Подсказки по джаве

Этот документ не стоит рассматривать как замену презентациям и вебинарам, тк в нём содержится далеко не всё что вы проходили на лекциях, а только некоторые важные синтаксические моменты и прочие напоминалочки "в двух словах".

## Программирование на Java: переменные, операторы, работа с отладчиком

Метод ("команда") с именем **`main`** - блок кода, с которого начинается выполнение запущенного джава-приложения. Его описание выглядит так:

```java
...
  public static void main(String[] args) {
    // тут описание того что будет выполняться
    // при выполнении метода main
  }
...
```

**Переменная** - ячейка памяти. Её создать можно только внутри метода - она будет создана в момент выполнения джавой этой строки и уничтожена в момент завершения вызова этого метода. Переменная имеет имя, по которому можно к ней обращаться, и тип, регламентирующий что туда можно положить и что можно сделать со значением в ней. Шаблон объявления переменной: `ТИП ИМЯ;`; например, `String name` - ячейка с именем `name` и типом `String`, можем туда класть текст: `name = "Vasya"`.

**Отладчик** - ваш лучший друг, позволяющий ставить на паузу программу в нужном месте и идти по шагам, смотреть на какие строчки кода прыгает выполнение вашей программы и в каких ячейках памяти что лежит в каждый момент времени.

## Примитивные типы данных, условные операторы, выход за границы типов и погрешность вычислений

У каждой переменной есть её тип, определяющий что туда можно положить. Есть группа типов, называемых **примитивными** - всего восемь штук.

В `byte`, `short`, `int` и `long` можно класть числа (отличаются диапазоном допустимых значений); в `double` и `float` - дробные (например, `double height = 1.92;`); в `char` - строго один символ (например, `char c = 'W';`); в `boolean` - значение "да" (`true`) или "нет" (`false`), например, `boolean isBig = false;`. Объявленная переменная видна только внутри тех `{}`, в которых она объявлена, потому она и называется **локальной переменной**.

**Условный оператор** позволяет разветвлить выполнение программы:

```java
if (условие) {
  этот кусок кода будет выполняться если условие окажется правдой
} else {
  этот кусок кода будет выполняться если условие окажется ложью
}
```

## Testability, автотесты, введение в ООП: объекты и методы

**Объект** - сущность, которая имеет свою личную память и уникальные команды. Что объект запоминает и как себя ведёт определяется его типом. Тип объекта обычно описывается классом.

Для создания **класса** вам нужно выбрать ему имя и создать файл Имя.java (в идее это будет просто File -> New -> Class). Выглядеть это будет так:

```java
public class ИмяКласса {
  ...
}
```

Для того чтобы обучить объекты определённого типа новому методу (команде), нужно в классе его типа написать:

```java
  public тип_возвращаемого_значения имяКоманды(тип1 параметр1, тип2 параметр2,...) {
```

```
    // тут код того что нужно делать джаве если команду вызовут


    // если метод должен что-то отдать вызывающему, то нужно написать"
    // return значение_которое_нужно_отдать;
    // если метод ничего не отдаёт вызывающему, то типом возвращаемого значения ставим void
  }
```

например, внутри класса типа объектов `GeometryService` (геометр) метод грубого подсчёта площади круга:

```java
public class GeometryService {
  public double calcArea(double radius) {
    double area = 3.14 * radius * radius;
    return area;
  }
}
```

теперь мы можем в любом другом методе создать объект нашего сервиса:

```java
  GeometryService geo = new GeometryService();
```

и попросить его выполнить команду подсчёта площади круга:

```java
  double a = geo.calcArea(100);
```

## Система сборки Maven, управление зависимостями, автотесты на JUnit5

Программы на этапе разработки это обычно куча всяких файлов (например, .java-файлов) + куча всяких вспомогательных библиотек, кем-то написанных, но которые удобно применить в проекте. Программы которые скачивают вспомогательные инструменты, собирают итоговую программу для

запуска из всех нужных файлов проекта и много что делают ещё называются **сборщиками**. Одним из самых популярных сборщиков в джаве является **Maven**.

В идее уже встроена поддержка мавена, так что ничего устанавливать дополнительно не нужно. Достаточно при создании проекта указать что вы хотите мавен-проект. У вас появятся разные файлы и папки. Например, в `src/main/java` будут основные .java-файлы приложени, а в `src/test/java` - .java-файлы с описанием тестов этого приложения. Вся настройка мавен-проекта будет через файл `pom.xml`.

Чтобы добавить какой-то новый инструмент в проект - например, инструмент для тестирования JUnit - надо отредактировать pom.xml. Точную настройку pom.xml смотрите в лекции и в [напоминалочке](../extra/plugins.md).

У процесса сборки проекта мавеном есть несколько стадий, нас прежде всего будут интересовать `test` (на ней запускаются тесты) и `verify` (на ней делаются дополнительно подключённые вами проверки). Для того чтобы запустить сборку вплоть до нужной стадии, воспользуйтесь панелью-меню мавена в идее справо или же просто нажмите два раза Ctrl и введите `mvn СТАДИЯ`, где СТАДИЯ это нужная вами стадия сборки.

Тесты под JUnit будут писаться в джава-классах в `src/test/java`, каждый тест будет писаться в отдельном методе и помечаться `@Test` над методом. Общая схема теста выглядит следующим образом:

```java
...
@Test
public void имяТеста() {
  // сперва создаём и заполняем всё что нужно для этого теста
  // например, создаём тестируемый объект тестируемого класса

  // затем вызываем тестируемую команду созданного объекта
  // получаем фактический результат

  // просим JUnit проверить что фактический результат совпадает
  // с ожидаемым. если нет, то JUnit обрушит тест, показав что
  // тестируемый объект/метод/... проверку не прошёл
```

```
  // делаем это через assert*
}
...
```

## Циклы, параметризованные тесты и аннотации

**Массив** это объект, олицетворяющий собой пронумерованный набор из однотипных ячеек. Для работы с массивами у джавы особый синтаксис языка.

Для того чтобы создать массив, вам необходимо указать количество его ячеек; изменить это количество у данного объекта потом будет нельзя, только создав новый. Команда создания массива выглядит следующим образом: `ТИП_ЯЧЕЙКИ[] ИМЯ_ПЕРЕМЕННОЙ = new ТИП_ЯЧЕЙКИ[КОЛИЧЕСТВО_ЯЧЕЕК];`.

Здесь и далее под `МАССИВ` понимается переменная/параметр/поле/.. - ячейка, в которой лежит массив. Чтобы достать из ячейки массива укажите её номер: `МАССИВ[НОМЕР_ЯЧЕЙКИ]`. Чтобы положить что-то в ячейку сделайте присваивание по аналогии с обычной переменной: `МАССИВ[НОМЕР_ЯЧЕЙКИ] = ЗНАЧЕНИЕ_ДЛЯ_ЯЧЕЙКИ`.

**Цикл** - указание джаве повторять какой-то кусок кода пока выполняется условие. Самый простой и универсальный это `while`:

```java
while (УСЛОВИЕ) {
  // Этот будет повторяться пока УСЛОВИЕ не станет ложью
}
```

Для работы с массивами удобен цикл for-each:

```java
for (ТИП_ЭЛЕМЕНТА_МАССИВА НАЗВАНИЕ_ПЕРЕМЕННОЙ : МАССИВ) {
  // Этот код выполнится по разу для каждого элемента массива.
```

```
  // Каждый раз в НАЗВАНИЕ_ПЕРЕМЕННОЙ будет лежать очередной элемент массива.
  // Элементы массива будут перебираться по-порядку
}
```

При таком цикле мы на каждой итерации (повторе) не будем знать номер рассматриваемой ячейки, только её содержимое. Чтобы знать и номер ячейки, нужно будет завести дополнительную переменную, например, так:

```java
int index = -1;
for (ТИП_ЭЛЕМЕНТА_МАССИВА НАЗВАНИЕ_ПЕРЕМЕННОЙ : МАССИВ) {
  index = index + 1;
  // Ваш код обработки элемента массива
  // Значение элемента будет в переменной НАЗВАНИЕ_ПЕРЕМЕННОЙ
  // Номер ячейки этого элемента в массиве в переменной index
}
```

Используйте отладчик для пошагового анализца выполнения цикла.

## Выстраивание процесса непрерывной интеграции (CI): Github Actions. Покрытие кода с JaCoCo, статический анализ кода: CheckStyle, SpotBugs

У мавена есть стадия сборки `verify`, к которой мы можем прикрутить различные проверки и дополнительные действия.

[JaCoCo](../extra/plugins.md#jacoco) - плагин для мавена, который позволяет проследить, какие строчки тестируемого джава-кода выполнились хотя бы раз после прогона всех тестов. В режиме отчётов он будет генерировать html-страничку с отчётом о покрытии кода, в режиме проверки - проверять уровень покрытия и обрушать сборку если он меньше вами указанного порога.

**Github Actions** позволяют запускать команды при пушах, пулл-реквестах в репозитории. Мы их настраиваем [так](../extra/plugins.md), чтобы гитхаб запускал мавен-сборку до стадии verify, таким

образом у нас автоматически будут прогоняться тесты и выполняться прикрученные к фазе `verify` проверки. Если какой-то тест или проверка не пройдёт, то гитхаб нам покажет крестик. Мы всегда можем открыть логи Github Actions и прочитать в них как выполнялись указанные нами автоматические действия и что пошло нее так если нас не устраивает их итог.

## Объектно-ориентированное программирование и проектирование

Класс описывает устройство и поведение объектов. Поведение описывается методами (см. выше), а собственная память объектов - **полями**. Описанный ниже класс `Human` заставляет объекты этого класса запоминать у себя "в голове" две вещи: текст в ячейке с именем `name` и целое число в ячейке с именем `age`:

```java
public class Human {
  public String name;
  public int age;

  // методы
}
```

Внутри метода вы можете обращаться к его ячейкам памяти просто по имени (как тут в методе `isTeenager` к полю `age`) или через `this`:

```java
public class Human {
  public String name;
  public int age;

  public boolean isTeenager() {
    // или if (this.age >= 13 && this.age <= 19) {, что тоже самое
    if (age >= 13 && age <= 19) {
      return true;
    } else {
      return false;
```

```
    }
  }
}
```

У каждого члена класса (поля, метода,..) можно настроить доступ. С доступом `private` к этому члену класса можно будет обратиться только в коде этого же класса, с доступом `public` - из любого места программы. Полям обычно не ставят `public`, а если хотят чтобы у объектов можно было получить или поменять значение каких-то данных, то для этого создают public-методы - **геттеры** и **сеттеры** соответственно.

**Early exit** - проверки в начале метода с выходом из него если хотя бы одна не прошла, позволяет не делать объекту недопустимую операцию, например, выставление человеку отрицательного возраста:

```java
...
  public void setAge(int newAge) {
    if (newAge < 0) {
      return;
    }
    age = newAge;
  }
...
```

## Объекты с внутренним состоянием, управление состоянием при тестировании

**Конструктор** это специфичный метод, который вызывается при создании объекта класса. Тк он вызывается в самом конце вызова `new`, то его часто используют чтобы объект мог сделать подготовительные действия, заполнить свои ячейки памяти; все аргументы, переданные при `new` передаются в параметры конструктора.

```java
```

```
public class ИмяКласса {

  public ИмяКласса(ПАРАМЕТРЫ) { // так объявляется конструктор
    // код, который будет выполняться последним шагом при new
  }
}
```

```java
...
ИмяКласса объект = new ИмяКласса(АРГУМЕНТЫ); // последним шагом будет вызван конструктор, в его параметры передадутся АРГУМЕНТЫ
...
```

## Композиция и зависимость объектов. Mockito при создании автотестов

Цикл `for` имеет ещё одну более универсальную форму: `for (int i = 0; УСЛОВИЕ; i++) { блок для повтора }`. В таком виде блок для повтора будет повторяться пока `УСЛОВИЕ` будет выполняться, причём на каждой итерации (повторе) в переменной `i` будет число на единицу большее чем на предыдущем повторе (а начнётся всё с 0).

**Репозиторий** - объект сервиса, отвечающий за хранение других объектов. Обычно не содержит какой-то заумной логики и сосредотачивается чисто на вопросах сохранения, обновления и удаления элементов. Сперва в качестве памяти нашего репозитория будет поле с типом массива. Особенность будет в том, что массив не может меняться в количестве своих ячеек, потому при добавлении элементов нам придётся пересоздавать массив с большим количеством ячеек, копируя всё из старого в новый. В реальности сохранение может быть и не в массив, а в другие структуры данных (коллекции, что будут пройдены позднее) или же вообще в файл, базу данных или куда-нибудь по интернету.

**Менеджер** - объект сервиса, сосредоточенный на бизнес-логике приложения. Чтобы менеджер не задумывался о том как ему хранить объекты, ему обычно передают объект-репозиторий, с помощью которого он всё и делает.

Если у нас один объект использует другой (как менеджер репозиторий), то затрудняется его тестирование. Ведь, например, неправильное поведение на тесте у менеджера может означать, что менеджер не при чём, а виноват неисправный репозиторий. Чтобы избавиться от этой неопределённости, мы можем создать **мок** репозитория - созданный в тесте объект, "притворяющийся" репозиторием в рамках этого теста и умеющий себя "правильно" вести только в рамках него. Создать такой объект и научить его правильно отвечать на часто заранее заготовленные вопросы и ответы можно с помощью **Mockito**.

## Наследование и расширяемость систем. Проблемы наследования

Мы можем у одного класса (**ребёнок**) указать что он `extends` другого класса (**родитель**) (например, `public class Singer extends Person { содержимое класса }`) и тогда, грубо(!) и упрощённо(!!) говоря:

1. В ребёнка "скопируется" всё что было в родителе кроме конструкторов, т.е. у объектов дочернего класса появятся те же поля и методы, что у родительского.

1. Вы можете добавлять новые методы в дочерний класс или же менять поведение тех методов, что вам пришли от родителя.

1. Полиморфизм: джава вам позволит класть в ячейки родительского типа объекты дочернего типа; при вызове метода у объекта будет выполняться та версия метода, которая описана в типе объекта, а не ячейки.

## Исключительные ситуации и их обработка. Тестирование исключений

Если происходит ошибка, джава создаёт объект с описанием проблемы и пытается выйти из каждого вызова в котором она находится, не выполняя их код дальше. Если вы не остановите этот процесс, приложение завершится с ошибкой.

Объект с описанием проблемы - обычный джавовский объект какого-то определённого класса. `Throwable` - предок всех таких классов, которые могут быть типами для таких объектов, его напрямую вы не используете. `Error` - ребёнок `Throwable`, объединяющий в своих потомках типы объектов описания тех ошибок, которые останавливать лучше не надо, в таких случаях лучше программе умереть (самое частое - нехватка оперативной памяти). `Exception` - ребёнок `Throwable`, объединяющий в своих потомках все остальные типы ошибок, ловить их вы можете. У `Exception` есть особый ребёнок - класс `RuntimeException`, ошибки которые описываются объектами этого типа и его потомков джава вас не заставит обрабатывать, в отличие от других потомков `Exception`.

Чтобы поймать ошибку, нужно вокруг кода где она может возникнуть поставить блок `try`; если в коде ниже в блоке `{}` у `try` возникнет ошибка, объект описания которой имеет тип `RuntimeException` (вы

можете поставить и другой) или его потомка, то выполнится блок `catch` и программа не будет умирать дальше:

```java
try {
  код где может возникнуть ошибка
} catch (RuntimeException e) {
  код, который будет выполняться если ошибка произойдёт
  в e будет лежать объект с описанием ошибки
}
```

Чтобы протестировать, что какой-то код выкинет ошибку и именно нужного вам типа, используйте такой ассерт:

```java
assertThrows(КЛАСС_ОБЪЕКТА_ОПИСАНИЯ_ОШИБКИ.class, () -> {
  код, который должен выбросить эту ошибку
});
```

## Интерфейсы для организации малой связности. Обобщённое программирование (Generics)

**Интерфейсом** в джаве вы можете задать набор методов, которые должен реализовать класс (если он указал у себя что реализует этот интерфейс). Любую ячейку (переменную, параметр, поле,..) вы можете объявить с типом интерфейса и положить объект любого класса, имплементирующего этот интерфейс (полиморфизм на интерфейсах). Чтобы объявить, что класс имплементирует какой-то интерфейс, просто укажите это через `implements`: `public class ИмяКласса implements ИмяИнтерфейса {`.

Дженерики-классы и интерфейсы позволяют шаблонизировать описание класса под какой-нибудь тип (синтаксически через угловые скобки - `<>`). Например, интерфейс `Comparable` используется для задания логики сравнения объектов; если вы его укажете `Comparable<ИмяКласса>`, то это будет интерфейс, задающий логику сравнения объектов класса `ИмяКласса` и его потомков. `Comparable<ИмяКласса>` требует реализовать один метод - `public int compareTo(ИмяКласса второйОбъект)` и возвращать отрицательное число, если объект у которого вызвали этот метод

должен считаться меньшим чем `второйОбъект` (параметр), 0 если они должны считаться равными и положительное число если должен считаться больше чем `второйОбъект`.

Для сортировки массива просто вызовите `Arrays.sort(МАССИВ)`. Если вы хотите задать логику сравнения элементов, передайте объект компаратора вторым аргументом.

## Collections Framework. CRUD и тестирование систем, управляющих набором объектов

В джаве есть ряд готовых классов и интерфейсов, облегчающих работу с наборами элементов (вспомним, что массивы не могут добавлять или убирать ячейки). Обычно они называются **коллекциями**.

`List<ТИП_ЭЛЕМЕНТА>` - называется **списком**, интерфейс, объединяющий коллекции пронумерованных элементов. Самая частая реализация этого интерфейса - `ArrayList`: например, `List<String> list = new ArrayList<>();`. Смотреть значения в списке `list.get(ИНДЕКС)` даст вам значение из ячейки по её номеру (индексу), `list.set(ИНДЕКС, ЗНАЧЕНИЕ)` его поменяет. В отличие от массивов, можно добавлять и удалять новые элементы (ячейки).

`Set<ТИП_ЭЛЕМЕНТА>` - интерфейс **множества** - в этой коллекции элементы хранятся без нумерации и не по-порядку, зато добавление, удаление и проверка на наличие работают быстро. Самая часто используемая реализация - `HashSet`, но она требует от типа элементов правильной реализации метода `hashCode` и накладывает ограничения на изменение объектов во время их пребывания во множестве.

`Map<ТИП_КЛЮЧА, ТИП_ЗНАЧЕНИЯ>` - интерфейс **мапы**, также это называют иногда **ассоциативным массивом**. Похож на массив или список, если представить что в качестве индексов у нас могут теперь быть произвольные объекты (а точнее типа `ТИП_КЛЮЧА`). Чтобы положить значение по ключу надо вызвать `map.put(КЛЮЧ, ЗНАЧЕНИЕ)`, чтобы достать - `map.get(КЛЮЧ)`. Самая часто используемая реализация - `HashMap`, но она требует от типа элементов правильной реализации метода `hashCode` у