектом является экземпляр класса `ByteArrayInputStream`.

Как и все потоки ввода `BufferedInputStream` обладает методом `read()`, который считывает данные. При этом каждый байт считывается при помощи метода read из массива buffer.

Фактически все то же самое можно было сделать и с помощью одного `ByteArrayInputStream`, не прибегая к буферизированному потоку. Класс `BufferedInputStream` просто оптимизирует производительность при работе с потоком `ByteArrayInputStream`. Естественно вместо `ByteArrayInputStream` может использоваться любой другой класс, который унаследован от `InputStream`.

Класс `BufferedOutputStream` аналогично создает буфер для потоков вывода. Этот буфер накапливает выводимые байты без постоянного обращения к устройству. Когда буфер заполнится, будет произведена запись данных.

`BufferedOutputStream` определяет два конструктора:

- `BufferedOutputStream(OutputStream outputStream)`
- `BufferedOutputStream(OutputStream outputStream, int bufSize)`

Первый параметр - это поток вывода, который унаследован от `OutputStream`, а второй параметр - размер буфера.

Пример использования `BufferedOutputStream` для записи в файл:

```
import java.io.*;
```

```
public class Program {

    public static void main(String[] args) {

        String text = "Hello world!"; // строка для записи
        try(FileOutputStream out=new FileOutputStream("notes.txt");
                BufferedOutputStream bos = new BufferedOutputStream(out))
        {
            // перевод строки в байты
            byte[] buffer = text.getBytes();
            bos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex){

            System.out.println(ex.getMessage());
        }
    }

}
```

Класс `BufferedOutputStream` в конструкторе принимает в качестве параметра объект `OutputStream` - в данном случае это файловый поток вывода `FileOutputStream`. И также производится запись в файл. Опять же `BufferedOutputStream` не добавляет много новой функциональности, он просто оптимизирует действие потока вывода.

Лучшие практики при работе с потоками ввода-вывода в Java:

- Использование try-with-resources: Для автоматического закрытия потоков рекомендуется использовать конструкцию try-with-resources.

- Указание кодировки: При работе с символьными потоками не будет лишним указать нужную кодировку.
- Буферизация: Для увеличения производительности при работе с файлами рекомендуется использовать буферизацию с помощью классов BufferedReader и BufferedWriter.

Контроль знаний

1. Дать определение файловой системе.

2. Какое исключение необходимо обрабатывать при работе с большинством методов файловой системы?

3. Для чего предназначены класс File и интерфейс Files и какие предоставляют методы?

4. Описать отличие байтовых потоков от символьных.

5. Какой интерфейс нужно реализовывать классу для его использования в конструкции try-with-resources?

6. Как работают и для чего нужны буферизованные потоки?

Модуль 3

Клиент-серверная архитектура

> **Клиент-серверная архитектура**
>
> Подход к построению программных систем, при котором функциональность разделяется между клиентской стороной, отвечающей за пользовательский интерфейс и взаимодействие с пользователем, и серверной стороной, предоставляющей данные и услуги, необходимые клиенту.

Этот подход позволяет создавать распределенные системы, где клиенты могут быть различными устройствами или приложениями, подключенными к сети, а серверы обеспечивают централизованный доступ к данным и ресурсам. Клиент-серверная архитектура (Клиент-серверная архитектура)

Инверсия контроля и внедрение зависимостей

Инверсия контроля (IoC) и внедрение зависимостей (DI) - это принципы разработки программного обеспечения, которые помогают создавать более гибкие, масштабируемые и тестируемые приложения. IoC переворачивает управление потоком выполнения приложения, делегируя ответственность за создание и управление объектами контейнеру управления. DI - это техника реализации IoC, при которой зависимости объектов передаются им извне, обычно через конструкторы или методы, вместо того чтобы создаваться внутри объекта самим объектом.

Инверсия контроля и внедрение зависимостей (Инверсия контроля и внедрение зависимостей)

Взаимодействие с базами данных

Взаимодействие с базами данных - это ключевой аспект многих программных приложений, который позволяет им сохранять и извлекать данные из постоянного хранилища. Базы данных предоставляют механизм для организации, хранения и управления данными, а различные технологии и API, такие, как JDBC, JPA и ORM-фреймворки, обеспечивают удобные способы работы с ними из приложений на языке Java.

Взаимодействие с базами данных (Взаимодействие с базами данных)

Клиент-серверная архитектура

Клиент-сервер - вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Фактически клиент и сервер - это программное обеспечение. Обычно эти программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине. Программы-серверы ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде:

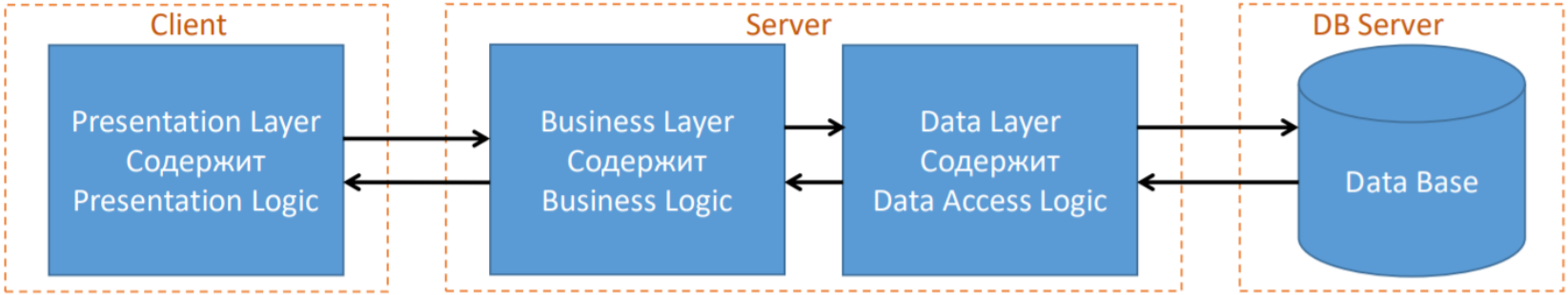
- данных (например, загрузка файлов посредством HTTP, FTP, BitTorrent, потоковое мультимедиа или работа с базами данных);
- сервисных функций (например, работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями или просмотр web-страниц во всемирной паутине).

Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, ее размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило, совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой. Из-за особой роли такой машины в сети, специфики ее оборудования и программного обеспечения, её также называют сервером, а машины, выполняющие клиентские программы, соответственно, клиентами.

Архитектуру «клиент-сервер» принято разделять на три класса: одно-, двух- и трехуровневую. Однако, нельзя сказать, что в вопросе о таком разделении в сообществе ИТ-специалистов существует полный консенсус. Многие называют одноуровневую архитектуру двухуровневой и наоборот, то же можно сказать о соотношении двух- и трёхуровневой архитектур. Далее будет рассмотрена трехуровневая архитектура.

Многоуровневая архитектура «клиент-сервер»

3-уровневая архитектура



Уровень представления

Статический или динамически сгенерированный контент отображаемый через браузер (front-end)

Уровень логики

Уровень подготовки данных для динамически генерируемого контента, уровень сервера приложений (application server). Middleware платформы: JavaEE, ASP.NET, PHP и т. д.

Уровень данных

База данных включающая в себя данный и систему управления над ними или же готовая RDBMS система, предоставляющая доступ к данным и методы управления (back-end)

3layersarch.png

HTTP-протокол {id="http", collapsible="true"}

Изначально HTTP задумывался как протокол передачи HTML-страниц. Долгое время так и было, но сейчас программисты частенько передают по нему и строки, и медиафайлы. В общем, этот протокол универсальный и гибкий, и использовать его действительно просто.

Структура HTTP Сразу стоит отметить, что HTTP-протокол состоит только из текста. Каждое сообщение состоит из трех частей:

1. Стартовая строка (Starting line) — определяет служебные данные.
2. Заголовки (Headers) — описание параметров сообщения.
3. Тело сообщения (Body) — данные сообщения. Должны отделяться от заголовков пустой строкой.

По HTTP-протоколу можно отправить запрос на сервер (request) и получить ответ от сервера (response). Запросы и ответы немного отличаются параметрами.

Пример простого HTTP-запроса:

```
GET / HTTP/1.1
Host: javarush.com
User-Agent: firefox/5.0 (Linux; Debian 5.0.8; en-US; rv:1.8.1.7)
```

В стартовой строке указаны: `GET` — метод запроса; `/` — путь запроса (path); `HTTP/1.1` — версия протокола передачи данных. Затем следуют заголовки: `Host` — хост, которому адресован запрос; `User-Agent` — клиент, который отправляет запрос.

Тело сообщения в запросах типа `GET`, как правило, отсутствует.

В HTTP-запросе обязательны только стартовая строка и заголовок Host.

Методы HTTP {id="methods", collapsible="true"}

HTTP-запрос должен содержать метод. Всего их девять: GET, POST, PUT, OPTIONS, HEAD, PATCH, DELETE, TRACE, CONNECT. Самые распространенные — GET и POST. Кроме того, можно добавлять пользовательские методы, при условии, что и клиентская, и серверная сторона их различают и умеют обрабатывать.

`GET` — запрашивает контент из сервера. Поэтому у запросов с методом `GET` нет тела сообщения. Но при необходимости можно отправить параметры через path в таком формате:

https://url/path/something/?name1=value1&name2=value2

Здесь: url — хост, /path/something — путь запроса, ? — разделитель, обозначающий, что дальше следуют параметры запроса.

В конце перечисляются параметры в формате ключ=значение, разделенные амперсандом.

`POST` — публикует информацию на сервере. POST-запрос может передавать разную информацию: параметры в формате ключ=значение, JSON, HTML-код или даже файлы. Вся информация передается в теле сообщения.

Например:

```
Accept: application/json
Content-Type: application/json
Content-Length: 28
Host: javarush.com

{
"Id": 12345,
"User": "John"
}
```

Запрос отправляется по адресу javarush.com/user/create/json, версия протокола — HTTP/1.1. Accept указывает, какой формат ответа клиент ожидает получить, Content-Type — в каком формате отправляется тело сообщения. Content-Length — количество символов в теле.

HTTP-запрос может содержать много разных заголовков. Подробнее с ними можно ознакомиться в спецификации протокола.

HTTP-ответы {id="responses", collapsible="true"}

После получения запроса, сервер его обрабатывает и отправляет ответ клиенту:

```
Content-Type: text/html; charset=UTF-8
Content-Length: 98

<html>
  <head>
    <title>An Example Page</title>
  </head>
  <body>
    <p>Hello World</p>
  </body>
</html>
```

Стартовая строка в респонсе содержит версию протокола (HTTP/1.1), Код статуса (200), Описание статуса (OK). В заголовках — тип и длина контента. В теле ответа — HTML-код, который браузер прорисует в HTML-страницу. Response Status Codes С телом сообщения и заголовками все ясно, а о кодах статусов стоит сказать пару слов. Response Status Codes всегда трехзначные, и первая цифра кода указывает категорию ответа:

- 1xx — информационный. Запрос получен, сервер готов к продолжению;
- 2xx — успешный. Запрос получен, понятен и обработан;
- 3xx — перенаправление. Следующие действия нужно выполнить для обработки запроса;
- 4xx — ошибка клиента. Запрос содержит ошибки или не отвечает протоколу;

- 5xx — ошибка сервера. Сервер не смог обработать запрос, хотя был составлен верно; Вторая и третья цифры в коде детализируют ответ.

Например:

- 200 OK — реквест получен и успешно обработан;
- 201 Created — реквест получен и успешно обработан, в результате чего создан новый ресурс или его экземпляр;
- 301 Moved Permanently — запрашиваемый ресурс был перемещен навсегда, и последующие запросы к нему должны происходить по новому адресу;
- 307 Temporary Redirect — ресурс перемещен временно. Пока к нему можно обращаться, используя автоматическую переадресацию;
- 403 Forbidden — запрос понятен, но нужна авторизация;
- 404 Not Found — сервер не нашел ресурс по этому адресу;
- 501 Not Implemented — сервер не поддерживает функциональность для ответа на этот запрос;
- 505 HTTP Version Not Supported — сервер не поддерживает указанную версию HTTP-протокола. Вдобавок к статус-коду ответа также отправляется описание статуса, благодаря которому интуитивно понятно, что значит конкретный статус. HTTP-протокол очень практичен: в нем предусмотрено большое количество хедеров, используя которые можно настроить гибкое общение между клиентом и сервером. Все хедеры запросов и ответов, методы запросов и статус-коды ответов невозможно рассмотреть в одной статье. При необходимости можешь почитать официальную спецификацию протокола, которая описывает все нюансы.

HTTP-протокол принято использовать на порту 80, поэтому при работе с адресом, который заканчивается портом 80, можно быть уверенным, что к нему нужно обращаться по HTTP.

С развитием технологий и активным перемещением персональных данных в интернет пришлось задуматься о том, как обеспечить дополнительную защиту информации, которую клиент передает серверу. В результате появился протокол HTTPS. HTTPS синтаксически идентичен протоколу HTTP, то есть использует те же стартовые строки и заголовки. Единственные отличия — дополнительное шифрование и порт по умолчанию (443).

Java Servlets

Java Servlet API — стандартизированный API, предназначенный для реализации на сервере и работе с клиентом по схеме запрос-ответ.

Сервлет — это класс, который умеет получать запросы от клиента и возвращать ему ответы. Сервлеты в Java — именно те элементы, с помощью которых строится клиент-серверная архитектура. Они работают в контексте веб-контейнера, такого как Apache Tomcat или Jetty, и принимают HTTP-запросы от клиента, обрабатывают их и отправляют обратно HTTP-ответы.

Жизненный цикл сервлета Java включает в себя несколько этапов:

1. Инициализация: Сервлет инициализируется веб-контейнером при его первом вызове или при запуске контейнера.
2. Обработка запроса (Request Handling): Каждый раз, когда сервлет получает HTTP-запрос от клиента, он вызывает метод `service()`, который затем вызывает метод `doGet()`, `doPost()`, `doPut()` или `doDelete()` в зависимости от типа запроса. Разработчик должен переопределить соответствующий метод для обработки запроса.
3. Генерация ответа (Response Generation): Сервлет генерирует HTTP-ответ на основе полученного запроса. Ответ может быть HTML-страницей, JSON-данными или любым другим типом данных, в зависимости от логики приложения.
4. Уничтожение (Destruction): При завершении работы веб-контейнера или перезапуске приложения вызывается метод `destroy()`, который позволяет освободить ресурсы, занятые сервлетом.

Контроль знаний

1. Какие существуют уровни клиент-серверной архитектуры?

2. Описать роль клиента, сервера и базы данных в трехуровневой клиент-серверной архитектуре.

3. Для чего предназначены HTTP-протокол и из чего он состоит?

4. В чем основное различие HTTP-методов GET и POST?

5. За что отвечают статусы ответа HTTP групп 100, 200, 300, 400, 500?

6. Описать жизненный цикл сервлета и методы, используемые сервлетами для обработки запросов.

Инверсия контроля и внедрение зависимостей

Модель программирования, принятая в Jakarta EE, позволяет разработчику сосредоточиться именно на том, на чем требуется – то есть, на бизнес-логике. Больше не требуется наследовать классы API; разработчик может излагать логику своей предметной области на обычном языке Java и преимущественно декларативно (при помощи аннотаций) управлять поведением сервера приложений. Таким образом, фреймворк гладко интегрируется в ваш код и, в сущности, его столь же легко оттуда убрать. При проектировании рассчитывайте не на переиспользование, а на легкое удаление.

Внедрение зависимостей (Dependency injection, DI) — это стиль настройки объекта, при котором поля объекта задаются внешней сущностью. Другими словами, объекты настраиваются внешними объектами. DI — это альтернатива самонастройке объектов. Данная практика заключается в том, чтобы передавать объекты, необходимые для работы определенного компонента или модуля, через внешний механизм, такой как конструктор, метод или свойство, вместо того, чтобы создавать эти объекты внутри компонента. Это позволяет сделать компоненты более независимыми и переиспользуемыми. Вместо того чтобы привязывать компоненты к конкретным реализациям, они зависят от абстракций. Это делает их более гибкими и облегчает их тестирование. Внедрение зависимостей также способствует улучшению читаемости кода и его поддержке. Когда зависимости явно передаются в компонент, это делает понятным, какие ресурсы используются, и позволяет быстро понять, какие внешние компоненты взаимодействуют с данным.

Инверсия контроля (Inversion of Control, IoC) - это принцип разработки программного обеспечения, который предполагает, что управление потоком выполнения не осуществляется напрямую компонентами системы, а делегируется внешнему фреймворку или контейнеру.

В контексте IoC контейнер (например, контейнер внедрения зависимостей) принимает на себя ответственность за создание и управление объектами и их зависимостями. Вместо того чтобы компоненты самостоятельно создавать или

получать свои зависимости, они сообщают контейнеру о своих потребностях, и контейнер внедряет эти зависимости при необходимости.

Этот подход позволяет уменьшить связанность (coupling) между компонентами системы, так как компоненты не обязаны заботиться о том, как создавать или настраивать свои зависимости. Они просто объявляют свои зависимости, и контейнер IoC берет на себя заботу об их управлении.

Инверсия контроля при использовании Jakarta

В Jakarta EE компоненты представляют собой модули приложения, такие как сервлеты, EJB (Enterprise JavaBeans), CDI (Contexts and Dependency Injection), JPA (Java Persistence API) и другие. Эти компоненты обеспечивают бизнес-логику, управление данными, взаимодействие с пользователем и другие аспекты приложения.

> **Контейнер Jakarta EE:**
>
> Контейнер Jakarta EE является рантайм-средой, которая управляет жизненным циклом и выполнением компонентов приложения. Он обеспечивает инфраструктуру для создания, управления и взаимодействия с компонентами. Контейнеры могут быть различными для различных типов компонентов, таких как контейнер сервлетов, EJB-контейнер, контейнер CDI и т.д.

Механизмы инверсии контроля в Jakarta EE:

- Аннотации: В Jakarta EE используются аннотации для указания контейнеру о конфигурации компонентов и их зависимостях. Например, аннотация @Inject используется для указания контейнеру, что компоненту нужно внедрить зависимость.

```
public class MyService {
@Inject
private MyDependency dependency;
// ...
}
```

- Конфигурационные файлы: Для настройки компонентов Jakarta EE можно использовать специальные XML-файлы, такие, как web.xml для веб-компонентов (например, сервлетов) и ejb-jar.xml для EJB-компонентов. Эти файлы позволяют определить, какие компоненты должны быть созданы и как они должны быть настроены.
- Контейнеры и управление жизненным циклом: Контейнер Jakarta EE берет на себя ответственность за создание, управление и уничтожение компонентов в соответствии с их жизненным циклом. Например, контейнер сервлетов создает экземпляры сервлетов при необходимости и уничтожает их, когда они больше не нужны.

Пример использования IoC с помощью CDI в Jakarta EE для внедрения зависимости:

```
public class MyService {
@Inject
private MyDependency dependency;

    public void doSomething() {
        dependency.doSomethingElse();
    }

}
```

В этом примере MyService объявляет зависимость от MyDependency с помощью аннотации @Inject. При выполнении приложения контейнер CDI будет внедрять экземпляр MyDependency в MyService.

Аннотации Jakarta EE

1. @Stateless: @Stateless - это аннотация, которая используется для обозначения класса как EJB-компонента не хранящего состояние клиента между вызовами. Такой компонент создается и уничтожается контейнером EJB по мере необходимости.

```
@Stateless
public class MyStatelessBean {
}
```

2. @Inject: @Inject используется для внедрения зависимостей в компоненты. Эта аннотация сообщает контейнеру, что компонент нуждается в экземпляре указанного типа.

```
@Stateless
public class MyStatelessBean {
@Inject
private MyDependency dependency;
}
```

3. @Named: @Named используется для именования компонентов, которые будут использоваться в EL (Expression Language), например, в JSF (JavaServer Faces).

```
@Named("myBean")
@RequestScoped
public class MyBean {
```

```
// Бизнес-методы
}
```

4. @PersistenceContext: @PersistenceContext используется для внедрения EntityManager в компоненты, работающие с базой данных через JPA (Java Persistence API).

```
@Stateless
public class MyStatelessBean {
@PersistenceContext
private EntityManager entityManager;
}
```

Dependency Lookup

> **Dependency Lookup**
>
> это механизм, который позволяет компонентам самостоятельно запрашивать зависимости из контейнера вместо того, чтобы они были внедрены автоматически. В Jakarta EE это часто используется с помощью JNDI (Java Naming and Directory Interface).

```
public class MyBean {
public void doSomething() {
InitialContext ctx = new InitialContext();
MyDependency dependency = (MyDependency) ctx.lookup("java:global/app/MyDependency");
// использование зависимости
```

```
}
}
```

Контроль знаний

1. Что такое инверсия контроля и для чего она нужна?

2. Как инверсия контроля реализована в Jakarta EE и какие конфигурационные файлы нужны для ее настройки?

3. Что такое внедрение зависимостей?

4. Перечислить и описать аннотации, применяемые для внедрения зависимостей в Jakarta EE.

5. Что такое Dependency Lookup?

Взаимодействие с базами данных

Персистентность данных является важным аспектом при разработке Java-приложений, особенно когда речь идет о долгосрочном хранении информации. Базы данных (БД) предоставляют надежный механизм для сохранения данных между запусками приложения и обеспечивают доступ к ним через структурированный интерфейс. В этой статье мы рассмотрим, почему использование баз данных в Java-приложениях является необходимым шагом для обеспечения персистентности данных.

Зачем нужны базы данных?

- Сохранение состояния: Java-приложения могут быть долгоживущими сущностями, и важно сохранять их состояние между запусками. Базы данных предоставляют механизм для сохранения и восстановления этого состояния.
- Обеспечение целостности: БД обеспечивают средства для обеспечения целостности данных, таких как проверка на уникальность, ограничения целостности и транзакции, что помогает предотвратить ошибки и сохранить данные в согласованном состоянии.
- Эффективный доступ: С использованием БД приложения могут эффективно извлекать и обновлять данные, даже при работе с большими объемами информации.

Пример кода:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnection {
private static final String URL = "jdbc:mysql://localhost:3306/mydatabase";
```

```
private static final String USER = "username";
private static final String PASSWORD = "password";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASSWORD);
    }
}
```

`Java Database Connectivity` (JDBC) предоставляет стандартный способ взаимодействия с базами данных из Java-приложений. Он предоставляет API для выполнения SQL-запросов, обработки результатов и управления транзакциями. В этой статье будут рассмотрены основы работы с JDBC для взаимодействия с базами данных в Java-приложениях.

Основные шаги работы с `JDBC`:

- Загрузка драйвера: Перед началом работы с базой данных необходимо загрузить JDBC-драйвер для используемой базы данных.
- Установка соединения: После загрузки драйвера устанавливается соединение с базой данных, используя соответствующие параметры подключения.
- Выполнение запросов: Можно создавать и выполнять SQL-запросы к базе данных с помощью объектов Statement или `PreparedStatement`.
- Обработка результатов: После выполнения запроса мы можем обрабатывать результаты, полученные от базы данных.
- Закрытие ресурсов: Важно закрывать все ресурсы, связанные с соединением с базой данных, после их использования, чтобы избежать утечек ресурсов.

Пример кода:

```
import java.sql.*;

public class JDBCDemo {
public static void main(String[] args) {
try {
Connection connection = DatabaseConnection.getConnection();
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery("SELECT * FROM users");

            while (resultSet.next()) {
                System.out.println("User ID: " + resultSet.getInt("id") + ", Name: " +
resultSet.getString("name"));
            }

            resultSet.close();
            statement.close();
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

}
```

`Java Persistence API` (JPA) - это спецификация Java EE, которая предоставляет стандартный способ работы с объектно-реляционной моделью данных в Java-приложениях. `EntityManager` - основной интерфейс JPA, который используется

для управления жизненным циклом объектов и выполнения операций с базой данных. В этой статье мы рассмотрим, как использовать EntityManager и JPA для работы с базами данных в Java-приложениях.

Основные понятия JPA:

- `Entity`: Объекты, которые могут быть сохранены в базе данных, называются сущностями (entities). Они обычно представляют собой классы Java с аннотациями, указывающими JPA на связь между объектом и таблицей в базе данных.
- `EntityManager`: Этот интерфейс предоставляет методы для выполнения операций над сущностями, такими как поиск, вставка, обновление и удаление.
- `EntityManagerFactory`: Фабрика, которая создает экземпляры EntityManager.
- `Transaction`: Операции над базой данных обычно выполняются в рамках транзакций, которые гарантируют атомарность, согласованность, изолированность и долговечность изменений.

Пример кода:

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.EntityTransaction;
import jakarta.persistence.Persistence;

public class JPAClient {
public static void main(String[] args) {
EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
```

```
        try {
            tx.begin();

            // Создание новой сущности
            User user = new User("John Doe", "johndoe@example.com");
            em.persist(user);

            // Изменение существующей сущности
            user.setEmail("newemail@example.com");
            em.merge(user);

            // Удаление сущности
            em.remove(user);

            tx.commit();
        } catch (Exception e) {
            if (tx != null && tx.isActive()) {
                tx.rollback();
            }
            e.printStackTrace();
        } finally {
            em.close();
            emf.close();
        }
    }
}
```

Контроль знаний

1. Что такое JDBC и JPA?

2. Какие классы используются для установления соединения с базой данных?

3. Что такое EntityManager?

4. Как работать с транзакциями при помощи JPA?

Итоговое задание

Название проекта: Управление задачами (Task Manager)

Постановка задачи: Разработать веб-приложение для управления задачами, которое позволяет пользователям создавать, просматривать, редактировать и удалять задачи. Приложение должно быть построено с использованием Java 17, Jakarta EE, Hibernate для взаимодействия с базой данных, JUnit 5 и Mockito для тестирования.

Описание предметной области: Приложение предназначено для управления задачами в команде или личных проектах. Пользователи могут создавать задачи, указывать им приоритет, статус выполнения, сроки и другие детали. Задачи могут быть назначены на конкретных исполнителей или команды. Пользователи могут просматривать свои задачи, редактировать их и отмечать как выполненные.

Степень покрытия тестами:

Unit-тесты:

Тесты для классов, отвечающих за бизнес-логику (например, классы для работы с задачами, пользователями и т.д.); Тесты для проверки валидации данных (например, ввод корректных и некорректных данных при создании задачи). Интеграционные тесты:

Тесты для проверки взаимодействия с базой данных с использованием Hibernate; Тесты для эндпоинтов API для управления задачами. Mock-тесты:

Использование Mockito для создания заглушек при тестировании классов, взаимодействующих с внешними сервисами или ресурсами.

Описание сущностей:

Пользователь (User):

- Идентификатор (ID)

- Имя пользователя (Username)
- Пароль (Password)
- Электронная почта (Email)
- Роль (Role) - может быть администратором или обычным пользователем
- Команда (Team):

Команда

- Идентификатор (ID)
- Название команды (Team Name)
- Список участников команды (List Members)
- Задача (Task):

Задача

- Идентификатор (ID)
- Название задачи (Task Name)
- Описание задачи (Description)
- Приоритет (Priority)
- Статус (Status) - например, "в процессе", "завершено" и т.д.

- Дата начала (Start Date)
- Дата окончания (Due Date)
- Исполнитель (Assignee) - ссылка на пользователя или команду, которой назначена задача
- Комментарии (Comments) - список комментариев к задаче

Спецификация эндпоинтов:

- GET /tasks - Получить список всех задач.
- GET /tasks/ - Получить информацию о задаче по её идентификатору.
- POST /tasks - Создать новую задачу.
- PUT /tasks/ - Обновить информацию о задаче.
- DELETE /tasks/ - Удалить задачу.
- GET /users//tasks - Получить список задач, назначенных конкретному пользователю.
- GET /teams//tasks - Получить список задач, назначенных конкретной команде.
- POST /users - Создать нового пользователя.
- GET /users - Получить список всех пользователей.
- GET /users/ - Получить информацию о пользователе по его идентификатору.
- PUT /users/ - Обновить информацию о пользователе.
- DELETE /users/ - Удалить пользователя.

- POST /teams - Создать новую команду.
- GET /teams - Получить список всех команд.
- GET /teams/ - Получить информацию о команде по её идентификатору.
- PUT /teams/ - Обновить информацию о команде.
- DELETE /teams/ - Удалить команду.
- POST /teams//members/ - Добавить пользователя в команду.
- DELETE /teams//members/ - Удалить пользователя из команды.

Каждый эндпоинт должен возвращать соответствующие HTTP статусы в зависимости от успешности операции и корректности запроса.