**Общее правило:**

Для имен классов, переменных и методов используются латинские буквы и цифры.

1. Имя класса, переменной и метода должно начинаться с латинской буквы.
2. Имя класса должно начинаться с заглавной буквы
3. Имя переменной и метода должно начинаться со строчной буквы.
4. Метод convert и переменная name начинаются со строчной буквы.

**Осмысленные имена:**

Имена переменных, классов и методов должны быть осмысленными. Обычно это английское слово, которое обозначает действие или значение переменной. Например, переменные: user, code, js, valid, invalid. Плохой пример, переменные: a1, f2.

**Составные имена:**

Иногда описать единым словом класс, переменную или метод не получается. В этом случае допустимо использовать составное имя. Правила написания составных имен. Второе слово всегда начинается с заглавной буквы. Такой стиль называется CamelCase, потому что форма похожа на верблюда. Например, имена классов с составным именем ValidateInput, UserStore. Имена переменных с составным именем: rightIndex, leftIndex.

Есть еще один стиль сочетания составных имен — это использование символа подчеркивания.

Например, user\_validate. **В Java такой стиль не используется!**

НАЧАЛО.

Давайте сделаем обобщение.

1. Каждый файл java должен содержать класс.

2. Внутри класса должны быть методы.

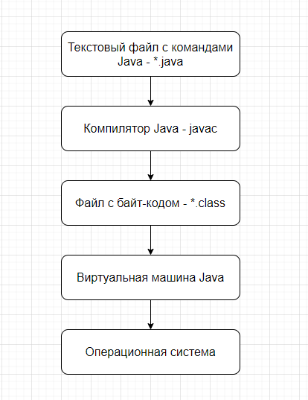
3. Внутри методов находятся операторы.

4. Операторы — это инструкции для выполнения.

Чтобы запустить программу на языке Java, нужно текстовый файл с командами преобразовать в инструкции, понятные компьютеру.

Этот процесс выполняет компилятор Java. Получившийся файл называется файлом с байт-кодом. Чтобы компьютер понял этот байт-код, его нужно запустить через виртуальную машину Java. Виртуальная машина позволяет запустить байт-код на любых операционных системах. Программист не тратит время на адаптацию программы для работы в Windows, Linux или Mac. Это выполняет виртуальная машина.

Виртуальная машина — это прослойка между вашей программой и операционной системой. Изобразим схематично процесс компиляции и запуска:

[](http://job4j.ru/api/images/imageTaskSource?imageId=153)

Когда мы нажимаем кнопку Run в IDEA, то среда выполняет все эти команды.

1. БАЗОВЫЙ СИНТАКСИС.
   1. Типы Данных.

Тип данных - характеристика набора данных, которая определяет:

1) Диапазон возможных значений данных из набора;

2) Допустимые операции, которые можно выполнять с этими значениями;

3) Способ хранения этих значений в памяти.

В Java все типы данных можно разделить на две группы: примитивные и ссылочные типы. Отличаются они по способу хранения в памяти.

В первом модуле мы будем говорить только про примитивные типы данных, а начиная со второго модуля (ООП) начнем активно использовать и ссылочные типы.

Примитивных типов в Java всего 8 штук, а ссылочных бесконечное множество.

**Примитивные типы:**

**byte** – используется для хранения байтовой информации в памяти. 8 bits.

Пример.

**Byte -** mem = 1;

**short** – целочисленный тип. 16 bits. Пример.

short size = 1;

**int** – целочисленный тип. 32 bits. Пример.

int length = 100500;

**long** – целочисленный тип. 64 bits. Пример.

long money = 900500;

**float**– числа с плавающей точкой. 32 bits. Пример.

float size = 1.05F;

**double** – числа с плавающей точкой. 64 bits. Пример.

double size = 100500.99;

**boolean** – описывают логический тип. 1 bit. Пример.

boolean exists = true;

**char** – символьный тип. 16 bits. Пример.

char exit = 'Y';

**Методы:**

Мы можем вызвать методы, но не можем воспользоваться вычисленным результатом. В методе main мы не можем получить доступ к переменным методов func1 и func2. Это свойство называется зоной видимости переменных. К нему мы вернемся в блоке ООП. Сейчас нужно запомнить, что переменные, объявленные в методах func1 и func2 нельзя использовать в методе main. В предыдущих уроках все вычисления выводились на консоль.

Как быть? В Java есть два типа методов: - из метода нельзя получить результат вычисления. - из метода можно получить результат вычисления.

Как понять, что из метода можно получить результат вычисления? Посмотрите на слово перед именем метода.

Если там написано void (с английского "пустой"), то из метода нельзя получить результат вычисления.

Если там написано ключевое слово (byte, short, int, long, float, double, boolean), то мы можем получить результат вычисления от такого метода.

Проверьте себя. Посмотрите на методы func1 и func2. Что написано перед именем метода? Остановитесь и ответьте, можем мы получить результат вычисления?

Правильно. Не можем, потому что методы обозначены, как void.

Отметим, какие изменения появились.

1. В каждом методе ключевое слово void заменено на int.

В каждом методе в конце появился оператор return.

return y;

Этот оператор указывает, значение какой переменной нужно считать результатом вычисления метода.

3. Изменения выше позволяют записать результат в переменную result1.

int result1 = MathFunc.func1(3);

**Преобразование примитивных типов:**

Обратите внимание, что переменная char может описываться не только символом, но и числом - причем диапазон значений попадает в диапазон значений типа int и при этом в отличие от int не может быть отрицательным. Это позволяет задавать значение char с помощью целочисленных значений, однако это не совсем удобно и лучше стараться этого избегать. Далее мы увидим, как можно привести тип char к типу int.

В Java возможны следующие преобразования между типами:

1. Между целочисленными значениями и значениями с плавающей точкой.

2. Целочисленные типы и типы с плавающей точкой в значения char и наоборот, поскольку каждый символ соответствует цифре в кодировке Unicode.

Единственным примитивным типом, который не может быть преобразован не в один из других примитивных типов, является тип boolean. Также любой другой примитивный тип не может быть преобразован в boolean.

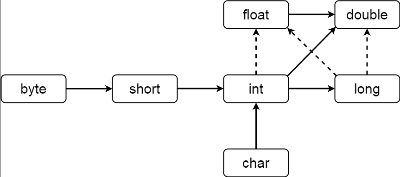
Преобразование типов может быть двух типов: неявное и явное.

Неявное преобразование типов выполняется в случае, если выполняются условия:

1. Оба типа совместимы.

2. Длина целевого типа должна быть больше длины исходного типа.

В этом случае происходит преобразование с расширением, которое можно описать следующей схемой:

[](http://job4j.ru/api/images/imageTaskSource?imageId=189)

на схеме сплошные линии обозначают преобразования, которые выполняются без потери данных. Штриховые линии говорят о том, что при таком преобразовании может произойти потеря точности.

С точки зрения расширяющего преобразования числовые типы данных, в том числе целочисленные и с плавающей точкой, совместимы друг с другом. Однако не существует автоматических преобразований числовых типов в тип char и boolean.

Однако необходимо быть крайне аккуратными, поскольку некоторые преобразования могут производиться автоматически между типами данных одинаковой разрядности или даже от типа данных с большей разрядностью к типу с меньшей разрядностью. На схеме это следующие цепочки: int -> float; long -> float и long -> double. Преобразования производятся без ошибок компиляции, однако при преобразовании мы можем столкнуться с потерей данных

Во всех остальных случаях применяется явное преобразование примитивных типов - реализуется с помощью указания в скобках типа, к которому необходимо привести переменную. Обычно в этом случае выполняются сужающие преобразования от типа с большей разрядностью к типу с меньшей разрядностью.

Сужающее приведение примитивных типов чаще всего используют для преобразования значений с плавающей точкой в целые числа. При этом дробная часть значения с плавающей точкой просто отбрасывается (т.е. значение с плавающей точкой округляется по направлению к нулю, а не к ближайшему целому числу. Получается, что правила округления в данном случае не работают).

Понижающее преобразование для типа char выполняется аналогичным образом. При этом значения варьируются от 0 до 65536. Почему такой большой диапазон? Все из-за того, что char хранит в себе не только буквы, но знаки препинания, иероглифы и т.п. Если же мы передадим значение, которое выходит за пределы от 0 до 65536, то будет происходит усечение и потеря данных.

Возможно столкнуться с ситуацией, когда нам приходится применять различные операции (например, сложение и произведение) над значениями разных типов. В этом случае действуют некоторые правила:

1. Если один из операндов операции относится к типу double, то и второй операнд преобразуется к типу double;

2. Если первое условие не соблюдено, а один из операндов операции относится к типу float, то и второй операнд преобразуется к типу float;

3. Если первые два условия не соблюдены, один из операндов операции относится к типу long, то и второй операнд преобразуется к типу long;

4. В любых других условиях все операнды операции преобразуются к типу int.

если же в операциях будут участвовать данные типа char - то они будут преобразованы к типу int:

* 1. Тестирование.

Что же такое тестирование? *Тестирование* – это процесс выполнения определенного сценария с получением результата, который в конце должен соответствовать ожидаемому.

В целом тестирование можно разделить на *автоматизированное* и *ручное*. Программист должен уметь применять оба подхода для проверки своей программы.

Схематично каждый тест можно представить в виде следующей схемы:

[](http://job4j.ru/api/images/imageTaskSource?imageId=202)

На схеме представлены следующие этапы тестирования:

Определение входных данных. Эти данные должен задавать сам программист.

В зависимости от входных данных программист должен определить ожидаемое поведение или результат.

Составление списка действий и сценария. Тест проверяет выполнение каждого действия по сценарию.

После выполнения сценария в программе будет произведено какое-то действие, изменено состояние программы либо выведен результат.

Завершающий этап. Сравнение реального и ожидаемого результатов работы программы. Если они разные, значит, программа работает неверно.

**Специальные библиотеки для тестирования:**

Библиотека Junit позволяет создавать независимые блоки кода для тестирования.

Есть общее соглашение о создании Junit тестов. При написании тестов на класс мы создаем такой же класс с добавлением к имени суффикса Test.

IDEA позволяет сгенерировать класс тест.

1. Поставьте курсор на имени класса.

[Изображение выглядит как текст

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=204)

2. В меню выберите пункт Code - Generate..

[Изображение выглядит как текст

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=205)

3. В появившемся окне выбрать пункт Test.

[Изображение выглядит как текст

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=206)

4. В появившемся диалоге выбрать пункт JUnit4.

[Изображение выглядит как текст

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=207)

5. Отметить методы, которые мы хотим протестировать.

[Изображение выглядит как текст

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=208)

6. Нажать сохранить.

IDEA сгенерирует каркас класса приведенного ниже. Обратите внимание файл создается в папке test.

[Изображение выглядит как текст, снимок экрана, игра

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=209)

Шаблон сравнивания:

— Assert.assertEquals(expected, out, eps);

**Import:**

Обратите внимание, что для подключения необходимого класса, нам требуется прописать полный путь к файлу в пакете. Однако такое нагромождение имен пакетов не совсем удобно, и в качестве альтернативы мы можем импортировать пакеты и классы с помощью директивы import, которая указывается после директивы package – в этом случае класс примет вид:

Директива import указывается в самом начале кода, после чего идет имя подключаемого класса (в данном случае класс MathFunction).

В данном случае мы подключили только один класс, однако если наш пакет ru.job4j.math будет содержать еще другие классы, то чтобы не подключать их все по отдельности – мы можем подключить весь пакет следующим образом: — import ru.job4j.math.\*;

Иногда может сложиться ситуация, когда мы используем два класса с одним и тем же названием из двух разных пакетов. В таком случае, если нам надо одновременно использовать два этих класса, то необходимо указывать полный путь к этим классам в пакете.

Рассмотрим еще один вариант формы импорта – статический импорт. Давайте обратим внимание, что в нашем примере в нашем классе MathFunction были определены 2 статических метода. Для того чтобы выполнить статический импорт надо вместе с директивой import используется модификатор static:

— import static ru.job4j.math.MathFunction.\*;

В данном случае происходит статический импорт класса MathFunction. Этот класс имеет статические методы и благодаря операции статического импорта мы можем использовать эти методы без названия класса.

**Аннотация:**

В этом уроке мы поговорим об элементе кода - @Test. Этот элемент называется аннотация.

Аннотация — это элементы программы, которые используют другие программы для анализа кода.

Аннотация всегда начинает с символа @. После символа @ идет имя аннотации.

Пример: @Test, @Override.

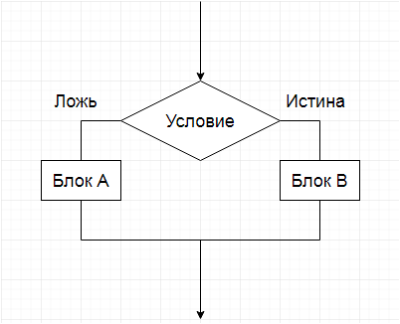
Библиотека JUnit использует @Test для указания, какой метод нужно запускать.

1. JUnit анализирует класс-тест. JUnit ищет методы, которые имеют аннотация @Test.

2. Когда JUnit находит метод с @Test, он запускает этот метод в методе main.

* 1. Операторы ветвления.

Как и в реальной жизни, в программе нужно делать выбор куда идти. Такая конструкция описывается операторами условий. Операторы условий или ветвлений позволяют выполнить определенный блок кода и пропустить другой.

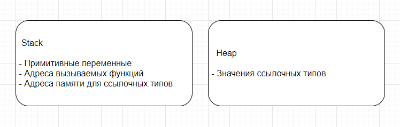
[](http://job4j.ru/api/images/imageTaskSource?imageId=222)

В Java память разбита на две области: Stack и Heap. Примитивные и ссылочные переменные хранятся в разных областях памяти. В этом уроке мы затронем только класс String.

Оператор равенства == для ссылочных типов использовать нельзя. Это связано с тем, что ссылочные типы хранят значение в Heap. В Stack хранятся: значение примитивных переменных, адреса вызываемых методов, адреса памяти для ссылочных типов. Heap содержит значения ссылочных типов.

Оператор равенства == сравнивает значения, которые записаны в переменную.  
В случае со ссылочным типом в переменную записывается адрес памяти, а не сами значения (числа, строки).

В случае с примитивным типом в переменную записываются значения (числа, строки).

[](http://job4j.ru/api/images/imageTaskSource?imageId=223)

Рассмотрим пример сравнения строк через оператор равенства ==.

Изображение выглядит как текст

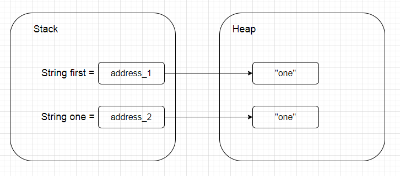
Автоматически созданное описание

Вывод консоли.

[Изображение выглядит как текст

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=224)

Java считает, что строки неравны. Объясним причину.

[](http://job4j.ru/api/images/imageTaskSource?imageId=225)

В переменных first и one хранятся разные адреса памяти. Оператор равенства сравнивает адреса и возвращает false. Запомните: для ссылочных типов нельзя использовать оператор равенства (==). Чтобы проверить равенство ссылочных типов, нужно использовать метод **equals**.

Изображение выглядит как текст

Автоматически созданное описание

Вывод консоли.

[Изображение выглядит как текст

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=226)

Метод equals для String сравнивает значения, записанные в Heap. Если области памяти заполнены одинаково, то метод вернет значение true.

**Блок – схема:**

В этом уроке мы поговорим о блок-схемах.

Блок-схема — это графическое изображение выполнения программы.

На блок-схеме отображаются состояния и связи между состояниями.

1. Начало и конец — это круг. Они будут всегда в ваших схемах.

[](http://job4j.ru/api/images/imageTaskSource?imageId=228)

2. Инструкции — это прямоугольники. В них мы объявляем переменные, вызываем методы.

[Изображение выглядит как стол

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=229)

3. Операторы сравнения — это ромб. У ромба всегда две выходящие стрелки.

[Изображение выглядит как текст, седзи, часы, с плиткой

Автоматически созданное описание](http://job4j.ru/api/images/imageTaskSource?imageId=230)

Внутри ромба указываем выражение, которое принимает значения "истина" или "ложь".

**Тернарное сравнение:**

В этом уроке мы познакомимся с конструкцией тернарного сравнения. Представьте, что вам нужно записать в переменную значение в зависимости от условия. Например, нам нужно вывести надпись можно смотреть фильм или нет.

**Изображение выглядит как текст

Автоматически созданное описание**

Здесь используется оператор if-else. В java этот код можно переписать с помощью тернарного оператора

**Изображение выглядит как текст

Автоматически созданное описание**

Давайте запишем тернарный оператор в виде шаблона:

ТИП\_ПЕРЕМЕННОЙ ИМЯ\_ПЕРЕМЕННОЙ = ЛОГИЧЕСКОЕ\_ВЫРАЖЕНИЕ ? ЗНАЧЕНИЕ\_1 : ЗНАЧЕНИЕ\_2;

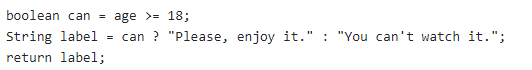
1) ТИП\_ПЕРЕМЕННОЙ - может быть примитивный тип (byte, short, int, long, float, double, boolean, char) или String.

2) ИМЯ\_ПЕРЕМЕННОЙ - произвольное слово, состоящее из букв латинского алфавита.

3) ЛОГИЧЕСКОЕ\_ВЫРАЖЕНИЕ - здесь может быть любая операция сравнения (>, <, >=, <=, ==, !=) или переменная boolean типа.

В нашем примере мы использовали операцию "больше" или "равно".

Перепишем тернарный оператор с использованием boolean переменной.

ЗНАЧЕНИЕ\_1 - значение, которое будет записано в переменную ИМЯ\_ПЕРЕМЕННОЙ, если выражение ИСТИНА (true).

ЗНАЧЕНИЕ\_2 - значение, которое будет записано в переменную ИМЯ\_ПЕРЕМЕННОЙ, если выражение ЛОЖЬ (false).

В нашем примере, если age будет меньше 18, то в переменную label запишется "You can't watch it". Если age будет больше или равен 18, то в переменную label запишется "Please, enjoy it".

**Логические переменные:**

1) **Конъюнкция. Логическое умножение:**

* left - true, right - true = true
* left - false, right - true = false
* left - true, right - false = false
* left - false, right - false = false
* **В Java соответствует оператор [ && - амперсант].**

**2) Дизъюнкция. Логическое сложение:**

* left - true, right - true = true
* left - false, right - true = true
* left - true, right - false = true
* left - false, right - false = false
* **В Java соответствует оператор [ || - pipe ].**

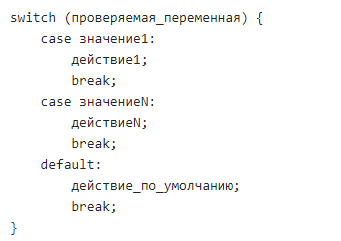
3) Логическое отрицание.

В Java соответствует оператор [ ! ].

**Оператор Switch:**

Операторы условия if или тернарный оператор используется если входные данные четко не ограничены. Например, мы проверяем, что число положительное, мы не проверяем каждое число 1, 2 и тд. В Java есть еще один оператор условия. Это оператор switch. Используется он тогда, когда нам входные данных хорошо известны и нам нужно для каждого возможного значения продумать свою логику.

Синтаксис у него такой:

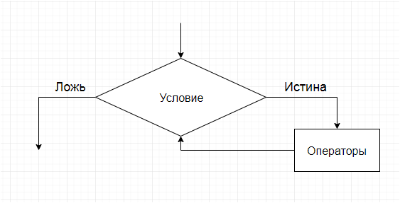


Логика такая, берется значение переменной и сопоставляется с каждый возможным значением, если сопоставление происходит успешно, то выполняется действие, которое есть внутри case. Если ничего сопоставить не удалось, то выполняется блок default.

* 1. Циклы.

Циклы позволяют многократно выполнить блок операций или тела цикла.

Давайте схематично изобразим конструкцию циклов.

[](http://job4j.ru/api/images/imageTaskSource?imageId=252)

Любой цикл содержит элемент, проверяющий необходимость выполнить тела цикла. По аналогии с конструкцией if, блок if выполнится в том случае, если условие внутри оператора if будет истина (true). В цикле то же самое, но после выполнение тела цикла идет повторная проверка условия и так до тех пор, пока условие не поменяет значение на ложь (false).

Начнем мы с цикла for. Цикл for позволяет выполнить заранее известное количество повторений. Опишем шаблон конструкции цикла for.

Изображение выглядит как текст

Автоматически созданное описание

**Инкремент** — операция во многих языках программирования, увеличивающая значение переменной ( i++).

**Декремент** — операция во многих языках программирования, уменьшающая значение переменной (i--).

**Вложенный цикл:**

Разберем использование такой конструкции, как вложенные циклы. Это конструкция выглядит довольно просто - когда мы в теле цикла, который называется внешним циклом, запускаем еще один цикл, который называется вложенным циклом. Каркас конструкции выглядит довольно просто.

Изображение выглядит как текст

Автоматически созданное описание

При этом выполняются следующие условия:

1. Общее количество итераций в конструкции вложенных циклов равно произведению количества итераций для каждого из циклов (при условии что у нас отсутствуют операторы break или continue в каком-либо из циклов).

2. Зона видимости переменных, объявленных во вложенном цикле, ограничивается этим циклом и эту переменную мы не сможем использовать во внешнем цикле, а также вне циклов вообще. Однако мы можем использовать переменную объявленную за пределами циклов вообще, а также переменную объявленную в теле внешнего цикла.

**Цикл while:**

Опишем шаблон создания этого цикла:

**Изображение выглядит как текст

Автоматически созданное описание**

В цикле while заранее неизвестно количество повторений. Давайте рассмотрим ситуацию, когда мы вычисляем ежемесячный доход.

**Изображение выглядит как текст

Автоматически созданное описание**

Тело цикла будет выполняться, пока переменная income меньше, чем 100. Внутри тела мы вычисляем доход. Здесь приведем простой пример, но в большой программе доход может быть разным и просчитать количество повторений через цикл for не будет возможности.

**Оператор прерывания цикла – break:**

Представьте, что вы используете цикл for и по условию вычисления вам нужно оставить цикл на данном этапе.

Чтобы прекратить выполнение цикла, нужно в теле цикла выполнить оператор break.

Изображение выглядит как текст

Автоматически созданное описание

Пример. Программа будет складывать числа от 1 до N, если сумма чисел будет равна N, то напечатать "Success" и выйти из цикла.

Если сумма будет больше N, то на печатать "Not possible" и выйти из цикла.

Например, задано число 6. Его можно получить, сложив 1, 2, 3.

Если последовательное сложение не дает нужного числа, то выходим из цикла.

Для n = 5 сумма последовательности не равна заданному числу.

Изображение выглядит как стол

Автоматически созданное описание

* 1. Массивы.

Давайте начнем с шаблона создания массива.

тип\_данных[] имя\_переменной = new тип\_данных[количество\_ячеек];

Разберемся с каждым элементом в этом шаблоне.

1. тип\_данных - здесь может быть любой: int, double, boolean, char, String.

2. имя\_переменной - по аналогии с любой переменной нужно использовать латинские буквы.

3. new - этот оператор резервирует кусок памяти для данных, которые будут записаны в массив.

4. количество\_ячеек - массив в Java должен иметь фиксированных размер. Максимальный размер массива ограничен размером int.

Как же мы можем узнать размер нашего массива не обращая внимание на строку объявления и инициализации массива? Все очень просто - массив содержит специальное поле length, которое содержит в себе число элементов, которое было указано при объявлении массив. Выведем в консоль размер массива, который мы объявили выше:

Изображение выглядит как текст

Автоматически созданное описание

Массив резервирует в памяти указанное количество ячеек для хранения чисел типа, который мы указали. По умолчанию все элементы в нем заполняются начальными значениями.

Для каждого типа данных он свой.

1. byte, short, int, long - 0.
2. float, double - 0.0
3. char - '' (Пустой символ)
4. boolean - false.
5. String - null (Нулевая ссылка).

Чтобы заполнить элементы массива, нужно обратиться к ним через указатель - индекс. Каждой ячейке массива присвоен уникальный номер, или правильно сказать - индекс. Нумерация индексов начинается с 0.

Для заполнения ячейки массива используется следующая конструкция.



Например:

Изображение выглядит как текст, стол

Автоматически созданное описание

Чтобы прочитать ячейку из массива, можно использовать тот же подход.

Шаблон чтения ячейки из массива.



Например:

Изображение выглядит как текст

Автоматически созданное описание

Или можно использовать вывод переменной на консоль без переменной.



Предположим, что мы хотим записать числа в массив от 0 до 9 **включительно**.

Изображение выглядит как текст

Автоматически созданное описание

**Массивы и цикл for-each:**

Существует конструкция, которая в большинстве случаев упрощает работу с массивами в цикле for() - цикл for-each(). Объявляется он следующим образом:

Изображение выглядит как текст

Автоматически созданное описание

как видите, в данной конструкции отсутствует оператор ручного изменения переменной - шага для перебора. Такой цикл выполняет эту работу автоматически. На каждой итерации нам доступен уже готовый элемент из массива, обращаться к нему по индексу нет необходимости. Работает сразу со значением, а не с индексом!!!!

Изображение выглядит как текст

Автоматически созданное описание  
Вывод:

Изображение выглядит как текст

Автоматически созданное описание

Важно подчеркнуть следующие особенности цикла for-each:

1. Имя переменной, которая доступна на текущей итерации в цикле выбирается произвольно, главное, чтобы внутри цикла вы использовали именно это имя.

2. Тип у переменной и тип данных, которые хранятся в массиве, должны совпадать.

Использование цикла for-each считается предпочтительным, особенно если нам необходимо перебрать и выполнить какие-то действия над всеми элементами в массиве. Как вы уже наверно обратили внимание, никакой привязки к размеру массива в явном виде в таком цикле нет, соответственно выход за границу массива в такой конструкции в качестве проблемы - отпадает.

НО! Если внутри цикла нам необходимо производить какие-то манипуляции с индексом элемента в массиве - такая конструкция уже не подходит - в данном случае будет более предпочтительно использовать запись цикла for() с использованием индексов.

**Заполнение массива и тесты:**

Чтобы заполнить массив, нужно обратиться к каждой ячейке через индекс. Такая форма записи громоздкая. В Java есть укороченная форма записи массива. Ее можно использовать, когда нам заранее известны значения массива. Давайте опишем ее шаблоном.



В этом случае мы не указываем количество ячеек массива. Java сама вычисляет размер массива по указанным значениям.

Также для тестов массивов работает Метод assertArrayEquals. Внутри метод assertArrayEquals запускает цикл, в котором сравнивает ячейки массивов result и expected.

**Обход массива с последнего элемента:**

Имеется два способа обойти массив прямой и обратный.

1) Обратный.

Более классический способ, при котором первый элемент цикла будет считаться последним элементом массива, соответственно будем идти в обратном порядке.

Изображение выглядит как текст

Автоматически созданное описание

2) Прямой.

Данный метод предполагает, что обход по циклу будет проводиться с 0 или с индекса, который требуется по заданию и заканчиваться последним нужным. Основная идея заключается в том, чтобы найти закономерность нахождения последних элементов, в конченом итоге получается:



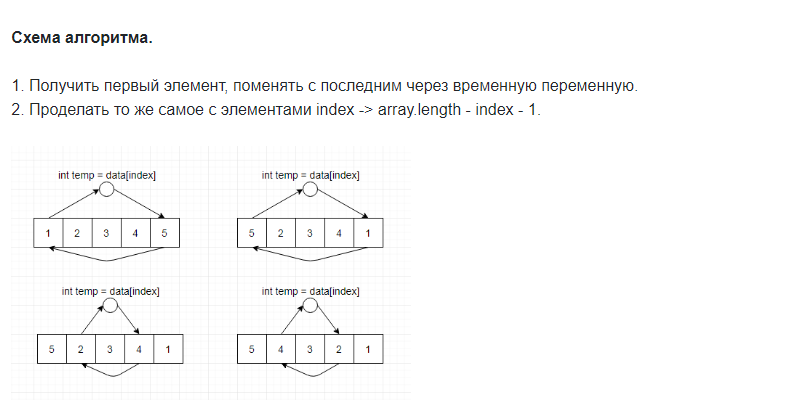
Соответственно получаем:

Изображение выглядит как текст

Автоматически созданное описание

**Типовые задания:**

1. **Перевернуть массив**.

Алгоритм:

1. **Массив заполнен true или false**.

Метод должен проверить, что все элементы в массиве являются либо true либо false. Решение задачи сводится к проверке всех элементов с эталонным. Эталонным можно взять любой элемент в массиве. Лучше взять элемент с индексом 0 (метод работает в большинстве задач). Например:

{true, true, true} - вернет true; массив однородный, он заполнен true.

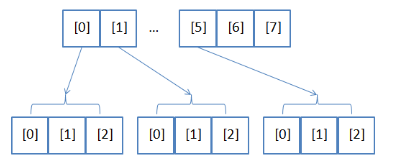
{true, false, true} - вернет false; массив не однородный, он заполнен и true и false.

{false, false, false} - вернет true; массив однородный, он заполнен false.

**Двумерный массив:**

В массиве в качестве значения можно записывать любые типы данных, в том числе в ячейку массива можно записать другой массив.

Можно создать массив, где элементами массива будут другие массивы.

****

Опишем шаблон создания такого массива:

****

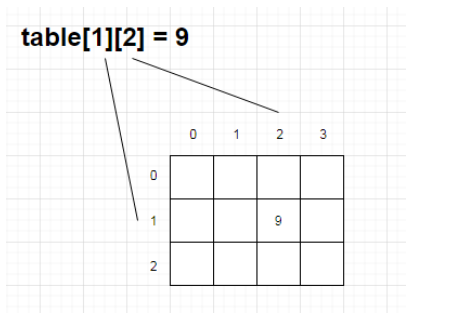
Создадим массив на 3 ячейки, где в каждой ячейке будет массив на 4 ячейки, содержащий числа int.

****

Чтобы обратиться к конечному элементу, то есть получить или заполнить ячейку, нужно указать два индекса: первый для внешнего массива, второй - для внутреннего.

****

Визуально двумерный массив можно представить как таблицу. В этом случае строка будет указывать первый индекс, а столбец второй.



Также имеется возможность сразу создать массив и заполнить его, делается это так:



Чтобы вывети длину двумерного массива, перед командной припиской length, нужно указать индекс, тогда мы получим длину массива по определенному индексу, также позволит вывести длину каждого подобного вложенного массива с помощью цикла.



Пример:

Изображение выглядит как текст

Автоматически созданное описание

String и StringBuilder:

**String:**

<https://javarush.ru/groups/posts/2347-klass-string-v-java>.

**StringBuilder:**

<https://javarush.ru/quests/lectures/questmultithreading.level02.lecture08>.

<https://javarush.ru/quests/lectures/questsyntaxpro.level09.lecture06>.

1. ООП.
   1. Классы и объекты.

Что же такое объект? Это элемент системы с присущими ему свойствами.

Чем нам программистам интересны объекты? А тем, что они позволяют абстрагироваться или отвлечься от внутреннего устройства объекта и сфокусироваться на свойствах и возможностях этого объекта.

Класс — это описание объекта. Когда нам нужно, мы просим виртуальную машину сделать для нас объект по описанию, который мы делаем в классе. В первом уровне мы создавали классы, но не создавали объектов. Почему такое возможно?

В языке Java объединены все парадигмы программирования: структурное программирование, ООП и функциональное программирование. Структурное программирование основано на примитивных типах и конструкциях: операторы условий, циклы и массивы. Это программирование решает задачи: поиск, сортировка и преобразование конечных данных. Этот подход использовался в первом модуле курса. Поэтому нам и не было необходимости создавать объекты на основании классов.

Парадигма ООП помогает решить задачу взаимодействия объектов. В свою очередь сам объект внутри использует структурное программирование. В этой главе мы посмотрим, как за счет ООП мы решим задачи расширения проекта.

Функциональное программирование решает аналогичную задачу со структурным подходом, но не используют переменные. В функциональном программировании используются только конечные данные.

**Практическое применение**

Давайте разберемся, что происходит в этой строчке кода:



Здесь мы создаем переменную типа Cat с именем peppy. Далее мы записываем в эту переменную объект типа Cat. Важный момент.

**Оператор new резервирует кусок памяти и записывает в нее данные объекта класса Cat.**

***Пример работы с объектом класса Student:***

Давайте сделаем класс Student. Сразу добавим в него особый метод main и сделаем объект класса Student. Присвоим переменную petya. Давайте научим нашего героя играть мелодию. Для этого создадим метод. Метод должен находиться внутри тела класса. Чтобы вызвать код, написанный внутри метода, нам нужно обратиться к **переменной объекта Student** и через точку написать имя метода.

Вызываем метод music(), используя переменную petya объекта класса Student.

Изображение выглядит как текст

Автоматически созданное описание

Теперь если мы запустим наш код, то в консоли мы увидим "Tra tra tra".

Изображение выглядит как текст

Автоматически созданное описание

При работе с разными входными параметрами добавится несколько дополнительных задач. Термин "входящие параметры" обозначает переменные, которые будут доступны внутри метода и которые нужно проинициализировать при вызове этого метода.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

***Поле объекта:***

Для того, чтобы мы могли использовать память нашего объекта нам нужно создать поля. Термин "поле" обозначает переменную, которая связана с объектом. Все поля мы описываем внутри класса. Локальные переменные мы создаем внутри метода. Для того, чтобы обратиться к полю объекта, нам нужно использовать ключевое слово this. Это слово указывает, что нужно вывести содержимое поля.

**this.** - дальше указываем имя поля, которое нужно вывести.

Пример использования:

Изображение выглядит как текст

Автоматически созданное описание

***Конструктор:***

Конструктор — это специальный метод, который вызывается при создании нового объекта. Бывает неудобно инициализировать все переменные класса при создании его экземпляра и писать для этого отдельные методы. По этой причине бывает проще, чтобы какие-то значения были бы созданы по умолчанию при создании объекта. По сути, конструктор нужен для автоматической инициализации переменных. Конструкторы вызываются при создании нового объекта нашего класса. Имя конструктора совпадает с именем класса, в том числе и соблюдая регистр, при этом синтаксически конструктор похож на метод без возвращаемого значения.

Изображение выглядит как текст

Автоматически созданное описание

То, что стоит после оператора new и есть вызов конструктора класса Computer, и в данном случае после имени класса мы ставим круглые скобки. Это подчеркивает, что конструктор — это метод. Однако сразу возникает вопрос – что мы можем вызывать, если никаких конструкторов в классе у нас нет. Связано это со следующим:

- Если в классе не определенно ни одного конструктора, то для этого класса создается конструктор без параметров, или как его еще называют – **дефолтный конструктор или конструктор по умолчанию**. Для создания объекта Computer мы используем выражение new Computer(). При этом оператор new выделяет память для нашего объекта computer типа Computer. На следующем этапе происходит вызов конструктора по умолчанию, который не принимает никаких параметров. После выполнения этого выражения в памяти будет выделен участок памяти, где будут храниться все данные нашего объекта. А в переменную computer будет записана ссылка на созданный объект.

У нас есть возможность создать дефолтный конструктор в явном виде, и тогда уже не будет вопросов откуда мы его вызываем. Тогда наш класс будет иметь вид:

Изображение выглядит как текст

Автоматически созданное описание

Давайте добавим в наш класс несколько полей с разными типами: boolean, int и String. Класс будет выглядеть следующим образом:

Изображение выглядит как текст

Автоматически созданное описание

Добавим метод, который будет выводить в консоль содержимое наших полей:

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Мы видим что-то необычное – мы при создании объекта не указывали никаких значений, однако содержимое полей все равно выводится, т.е. им присваиваются некоторые значения. Это еще одна особенность создания объектов – если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Т.е. для числовых типов это значение 0, переменных типа boolean – false, а для типа String и других классов – null ссылка.

Также в дефолтном конструкторе мы можем определить поведение, которое будет инициализировать поля другими значениями, которые мы определим заранее:

Изображение выглядит как текст

Автоматически созданное описание

и вывод в консоль содержимого полей созданного таким конструктором объекта будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

Однако такого стоит избегать и не закладывать в дефолтный конструктор иную логику, отличную от инициализации полей значениями по умолчанию, поскольку потребители Вашего кода могут столкнуться с проблемой, что в поля будут записываться какие-то «магические» значения, которые они нигде не задавали.

Вместо этого подхода мы создадим конструктор, который будет принимать 3 параметра и записывать их в наши поля. Выглядит это следующим образом:

Изображение выглядит как текст

Автоматически созданное описание

Создавать объект мы будем следующим образом:



***Статические и не статические методы:***

В Java ключевым словом static помечают члены (поля и методы), которые принадлежат классу, а не экземпляру этого класса.

Таким образом, сколько бы объектов мы не создавали, всегда будет создан один член класса, доступный для использования всеми экземплярами класса.

Ключевое слово static может применяться:

1. Переменные (поля) класса;

2. Методы;

3. Импорты (это мы рассматривали ранее);

4. Вложенные классы.

Когда стоит использовать статические методы:

1. Для доступа или управления статическими переменными и другими статическими методами, которые не зависят от объектов;

2. Для служебных, вспомогательных классов и интерфейсов, поскольку не требуют создания объекта и соответственно, обеспечивают большую производительность;

3. Когда методу требуется доступ лишь к статическим полям класса.

В завершении стоит подытожить и подчеркнуть особенности статических методов и нестатических:

1. Статические методы нельзя переопределять. Про переопределение методов будет позже в курсе, сейчас просто запомните, что статические методы не имеют возможности быть переопределенными;

2. Статические методы не могут использовать ключевые слова this (ссылка на текущий объект) или super (ссылка на объект суперкласса);

3. Статические методы могут обращаться ко всем статическим переменным и другим статическим методам. Однако они не могут обращаться к переменным класса и нестатическим методам. Для этого им нужны ссылки на объект.

4. Нестатические методы могут обращаться непосредственно как к методам класса, так и к переменным класса;

5. Нестатические методы могут обращаться к статическим переменным и статическим методам;

***Ответвление:***

**Перегрузка:**

В Java есть механизм позволяющий создавать методы с одинаковыми именами, но с разными входными параметрами.

Например. Создадим метод, складывающий два числа и три числа.

Изображение выглядит как текст, водоплавающая птица, снимок экрана

Автоматически созданное описание

 Методы add - являются перегруженными. Это механизм позволяет сделать более удобным использование объекта Calculator. Вы запоминаете, что в классе есть метод сложения и по аналогии начинаете его использовать для сложения двух и трех чисел. Если бы такого механизма не было бы Вам нужно было бы создавать метод с другим именем.

Например, addForThree.

Давайте упростим наш код. Мы можем вызвать метод add c двумя аргументами в методе add c тремя аргументами.

Изображение выглядит как текст

Автоматически созданное описание

**Локальные переменные и поля:**

В самых первых уроках мы познакомились с понятием переменной. Переменная — это контейнер для нашей информации. Когда мы коснулись темы объектов, то появилось новое понятие - поле объекта. В чем отличие?

Поле объекта — это тоже переменная, которая связана с этим объектом. Пока мы работаем с объектом, мы можем работать и с этим полем.

Локальная переменная принадлежит методу. Работать с этой переменной мы можем только внутри этого метода. Как отличить локальную переменную от поля объекта?

— Локальная переменная объявляется внутри метода.

— Поле объекта объявляется в области класса, она не может объявляться в методе.

**Зона видимости переменных(scope):**

Переменная — это именованная область памяти. Имя переменной должно быть уникальным.

Рассмотрим 4 случая.

1. Попытка создать две переменные с одинаковым именем внутри метода.

Изображение выглядит как текст

Автоматически созданное описание

IDEA не компилирует такой код, потому что в методе есть две переменные с одинаковым именем.

Термин "scope" означает область, где переменная может быть использована. Обе переменные могут использоваться внутри метода. "Scope" переводится с английского как "зона видимости".

2. Разные методы имеют одинаковые переменные.

Изображение выглядит как текст

Автоматически созданное описание

В классе Reduce созданы две переменные с одинаковым именем. Этот код можно скомпилировать без ошибок. Почему нет ошибки? Все дело в scope. Переменные max доступны внутри своего метода. Зона их видимости разная. Компилятор понимает, какую переменную использовать в каждом случае.

3. Блоки if-else содержат одинаковые переменные.

Зона видимости определяется расположением фигурных скобок.

Изображение выглядит как текст

Автоматически созданное описание

Первая переменная max доступна внутри блока if () {}. Вторая переменная доступна внутри блока else {}. Зоны видимости этих переменных разные. Компилятор понимает, какую переменную использовать.

4. Поле класса и метод имеют одинаковые переменные.

Изображение выглядит как текст

Автоматически созданное описание

Поле класса доступно во всех методах класса. Переменная внутри метода доступна только внутри этого метода. Зона видимости у них разная, но есть общие области! Внутри метода min компилятор будет использовать локальную переменную, а не переменную поля.

Не пишите такой код. Он запутанный. Поля и переменные внутри метода называйте по-разному.

* 1. Наследование.

Основа ООП – создание объектов. Задача механизма наследования помочь нам создавать объекты.

Теперь давайте посмотрим на Java. Базовым элементом в Java является объект. Язык Java спроектирован таким образом, что в основе любого объекта лежит класс java.lang.Object. В этом классе есть методы, которые помогают унифицировать или стандартизовать работу объектов в вашей программе. Какие манипуляции обычно выполняет программист с объектами? Он их сравнивает, копирует, выводит на консоль. Все эти действия описываются через методы класса java.lang.Object. А через механизм наследования они становятся доступны каждому объекту.

**Элементы наследования**

Давайте перечислим элементы, которые есть в классе и отметим, какие из них наследуются.

1. Конструкторы. Этот элемент не наследуются, но он активно принимает участие в наследовании.

2. Поля класса. Они могут наследоваться и использоваться в дочернем классе, но я настоятельно не рекомендую это делать. В разделе "Инкапсуляция" мы рассмотрим тему использования полей через наследование.

3. Методы. Самый интересный для нас элемент в теме наследования. Они-то и делают основную работу в наследовании.

4. Множественное наследование

Java был создан на основании опыта C++. Ряд особенностей языка С++ был исключен из Java, чтобы упростить его применение. Это исключение коснулось множественного наследования. В Java класс может наследовать только один другой класс. Это ограничение можно обойти за счет другого механизма - полиморфизма.

**Пример**

Рассмотрим классический пример с кошечками.

Изображение выглядит как текст

Автоматически созданное описание

Класс имеет один метод. Теперь давайте сделаем класс ToyCat на основании класса Cat.

Изображение выглядит как текст

Автоматически созданное описание

Чтобы наследовать один класс от другого, нужно после имени класса написать ключевое слово **extends** и далее указать, какой класс нужно наследовать.



Обратите внимание в классе ToyCat появился метод canBeWashByWashMachine. Живую кошку стирать в машине не стоит, поэтому этот метод есть только у игрушечной кошки.

Теперь давайте создадим объекты.

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

Объект toy имеет два метода:

1) canPurr, который наследовался от Cat.

2) canBeWashByWashMachine, который объявлен в самом классе ToyCat.

Это возможно за счет наследования. Класс ToyCat унаследовал метод purr() от родителя - класса Cat.

**Иерархия классов и super**

Иногда в программе при наследовании в подклассах (наследниках, потомках и т.п.) требуется получить ссылку непосредственно на его суперкласс (класс родитель, предок и т.п.). В Java для этих целей используется ключевое слово **super**.

По аналогии с ключевым словом *this*, super может быть использован для следующих целей:

1. Для вызова конструктора суперкласса:

super(список\_параметров\_конструктора)

2. Для вызова метода суперкласса:

super.название\_метода(список\_ параметров \_метода)

Если в иерархии классов требуется передать параметры конструктору суперкласса, то все подклассы должны передавать эти параметры вверх по иерархии. Таким образом, из конструктора подкласса надо вызвать конструктор суперкласса с помощью super(). Когда метод super() вызывается из подкласса, то при этом вызывается конструктор его непосредственного суперкласса. Это утверждение также справедливо для многоуровневой иерархии.

Вызов метода *super()* должен быть всегда в первом операторе, который выполняется в теле конструктора подкласса.

**Пример:**

В следующем примере из конструктора класса Child мы вызываем конструктор класса Parent с помощью super() и таким образом мы передаем необходимые значения в конструктор суперкласса:

Класс Parent

Изображение выглядит как текст

Автоматически созданное описание

Класс Child

Изображение выглядит как текст

Автоматически созданное описание

Если в конструкторе наследника нет явного вызова super(), как например во втором конструкторе класса Child, то JVM подставляет этот конструктор первой строкой самостоятельно. Т.е. можно поставить так:

Изображение выглядит как текст

Автоматически созданное описание

Из этого следует, что суперкласс должен иметь конструктор без параметров, иначе возникнет ошибка компиляции.

Также важно понимать, что в иерархии классов конструкторы вызываются в порядке наследования, начиная с суперкласса и заканчивая подклассом. Более того, поскольку super() должен быть первым оператором, выполняемым в конструкторе подкласса, этот порядок неизменным, независимо от того, используется ли форма super().

**Переопределение метода**

Переопределение позволяет задать новое поведение существующего метода. Механизм переопределения (overriding) существует только при условии наследования.

Пример:

Класс TextReport

Изображение выглядит как текст

Автоматически созданное описание

Класс для вывода:

Изображение выглядит как текст

Автоматически созданное описание

Вывод:

Изображение выглядит как текст

Автоматически созданное описание

Создает наследник класса TextReport HtmlReport:

Изображение выглядит как текст

Автоматически созданное описание

Вызов:

Изображение выглядит как текст

Автоматически созданное описание

На консоли ничего не изменилось. Вывод остался прежним. Потому что мы используем метод generate, который определяем в классе TextReport. Он доступен в классе HtmlReport через механизм наследования.

Добавим такой же метод в класс HtmlReport, но изменим внутри него код:

Изображение выглядит как текст

Автоматически созданное описание

Вывод:

Изображение выглядит как текст

Автоматически созданное описание

Теперь мы видим, что вывод стал другой. Это произошло потому, что мы переопределили поведение метода generate.

**Аннотации**

Аннотация — это механизм, позволяющий произвести дополнительные действия с вашим кодом. Можно выделить два действия: анализ кода и генерация нового кода на основании вашего. То есть сама аннотация никак не влияет на ваш код, ее используют другие программы.

Поговорим про аннотацию @Override. Эта аннотация используется только для методов. Она позволяется подтвердить работу механизма переопределения.

1) Аннотации могут объявляться над классом, полем, методом.

2) Объявление аннотации выглядит следующем образом. Сначала пишется символ @, а после него имя.

**Пример**

Представьте, Вы опечатались в имени метода, ошибку эту вы не заметили. Далее Вы запустили программу и увидели, что программа работает неверно. Через некоторое время вы найдете причину поломки. Чтобы не пришлось искать долго, можно использовать аннотацию @Override. Давайте теперь добавим на сломанный метод аннотацию @Override.

IDEA сразу сигнализирует нам, что в коде какая-то ошибка.

Изображение выглядит как текст

Автоматически созданное описание

Давайте проигнорируем эту ошибку и попробуем запустить код.

Изображение выглядит как текст

Автоматически созданное описание

В этом случае проект запустить нельзя. То есть IDEA позволяет с помощью @Override проверить правильность вашего кода на стадии его компиляции. Этим уменьшает возникновение непредвиденных ошибок.

* 1. Инкапсуляция.
     1. Общие сведения.

В этом разделе мы познакомимся с первым механизмом ООП — это инкапсуляция или закрытие. Этот механизм позволяет скрыть реализацию кода класса от пользователя этого класса. А зачем что-то скрывать? Пользователь класса может записать данные или вызвать методы, которые повлекут ошибку в работе этого класса. Механизм инкапсуляции позволяет запретить обращаться к методам или полям.

Для реализации механизма используют модификаторы доступа. **Модификаторов всего 4:** *public, protected, default и private***.**

Модификатор доступа располагается вначале элемента: класса, конструктора, метода, поля класса. К локальным переменным модификаторы не применяются.

*public* - элемент, обозначенный этим модификатором доступа доступен везде в программе.

*protected* - элемент, обозначенный этим модификатором доступа доступен внутри пакета и в наследниках.

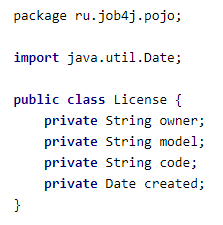
*default* - элемент, обозначенный этим модификатором доступа доступен внутри пакета.

*private* - элемент, обозначенный этим модификатором доступа доступен внутри класса.

**Модель данных**

В программировании есть понятие модель данных. Модель данных — это объект, который используется, как контейнер для примитивных или ссылочных типов данных. Например, паспорт. Паспорт содержит информацию о человеке. Сам паспорт ничего делать не умеет, но он нужен человеку для выполнение других действий.

Давайте рассмотрим пример модели данных - автомобильный техпаспорт.



Этот класс пока бесполезный. Мы не можем в него загрузить и прочитать данных. Чтобы это стало возможным на каждое поле мы добавляем методы get и set. Код этих методов набирать не надо. В IDEA есть механизм позволяющий сгенерировать такие методы. Эти методы еще называются аксессоры ( От анг. аccess - доступ).

Результат:

Изображение выглядит как текст

Автоматически созданное описание

Теперь это класс можно полноценно использовать.

**To String**

Данный метод поможет нам выводить объекты классов в виде строкового представления. Для того, чтобы код можно было переиспользовать и не думать о том, как выводить строковое представление объектов служит метод *toString()*, который определен в классе  Object, наследниками которого по умолчанию являются все классы в Java. Для того чтобы этот метод использовать в своих классах, необходимо использовать механизм переопределения. Это можно выполнить средствами Inteliji IDEA. Чтобы в будущем, при изменении данных выводить всех их нужно просто переопределить метод *toString()*.

* + 1. Форматирование данных. Методы format() и printf().

Познакомимся с форматированием строк в Java. За создание форматированного текстового вывода отвечает класс **java.util.Formatter**. Метод **format()** этого класса позволяет выводить различные данные - строки, числа, даты, время в желаемом виде, т.е. преобразует данные в тот формат, который нам нужен. Класс **Formatter** подробно разбирать не будем, так как различных видов форматирования и их особенностей очень много, а изучение форматирования продолжим на методе **format()**, также определенном и в классе String. Он позволяет удобно работать с форматированием и вызывать метод **format()** у строки.

Если мы посмотрим на реализацию метода **format()** класса String, то увидим, что внутри он создает объект класса **Formatter**, вызывает у него метод **format()** с заданными нами аргументами, и после возвращает строковое представление объекта **Formatter**:

Изображение выглядит как текст

Автоматически созданное описание

Методы **format()** специально определены в классе String и еще, например, в классах **PrintStream** и **PrintWriter**, т.е. в классах, которые работают с выводом строк, чтобы не нужно было отдельно работать с классом **Formatter** — это удобно и получается меньше кода. А в классах **PrintStream** и **PrintWriter** определён еще и метод **printf()**, который результат форматирования сразу выводит в консоль аналогично методам **print()** или **println()**.

Сразу стоит отметить различия методов **format(**) и **printf()**. Метод **format()** выполняет форматирование и возвращает новую строку. Её можно сохранить и использовать в дальнейшем. Метод **printf()** выполняет форматирование и выводит данные в консоль без сохранения. Для более глубокого понимания процесса в этом уроке вывод делается через цепочку методов (println + format). Все эти методы можно переписать с помощью **printf()** и получить такой же результат за исключением отсутствия у метода **printf()** перевода курсора на новую строку.

Форматирование производится по шаблону - в метод **format()** передаётся строка с указанными местами для подстановки аргументов, а также сами значения, которые необходимо вставить. Форматирование применяется в шаблонных текстах или строках, где требуется подстановка в какие-то места шаблона различных значений, принимаемых в виде параметров.

Разберем пример:



В первый параметр метода **format()** передаётся строка с указанным спецификатором **%s**, это некий шаблон. Спецификатор **%s** здесь означает тип аргумента, который будет передан на форматирование (подстановку). Он означает любой тип параметра, который будет приведён к строке. Это самый универсальный вариант подстановки значения в параметр. Вторым аргументом в данной строке передано значение для подстановки - строка **name**, которую принимает метод **getGreeting()**. В результате метод **format()** создает строку по указанному шаблону с подставленным значением.

Вывод:



Параметров в шаблоне мы можем указать несколько, например:



Вывод:



**Спецификаторы.**

В последнем примере мы применяли только спецификатор **%s**. Данный спецификатор хоть и можно применить к любому типу, но вывод некоторых типов может быть не тем, который мы хотим получить, так как **%s** выводит строковое представление аргумента, а наша цель - получить вывод данных именно в том формате, который мы хотим. Поэтому под каждый тип данных лучше указывать спецификатор именно того типа, который нужно получить.

Пример:



Мы выводим на печать отформатированные в удобном формате дату и время.

Вывод:



Задают вид даты и времени здесь спецификаторы **%t** с флагами **F** и **T**. Флаг **F** выводит дату в формате ISO 8601 - ГГГГ-ММ-ДД, а флаг **T** выводит время в формате ЧЧ:ММ:СС. Флагов имеется несколько видов. С этими и другими флагами ко всем спецификаторам можно ознакомиться в к классу **Formatter**.

**Специальные настройки.**

С помощью них можно указывать позицию аргумента в списке, устанавливать специальные флаги, минимальное количество выводимых символов и ограничение количества символов.

Синтаксис записи всех настроек выглядит так:



Данная запись называется инструкцией.

- С помощью **индекса аргумента** можно указать позицию в списке аргументов (в виде целого числа). Например, на первый аргумент можно сослаться в виде 1$, а на четвертый - 4$. Если это явно не указано, то все аргументы должны быть переданы в метод в порядке объявления в шаблоне ссылок на них.

- **Флаги** — это отдельные символы для изменения результата форматирования. Например, флаг "+" добавит знак"+" к получившемуся результату, а флаг "," выведет результат с разделителем в виде запятой у тысяч.

- **Ширина** определяет количество выводимых символов. Задаётся целым положительным числом. Если перед ним поставить 0, то если общее количество символов строки вывода будет менее этого значения, разница будет заполнена нулями.

- **Точность** мы уже разобрали выше. Через указание точки и целого числа можно задать выводимое после точки количество символов.

Пример с применением всех 4 видов настроек:



Вывод:



[Formatter](https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Formatter.html).

* + 1. Record.

**Record** дает возможность записать стандартную модель данных в виде одной строки, то есть с помощью **record** можно избавиться от шаблонного кода, который приходится писать в каждой модели данных.

**Record** — это **неизменяемая** модель данных, в которой:

- Поля, переданные в параметрах, являются **private** и **final** по умолчанию.

- Код имеет канонический конструктор, параметры которого — это все поля данного класса.

- Доступ к полям осуществляется через методы типа имя\_поля(). Тот же геттер, только без слова **get** в имени метода. Методов типа **set** класс **record** не имеет, так как все поля объявлены как **final**, поэтому **record** является неизменяемым.

- Определены методы **equals()** и **hashcode()**, включающие все поля класса.

- Определен шаблонный метод **toString()** для вывода полей класса на печать.

Есть небольшая разница в каком виде выводится объект на печать, потому что в **record** для вывода объекта на печать используется **StringJoiner**, а в обычном классе мы используем стандартную реализацию **toString()**, которую предлагает IDEA. Если требуется точное совпадение, достаточно явно переопределить метод **toString()** в **record** нужной реализацией:

Пример:

Изображение выглядит как текст

Автоматически созданное описание

Точно так же можно переопределять в **record** свои реализации конструкторов и методов, если это необходимо.

**Record** удобно применять, когда у модели могут добавляться или удаляться поля. Достаточно просто вписать новое поле в запись **record** и в класс автоматически будут внесены изменения. В случае же с обычным классом нужно будет вручную вписывать поле в модель, менять конструктор, методы **equals()**, **hashcode()**, **toString()**, добавлять или удалять геттеры.

**Особенности.**

**Record** имеет некоторые особенности, перечисленные ниже:

- В **record** можно добавлять *только* статические переменные (как константы, так и обычные статические переменные). Например:



- В **record** можно добавлять свои методы, как нестатические, так и статические. Например:

Изображение выглядит как текст

Автоматически созданное описание

- **Record** является **final**, поэтому наследование этого класса невозможно. Сам **record** наследовать другие классы не может, так он унаследован от **java.lang.Record**. Но **record** может реализовывать интерфейсы

Изображение выглядит как текст

Автоматически созданное описание

**Конструкторы.**

Отдельно стоит обсудить конструкторы **record**.

- По умолчанию **record** имеет канонический конструктор — это конструктор, принимающий все поля класса в качестве параметров. Канонический конструктор выглядит как:

Изображение выглядит как текст

Автоматически созданное описание

Такой же конструктор неявно создается в **record** по умолчанию. Если его явно объявить в **record**, то будет использоваться объявленный конструктор вместо канонического. Конструктор по умолчанию при объявлении явного канонического конструктора создан не будет.

- Если нам нужен конструктор с какой-то логикой помимо обычной инициализации полей, можно создать компактный конструктор:

Изображение выглядит как текст

Автоматически созданное описание

Данный конструктор называется компактным, так как в нем поля не инициализируются явным образом. Инициализация проходит неявно перед выполнением команд, находящихся внутри конструктора. Данная инициализация автоматически будет проходить в любом переопределенном конструкторе. Это гарантирует, что все поля будут инициализированы.

Если в **record** уже объявлен компактный конструктор, то явно объявить еще и канонический конструктор в этом классе невозможно, так как при объявлении компактного конструктора автоматически создается конструктор по умолчанию, чтобы провести инициализацию всех полей перед выполнением команд компактного конструктора, следовательно нельзя еще раз объявить в классе такой же конструктор с теми же параметрами.

* + 1. Enum.

**Enum** — это класс-перечисление. Перечисление — это набор констант, связанных между собой по смыслу. Константы здесь — это статические объекты, создающиеся в единственном экземпляре. Обычно перечисление используют для работы со значениями, которые никогда не меняются. Например, дни недели, времена года, карточные масти или даже записи полей шахматной доски. Перечисление можно использовать и для групп констант другого вида. Например, для перечисления в программе статусов работы в автосервисе ("Принят на сервис", "В работе", "Ожидание запчастей", "Готов" и т.д.). Такие статусы никогда не меняются и имеют логическую связь между собой.

**Enum** создается как обычный класс в Java, только нужно выбрать соответствующее значение **enum**. Создадим перечисление статусов автомобиля в автосервисе **Status** и заполним его:

Изображение выглядит как текст

Автоматически созданное описание

Теперь мы можем создавать переменные типа **Status** и применять их. Создадим класс **CarService** и пару переменных из нашего перечисления:

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

Вывод:

Изображение выглядит как текст, оранжевый, темный, закрыть

Автоматически созданное описание

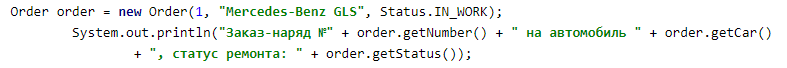
Далее создадим класс заказ-наряд **Order**:

Изображение выглядит как текст

Автоматически созданное описание

Класс **Order** содержит поля номер заказ-наряда, модель автомобиля и статус его ремонта в автосервисе (в виде объекта **Status**). То есть мы можем использовать тип **Status** в объектах как обычное поле класса.

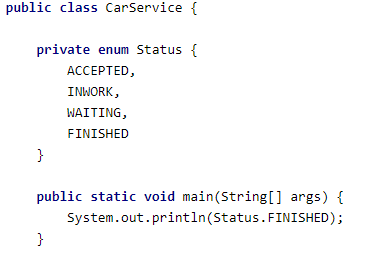
Создадим и выведем объект заказ-наряда **Order** на печать в классе **CarService**:



Вывод:



Перечисление может объявляться отдельно, а может быть объявлено в другом классе в виде внутреннего класса. Например, объявление перечисления **Status** в классе **CarService** выглядело бы так:



Обратите внимание, что модификатор **enum** здесь **private**. Если **enum** вложен в другой класс, он может быть как **public**, так и **private**. Если же **enum** не является внутренним классом, то он может иметь только модификатор **public**.

Перечисление (**enum**) наследуется от **java.lang.Enum**, поэтому он не может наследовать другие классы.

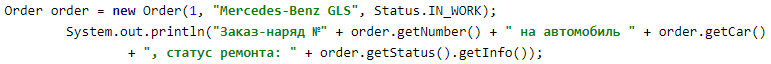
- Перечисление может реализовывать интерфейсы.

- Перечисление может содержать поля, методы и конструкторы. Добавим в перечисление поле **info**, создадим конструктор, принимающий это поле, и геттер, чтобы получать значение поля **info**.

Изображение выглядит как текст

Автоматически созданное описание

Обновим код вывода заказ-наряда в методе **main** класса **CarService**:



Вывод:



**Важно!** В нашем коде конструктор в перечислении неявно имеет модификатор **private**. Других модификаторов конструктор перечисления иметь не может, соответственно, создать объект перечисления мы можем только внутри него самого. Каждый из экземпляров перечисления создаётся только при первом обращении к нему. Каждое значение (константа) нашего перечисления — это объект анонимного класса. Создав, конструктор с полем **info**, мы сразу должны передать этот параметр в каждое значение нашего перечисления, то есть ранее у нас был просто ACCEPTED, а стал



Данная строка "под капотом" выглядит как:



Отсюда следует, что перечисление — это список неизменяемых готовых объектов, которыми можно пользоваться.

Так как каждое значение перечисления — это объект анонимного класса, то в нём можно объявлять свои методы:

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Сначала в перечисление **Status** мы добавили абстрактный метод **getMessage()**, который обязывает каждое значение перечисления переопределить этот метод:



Далее в тело каждого из значений мы добавили по переменной **message** и по методу **getMessage()**, который переопределяет абстрактный метод из **Status**, возвращая переменную **message**. Таким образом, каждое значение — это отдельный объект со своим поведением.

**Методы enum.**

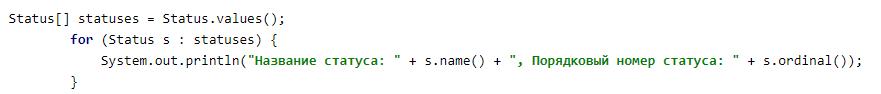
Далее представлены некоторые полезные методы класса **Enum**.

- **name()** - возвращает имя константы (значения) в том же виде, в каком оно объявлено.

- **ordinal()** - возвращает порядковый номер константы (значения), по которому её экземпляр находится в перечислении (нумерация с нуля).

- **values()** - возвращает массив всех элементов перечисления.

Добавим вывод работы этих методов в метод **main()** класса **CarService**:



В следующей строке метод **values()** получает массив, содержащий список значений из класса **Status**:



Вывод:

Изображение выглядит как текст

Автоматически созданное описание

- **valueOf()** - получает объект типа класса перечисления по его строковому представлению.

Изображение выглядит как текст, оранжевый, снимок экрана

Автоматически созданное описание

В данном коде мы получаем объект типа **Status** из одноимённой строки.

Вывод:



Если в классе **Status** нет такого значения (константы), то будет выброшено исключение **IllegalArgumentException**. Имя константы должно полностью совпадать, включая регистр. Например, такая запись вызовет исключение.



Методы **values()** и **valueOf()** не имеют полной реализации в классе **Enum**, они добавляются в программу на этапе её компиляции.

**Switch.**

Также, так как значения **enum** не меняются, то при работе с ними можно использовать оператор **switch**. Добавим в метод **main()**:

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

* 1. Полиморфизм.

Полиморфизм - позволяет достигнуть взаимозаменяемости. **То есть часть программы не имеет жесткой связи и ее можно заменить на другую программу**. Полиморфизм окружает нашу жизнь. Например: компьютер и его отдельные детали или пример из программирования — это ваша программа и база данных. Полиморфизм в Java достигается за счет использования сущности interface.

Интерфейсы обладают особыми свойствами.

1. Не могут хранить состояния. Внутри интерфейса нельзя создать поля для сохранения в них данных.

2. Могут иметь множественное наследование. В java нет множественного наследования относительно классов. Зато в интерфейсах такая возможность есть.

3. Объявление методов без реализации. По сути, это и есть контракт, как должен работать тот или иной объект, чтобы он мог использоваться в другой системе. Например, USB вход. Универсальный интерфейс. С помощью него можно подключить любое устройство. Причем можно как читать данные, так и просто заряжать устройство.

4. Реализация методов. По сути интерфейс — это тот же класс без состояния с возможностью множественного наследования.

Можно заметить, что заменить одну часть программы на другую можно через наследование. Это верно. Но в этом случае мы привязываемся к базовому классу. Представьте, что вы пишете программу для работы с базой данных. В какой-то момент времени вы поняли, что системе требуется еще хранилище в файлах. Вам нужно наследовать базовый класс и переопределять поведение. Хорошо, если базовый класс не загружает начальное состояние в конструкторе, а если он делает запрос в базу данных это добавит дополнительное действие в вашу реализацию.

В случае с интерфейсами мы описываем, какие методы должно иметь хранилище. В любой момент времени мы можем создать новый класс, который будет осуществлять этот контракт.

**Пример легкий:**

Создание:

Изображение выглядит как текст

Автоматически созданное описание

Объявление 2ух методов:

Изображение выглядит как текст

Автоматически созданное описание

Создание класса от интерфейса:

Изображение выглядит как текст

Автоматически созданное описание

**Пример:**

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

Интерфейс содержит определение методов, но не содержит их реализации.

Теперь давайте применим эти интерфейсы к нашим хранилищам.

Изображение выглядит как текст

Автоматически созданное описание

И FileStore.

Изображение выглядит как текст

Автоматически созданное описание

Теперь хранилища не привязаны друг к другу. Осталось отвязать сам сервис.

Изображение выглядит как текст

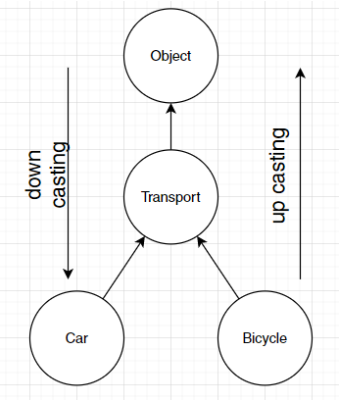
Автоматически созданное описание

Теперь все классы не связаны между собой. Это позволяет заменить одну часть программы на другую без переписывания кода.

**Приведение типов:**

Как мы знаем все классы в Java по умолчанию наследуются от класса java.lang.Object. То есть любой класс в Java обладает такими же свойствами, как и java.lang.Object. Механизм приведение типа позволяет относиться к объекту класса, так же, как и к объекту любого класса в его иерархии.

Рассмотрим это на примере иерархии транспорта.



Создадим классы:

Родительский:

Изображение выглядит как текст

Автоматически созданное описание

Дочерний1:

Изображение выглядит как текст

Автоматически созданное описание

Дочерний2:

Изображение выглядит как текст

Автоматически созданное описание

Давайте создадим класс HierarchyUsage, в котором продемонстрируем понятие приведение типов.

Изображение выглядит как текст

Автоматически созданное описание

Приведение к другим типам:

Изображение выглядит как текст

Автоматически созданное описание

То есть мы приводим объект Car к типу любого класса в его иерархии. Такое приведения типов называется повышающим приведение (up casting). Мы приводим конкретный тип к более общему типу данных. Если посмотреть на схему, то мы как бы поднимаемся вверх. Синтаксис этого приведения типов ничем не отличается от обычной записи объекта в переменную.

Другой пример приведения называется понижающим приведением (down casting). На диаграмме мы опускаемся вниз.

Изображение выглядит как текст

Автоматически созданное описание

Понижающего приведения в коде лучше избегать, по той причине, что код становится не стабильным и нам приходится догадываться, к какому типу данных нужно сделать приведение. Так же при понижающем приведении используется дополнительный синтаксис. Перед переменной нужно указать тип данных, к которому нужно привести. Этот тип указывается в обычных скобках.

Пример:

В чем же еще практическая польза всего приведения. А в том, что все те классы, что мы создали переписать на интерфейс Animal, в который мы вынесем метод, который будет доступен всем животным, однако для каждого он будет иметь собственную реализацию.

**Интерфейс Animal:**

**Изображение выглядит как текст

Автоматически созданное описание**

**Класс Cow:**

**Изображение выглядит как текст

Автоматически созданное описание**

**Класс Goose:**

**Изображение выглядит как текст

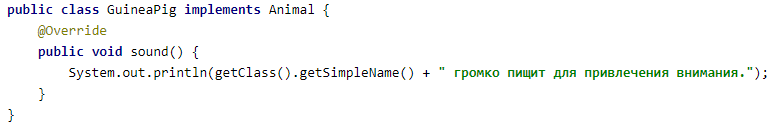
Автоматически созданное описание**

**Класс Dog:**

**Изображение выглядит как текст

Автоматически созданное описание**

**Класс GuineaPig:**

****

И вот в таком виде мы можем повторно использовать код по выводу массива объектов с его методами в консоль:

Изображение выглядит как текст

Автоматически созданное описание

Вывод:

Изображение выглядит как текст

Автоматически созданное описание

Т.е. мы создавали объекты разных классов, при этом приводя все их к общему для каждого из этих классов типу Animal. В таком виде мы их уже можем "сложить" в общий массив объектов типа Animal, и нам будет доступен общий для них метод sound(), однако для каждого класса он будет иметь свое поведение, определенное в каждом классе. Если бы не было приведения типов - уже такой "фокус" мы бы провернуть не смогли и, к сожалению, были бы сильно ограничены в средствах.

Приведение типов - очень важная составляющая реализации полиморфизма в Java. Очень важно разобраться в этом вопросе, в своем коде предпочтение отдавать использованию общих типов, которые определяются в интерфейсе, а не реализациями этих интерфейсов. В таком случае наш код становится более гибким, поскольку мы в любой момент можем заменить реализацию интерфейса, не нарушив общую структуру нашего кода.

**Тестирование:**

Тестирование можно разбить на три группы: *тестирование черным ящиком*, *тестирование серым ящиком* и *тестирование прозрачным (стеклянным) ящиком*.

Группы отличаются друг от друга степенью доступности содержимого кода.

1. В черном тестировании мы не знаем, что находится внутри. Создание теста делаем на основании документации.

2. В сером тестировании часть кода нам доступна, но не вся. Теперь мы можем взять документацию и дополнительно посмотреть, что происходит внутри.

3. В прозрачном тестировании нам доступен весь код.

Доступность кода позволяет нам написать тесты, которые будут затрагивать каждую строчку нашего кода. Таким образом мы гарантируем, что программа будет работать правильно.

* 1. Шаблон проектирования.

Термин шаблон проектирование означает схему решения часто возникающей задачи проектирования. Чтобы не тратить время на решение задачи по проектированию мы используем готовые решения. Все шаблоны разбиты на группы. Группы определяют общую задачу, которую решает этот шаблон.

**Стратегия (Strategy)** - шаблон относится к поведенческим шаблонам. Он позволяет изменить алгоритм работы программы в запущенной программе.

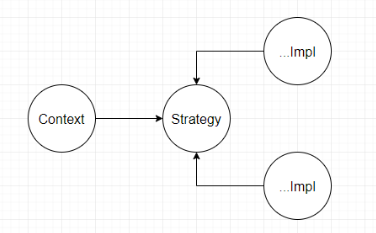
Нам нужны три базовых элемента: интерфейс Strategy, класс Context, класс StrategyImpl.

1. Класс Context - описывает основное поведение программы. Он использует действия интерфейса Strategy.

2. Интерфейс Strategy - описывает обобщенное действие.

3. Класс StrategyImpl - реализует конкретное действие системы. Этот класс реализует интерфейс Strategy.

Диаграмма:

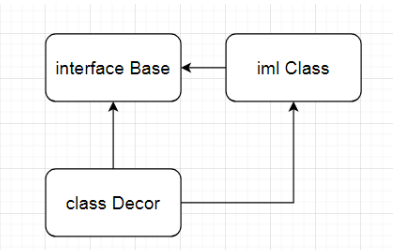


**Декоратор (Decorator)** ­ смысл Декоратора заключается в добавлении нового поведения в уже существующее поведение.

Base - интерфейс, описывающий поведение.

Class - ­класс, реализующий поведение.

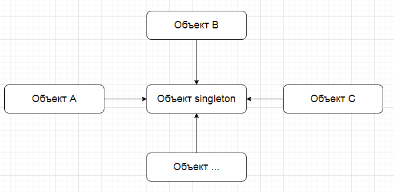
Decor ­ - класс, реализующий поведение и зависимый от другой реализации Base.



**Singleton -**  используется для создания класса, который может содержать только один экземпляр объекта в виртуальной машине. Шаблон singleton применяют в проектах, где используются внешние ресурсы: файловая система, база данных, связь с другими приложениями. В таких проектах доступ к ресурсу нужно ограничить использованием одного объекта на виртуальную машину. Это позволяет экономить ресурсы системы.

**Важно:** про шаблон singleton часто спрашивают на собеседовании. Поэтому его нужно выучить.

Диаграмма шаблона.



Объекты А, В, С используют один и тот же объект singleton.

**Реализация.**

Рассмотрим ситуацию, что в нашей системе потребовалось ввести протоколирование действий системы.

Изображение выглядит как текст

Автоматически созданное описание

Чтобы начать использовать этот класс, нам нужно создать объект этого класса. Но в этом случае каждый объект класса Log4File будет содержать свои данные.

Чтобы такой проблемы не было нам нужно.

1. Создать явно конструктор и указать у него модификатор private. Таким образом, никто не сможет создать объект этого класса.

2. Так же нужно запретить наследование этого класса. Для этого используем ключевое слово final.

Изображение выглядит как текст

Автоматически созданное описание

3. Вы спросите, а какой же тогда смысл в этом классе, если никто не может использовать его методы? На самом деле мы можем создать объект этого класса внутри самого класса. Остается один момент, а как дать ссылку на этот объект? У нас есть ключевое слово static, его можно использовать для полей класса. Если поле обозначено static, то оно привязано к классу, а не к объекту. Это позволяет сохранить ссылку на объект класса Log4File. Так же добавим public static метод, чтобы другие классы могли получить ссылку на объект Log4File.

Доработаем код.

Изображение выглядит как текст

Автоматически созданное описание

Теперь объект класс Log4File может быть только один на всю программу.

Чтобы получить объект класса Log4File мы используем метод Log4File.getInstance().

Изображение выглядит как текст

Автоматически созданное описание

При первом вызове метод getInstance() программа проверяет заполнено ли поле instance.

Если оно не заполнено, то мы создаем объект класса Log4File и записываем его в статическое поле.

* 1. Исключения.

В программировании есть термин - программа завершилась с исключительной ситуацией или в программе упал Exception. Это значит, что в процессе выполнения программы виртуальная машина не смогла выполнить инструкции, написанные в вашей программе. Exception in thread "main" указывает на то, что в вашей программе закралась ошибка. Компилятор и виртуальная машина не знает, как ее разрешить и сообщает об этом клиенту программы.

Если программа работает некорректно, однако не выводится ошибка самой программой (например, если программа по ошибке является бесконечной), то можно использовать оператор **throw** с передачей этому оператору объекта типа java.util.Exception.

Пример:

Изображение выглядит как текст

Автоматически созданное описание

Вывод будет:

Изображение выглядит как текст

Автоматически созданное описание

В Java есть встроенные классы, которые описывают исключительные ситуации. Старайтесь не создавать свои исключения, а использовать встроенные. Это позволит другим программистам быстрее понять причину возникновения проблемы. В конструкторе исключения нужно обязательно указать причину возникновения исключения. Никогда не оставляйте текст пустым. Все эти классы наследуются от класса *java.lang.RuntimeException.* Исключения этого типа имеют особенность. Эти исключения не обрабатывают.

1) *java.lang.IllegalArgumentException* **-** метод вызывается с некорректными параметрами**.**  
2) *java.lang.IllegalStateException* **-** метод вызывается с объекта в не корректном состоянии**.**  
3) *java.lang.NullPointerException* **-** методы вызывается у переменной, которая инициализирована null ссылкой.

**Пользовательские исключения**

Все исключения, кроме *java.lang.RuntimeException* (см. выше) должны быть обработаны. Все исключения (проверяемые и непроверяемые) в основе иерархии наследуются от класса java.lang.Exception. Cделаем наследника UserInputException.

Изображение выглядит как текст

Автоматически созданное описание

Класс java.lang.Exception имеет конструктор по умолчанию. Рекомендуется всегда добавлять конструктор с текстом. При использовании вашего исключения нужно будет добавить детальную информацию о возникновении исключения, а не просто получить пустой вывод.

Предположим, что мы разрабатываем программу для работы с базой данных.

Изображение выглядит как текст

Автоматически созданное описание

В коде мы делаем проверку, что пользователь ввел не пустой адрес к базе.

Если вы вставите такой код в IDEA, то увидите, что IDEA подсвечивает ошибку.

Изображение выглядит как текст

Автоматически созданное описание

**Unhandled exception** - необработанное исключение. Если мы используем в коде классы не имеющие в иерархии наследования java.langRuntimeException, то их нужно обрабатывать.

Существует два способа обработки исключений.

**1) Объявления исключений в сигнатуре метода.**

В Java есть ключевое слово **throws.** Его можно добавить после параметров метода и указать, какие прерывания могут случиться с этим методом.

Изображение выглядит как текст

Автоматически созданное описание

Через запятую можно указать несколько исключений. В этом случае мы передаем ответственность за исключение коду, который будет использовать наш метод. В нашем случае это метод main. C методом main мы можем поступить аналогичным способом и добавить в него перебрасывания исключений выше.

Изображение выглядит как текст

Автоматически созданное описание

**2) Конструкция try-catch.**

Эта конструкция позволяет выполнить методы, которые могут прерваться с исключением и обработать эти исключения.

 В методе main выполним метод load в блоке try-catch.

Изображение выглядит как текст, человек, снимок экрана

Автоматически созданное описание

Мы убрали из сигнатуры объявление throws и обернули метод load в конструкцию try-catch.

- Блок try содержит операторы, которые мы хотим выполнить.

- Блок catch содержит объявление исключение.

Если операторы в блоке try вызовут исключение UserInputException, то программа перейдет к выполнению блока catch.

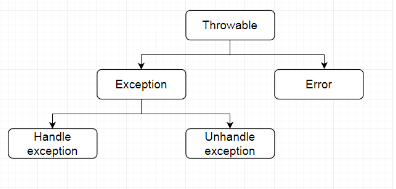
В блоке catch написан только один оператор.



Этот оператор выведет на консоль стек вызовов методов от начала программы до оператора, который вызвал прерывания. В большинстве случаев ваш код с блоком catch будет выглядеть так же.

**Иерархия исключений и множественный catch**

Все исключения в Java можно представить в виде дерева.



Throwable - базовый класс для всех нестандартных ситуаций при работе вашего приложения.

Error - ошибки, связанные с работой виртуальной машины. Такие как: выход за границы памяти, использование одинаковых библиотек с разными версиями.

Exception - ошибки, которые связаны с работой самой программы. Их можно разделить на два типа: проверяемые (checked) и непроверяемые (unchecked) исключения (RuntimeException).

Конструкция try-catch может содержать множественный блок catch. Это позволяет для конкретного прерывания произвести свою обработку.

Сделаем наследника от ru.job4j.ex.ElementNotFoundException.

Изображение выглядит как текст

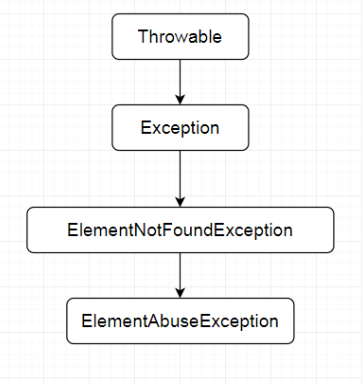
Автоматически созданное описание

ru.job4j.ex.ElementAbuseException - оно будет возникать, когда найденное слово попадает в запрещенный список.

Изображение выглядит как текст

Автоматически созданное описание

Получилась иерархия классов.



Предположим, что нам нужно разработать систему для отправки сообщений. У нас есть метод indexOf - этот метод проверяет, что в списке есть ключ шаблона сообщения. Метод может выкинуть исключение ElementNotFoundException. Метод sent отправляет сообщение, если ключ не входит в ключи запрещенных ключей. Метод может выкинуть исключение ElementAbuseException. Далее мы хотим сделать общий метод, который будет проверять, что такой ключ есть и отправлять сообщение.

Изображение выглядит как текст

Автоматически созданное описание

В коде выше в блоках catch представлена вся иерархия исключений, которая есть на рисунке выше. Далее мы рассмотрим, как эти блоки можно упростить и почему некоторые из них лучше вообще не ставить.

Чтобы вызывать метод indexOf и sent, нужно использовать блок try-catch, так как исключения, которые могут возникнуть в процессе выполнения этого метода, являются handle exception.  В случае если в блоке могут упасть разные сообщения, их нужно обработать в блоках catch. Каждый catch ловит свое исключение. Если классы находятся в одной ветке иерархии, то первым должен обрабатываться самый нижний класс в иерархии. В нашем случае это ElementAbuseException. В большинстве случаев нам нет нужды ловить конкретные исключения. В этом случае можно указать один блок catch с классом Exception.

Изображение выглядит как текст

Автоматически созданное описание

Такой код выглядит проще для понимания. Однако если вы не уверены в своем коде, то такой подход необходимо избегать. Почему? Наследником Exception является такой класс, как RuntimeException, а ловить исключения этого класса нежелательно, поскольку чаще всего выброс таких исключений — это нарушение логики программы. Поэтому стоит помнить об этом.

Также в коде пропал блок Throwable. Этот блок отлавливает любые нестандартные ситуации в нашей программе. Как вы помните в Java есть группа unhandle exception или RuntimeException. Эти исключения ловить через блок try-catch не надо. Если мы оставим блок catch, то мы будем захватывать RuntimeException тоже, а это не желательно. Почему нежелательно отлавливать RuntimeException? Потому, что эти исключения относятся к логическим ошибкам в коде и их нужно устранить за счет поиска ошибки в коде, а не за счет обработки исключения.

**ООА/П.**

ООА — это объектно-ориентированный анализ. ООА стоит над ООП.

Разработка программы начинается с технического задания. Техническое задание описывает желаемое поведение программы. ООА позволяет представить техническое задание в виде структур понятных разработчикам. В команде программистов за преобразование технического задания отвечает руководитель группы. То есть программист с большим опытом. Поэтому в большинстве заданий в курсе уже есть архитектурные решения. Эти решения вы должны запоминать и использовать для решения других задач.

В этом уроке мы познакомимся с первым этапом ООА — это выделение доменных моделей. Доменная модель — это базовые элементы программы, с помощью которых можно описать все поведение программы. Пример, проект Tracker. Здесь базовая модель — это класс Item. С ним мы делаем все действия. Пример, шахматы. Доменная модель: доска и фигуры. В реальной жизни мы везде используем доменные модели. Проблемы возникают в выборе моделей для программы.

**final**

Авторы языка Java заложили логику в применение ключевого слова *final* ­ сделать дальнейшее изменение объекта невозможным.

Данное ключевое слово можно применять для каждого из следующих трех случаев:

1. Для переменных (как локальных, так и полей класса);

2. Для методов;

3. Для классов.

**1. Final для переменных и полей**

Если нам необходимо, чтобы после инициализации переменной никто бы ее не смог изменить, при объявлении просто нужно добавить ключевое слово "final". Если мы задали переменную, у которой есть ключевое слово final, то изменить ее мы не сможем. Это означает что значение будет постоянным, в математике такие числа называются константами.

**2. Final для методов.**

К методам также можно применить модификатор final:

Что же означает использование методов использование ключевого слова final? Все очень просто - при наследовании данный метод нельзя будет переопределить (нельзя использовать метод). Если мы уберем ключевое слово final - ошибка компиляции исчезнет.

**3. Final для классов.**

Применять модификатор final можно и для классов - вернемся к предыдущему примеру и добавим модификатор. В данном случае будет обозначать, что мы не сможем создать наследников этого класса. Из-за того, что класс объявлен как final, можно в какой-то степени сказать, что и все его методы тоже становятся final, поэтому вообще использовать наследование в данном случае бессмысленно. Важно подчеркнуть важную особенность поведения ключевого слова final для ссылочных типов данных - после присвоения объекта, нельзя изменить ссылку на данный объект. Однако при этом мы можем изменить внутреннее состояние объекта. Рассмотрим пример - добавим класс User с двумя полями:

Изображение выглядит как текст

Автоматически созданное описание

Также добавим класс Main в котором в методе main() создадим один объект при этом при объявлении добавим ключевое слово final, после изменим внутренне состояние созданного объекта и выведем в консоль объект до и после изменения:

Изображение выглядит как текст

Автоматически созданное описание

Вывод в консоль будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

Как мы видим - внутреннее состояние объекта поменялось. Однако вот такая запись:

Изображение выглядит как текст

Автоматически созданное описание

приводит к ошибке.

Т.е. напоминаем - изменить внутреннее состояние объекта мы сможем (если для этого есть средства, в данном случае мы изменяли его с помощью сеттеров), однако ссылку на объект с ключевым словом final нам поменять не получится.

**Public record**

Уникальный модификатор класса, который автоматически позволяет создать полноценную модель данных, с переопределенными методами Hashcode, equals, toString.

1. Коллекции.

**Java Collection Framework** – это f интерфейсов и реализаций, которая является частью JDK и позволяет программистам пользоваться большим количеством структур для хранения набора данных. **Интерфейс Collection** используется для описания общего функционала, присущего коллекциям в Java. Важно подчеркнуть, что данный интерфейс является одним из ключевых понятий для Java Collection Framework.

Мы помним, что для хранения набора данных в Java предназначены массивы. Однако массивы не всегда удобно использовать в первую очередь потому, что массивы имеют фиксированный размер. Поэтому эту проблему в Java решают коллекции. Но дело не только в том, что коллекции обладают свойством автоматического расширения своего размера, но и то, что классы коллекций реализуют различные алгоритмы и структуры данных. Поэтому мы имеем хорошо оптимизированные структуры данных, которые описывают легкие способы работы с данными и при правильном выборе коллекции мы можем ускорить работу нашего приложения.

На вершине иерархии Java Collection Framework находятся 2 интерфейса:

1. Collection и 2. Map.

Эти интерфейсы разделяют все коллекции, входящие во в фреймворк на 2 части по типу хранения данных:

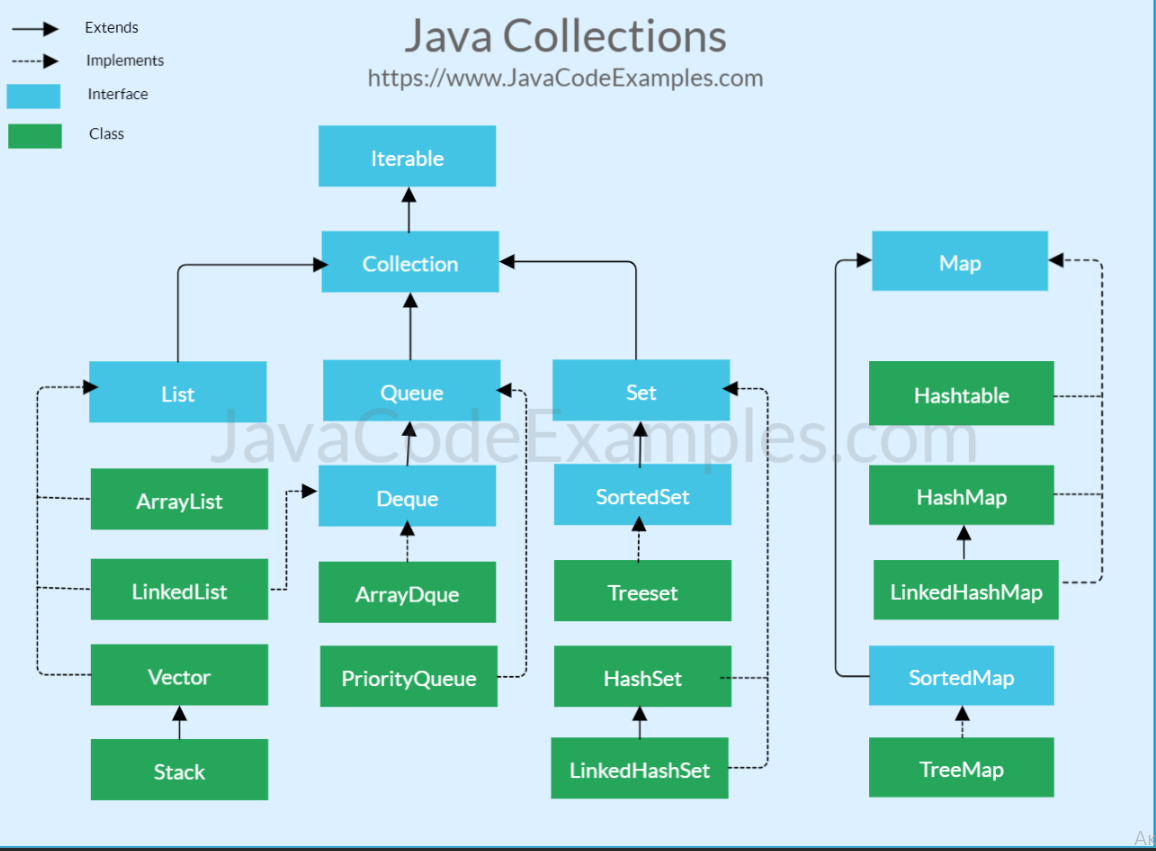
- простые последовательные наборы элементов;

- наборы пар «ключ-значение».

**Collection** – этот интерфейс определен в JDK с версии 1.2 и описывает основные методы работы с простыми наборами элементов, которые будут общими для всех его реализаций (к примеру size(), isEmpty(), add(E e) и т.д.). Интерфейс несколько был доработан с появлением generic в версии 1.5. В версии Java 8 было добавлено несколько новых методов для работы с лямбдами и Stream (например, stream(), parallelStream() и т.д.)

**Map** – этот интерфейс также находится в составе JDK с версии 1.2 и предоставляет программисту базовые методы для работы с данными вида «ключ-значение». По аналогии с Collection этот интерфейс был дополнен generic в версии Java 1.5 и в версии Java 8 появились дополнительные методы для работы с лямбдами, а также методы, которые часто реализовывались в логике приложения (getOrDefault(), putIFAbsent()).

Изобразим иерархию интерфейсов на схеме:



Интерфейс Collection является базовым для всех коллекций, определяя их общий функционал. Однако стоит обратить внимание, что этот интерфейс расширяет интерфейс Iterable, поэтому все объекты коллекций можно перебирать в так называемом улучшенном цикле for или в цикле for-each.

У интерфейса Iterable определен только один метод iterator(). Таким образом все наследники Collection (соответственно и Iterable) обязаны вернуть итератор. Итератор – это объект, который абстрагирует за единым интерфейсом доступ к элементам коллекции. К тому же итератор – это паттерн, который позволяет получить доступ к элементам любой коллекции без необходимости вникать в суть его реализации.

Рассмотрим методы, которые определены в интерфейсе Collection. Главным набором методов, который важен при рассмотрении любой коллекции является набор CRUD-операций (CRUD – сокращение от слов create, read, update and delete). На уровне Collection не существует способа обновить запись в коллекции, поэтому рассмотрим остальные возможности – добавить, прочитать и удалить. Будем использовать сигнатуры, которые представлены в документации, на данный момент не все обозначения для Вас ясны. Однако пока для Вас важно, что делают методы, для краткости – вместо заглавной E может быть использован любой ссылочный тип данных.

**1. Методы для добавления элементов:**

˗ boolean add(E e) – метод добавляет элемент в коллекцию и возвращает true если добавление выполнено успешно.

˗ boolean addAll(Collection<? extends E> c) – метод добавляет все элементы переданной в метод коллекции в коллекцию, у которой был вызван этот метод. Метод возвращает true если коллекция, у которой был вызван метод, изменилась.

**2. Чтение элементов из коллекции.**

Единственный способ, который определен на уровне Collection для чтения элементов из коллекции – это использовать итератор, на основе которого работает улучшенный цикл for или цикл for-each.

˗ Iterator<E> iterator() – метод возвращает объект Iterator, который позволяет проходить по элементам нашей коллекции.

**3. Удаление элементов.**

Для удаления можно использовать следующие методы:

˗ boolean remove(Object o) – метод удаляет объект, который мы передали в метод, и возвращает true, если удаление успешно, т.е. коллекция изменилась.

˗ boolean removeAll(Collection<?> c) – метод удаляет все элементы из коллекции, у которой был вызван этот метод, и которые также содержатся в переданной в метод коллекции. Таким образом после вызова этого метода в коллекции не будет тех элементов, которые содержатся в переданной в метод коллекции.

˗ boolean retainAll(Collection<?> c) – метод, который позволяет сохранить в коллекции только те элементы, которые содержатся в коллекции, которая была передана в метод. Таким образом в результате в коллекции останутся только те элементы, которые представлены в обоих коллекциях.

**4. Вспомогательные методы.**

˗ int size() – метод возвращает размер коллекции. Т.е. фактическое количество добавленных в коллекцию элементов.

˗ boolean isEmpty() – метод, который вернет true, если коллекция пока еще не содержит элементов.

˗ boolean contains(Object o) – метод возвращает true, если переданный в метод объект содержится в коллекции.

˗ void clear() – удаляет все элементы из коллекции. После применения этого метода коллекция будет пустой.

˗ Object[] toArray() – метод возвращает массив, который содержит все элементы, содержащиеся в коллекции, у которой был вызван этот метод.

Все методы Интерфейса List:

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html>

* 1. Коллекция ArrayList.

Начнем мы с изучения самой популярной коллекции — это java.util.ArrayList. Любая коллекция — это обычный класс. То есть, чтобы начать его использовать, нам нужно создать объект этого класса.



Давайте теперь пройдем по операциям.

**1. Добавление данных.**



При работе с ArrayList не нужно использовать index. То есть коллекции обеспечивают удобный интерфейс по сравнению с массивом.

**2. Замена или изменение данных.**



Для замены ячейки нужно использовать метод set с указанием индекса. Этот метод используется редко, потому что нам нужно знать позицию элемента. В бизнес-приложениях с индексом не работают. Далее в этой главе мы познакомимся с Map. Эта коллекция подходит для этой задачи лучше.

**3. Удаление данных.**



 Удаляем мы ячейку по индексу.

**4. Чтение данных.**

Изображение выглядит как текст

Автоматически созданное описание

Чтение данных в ArrayList происходит аналогично массиву. В ArrayList есть метод size, который возвращает размер коллекции. Метод get позволяет получить элемент по индексу. Однако, для коллекции и массивов предпочтительней использовать цикл for-earch.

Все методы:

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html>

* 1. Коллекция LinkedList.

В Java, коллекции представляют собой фреймворк. То есть набор классов, которые можно легко расширить. Все коллекции можно разбить по группам интерфейсов. Мы уже говорили про коллекцию **java.util.ArrayList**. Этот класс реализует **java.util.List**. У интерфейса java.util.List есть другая реализация **java.util.LinkedList**. Этот класс внутри использует связанные списки.

Основное отличие этих коллекций — это организация вставки и удаления элементов. Детально про отличия этих реализаций мы поговорим в следующем уровне. Интерфейс java.util.List имеет метод add(index, value). То есть мы можем добавить элемент в нужный индекс.

У этого метода интересное поведение. Если ячейка уже занята, то элементы сдвигаются вправо, а не заменяются, как в массиве.

Пример:

Изображение выглядит как текст

Автоматически созданное описание

Вывод:

Изображение выглядит как текст

Автоматически созданное описание

Это поведение мы будем использовать для решения разных задач.

Одной из удобных структур данных является очередь с приоритетами. Очередь с приоритетами — это коллекция, в которой элементы при добавлении располагаются в определенном порядке. Это позволяет их извлекать в нужном порядке. Например. У нас есть список задач на день с указанием их важности. Мы бы хотели выполнять задачи по их важности.

Все методы:

<https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/LinkedList.html>

* 1. Коллекция Set, HashSet

java.util.Set - классы, реализующие этот интерфейс не могут содержать дубликаты элементов. Их также называют множества.

Начнем мы знакомство с коллекции **java.util.HashSet**.

**1. Создание коллекции.**



Как вы уже поняли, в любой коллекции используется обобщенный тип данных. Напомню, что он указывает на тип данных, которым планируется хранить в коллекции. Кроме того, как и для случая коллекций, реализующих интерфейс List, начиная с JDK 7 при создании коллекции типа java.util.Set принято использовать diamond operator(<>), позволяющий не указывать тип в правой части, если он уже указан в левой:

**2. Вставка элементов.**



В коллекциях типа java.util.Set нет возможности указать индекс. Так же здесь нет возможности заменить элемент.

**3. Замена элемента.**

Коллекции java.util.Set не поддерживают замену элементов, потому что нет возможности явно указать ячейку. Это связано с внутренней реализацией коллекций типа java.util.Set.

**4. Чтение данных.**

Прочитать данные можно через цикл for-each.

Изображение выглядит как текст

Автоматически созданное описание

Вопрос, а как мы получаем данные из java.util.Set если там нет механизма извлечения данных. На самом деле все коллекции реализуют шаблон проектирования - итератор. Этот шаблон позволяет получить все элементы из любой коллекции. Шаблон итератор основан на интерфейсе **java.util.Iterable**.

Он имеет три основных метода:

**boolean hasNext()** - проверяет на наличие следующего элемента в списке.

**T next()** - возвращает элемент из коллекции и переводит указатель на следующий элемент.

**void remove()** - удаляет элемент из коллекции и переводит указатель назад.

Указатель — это механизм позволяющий извлечь одиночный элемент из коллекции. Например, для массива указатель будет индекс элемента.

Подробно про итераторы будет рассказано в следующем уровне.

На данный момент вам нужно запомнить, что для перебора элементов внутри java.util.List и java.util.Set нужно использовать цикл for-earch.

Сам цикл for-each внутри использует механизм итератора.

**Дубликаты в java.util.Set.**

Главной особенностью коллекций java.util.Set является хранение только уникальных элементов. Давайте посмотрим на примере добавления в коллекцию HashSet дубликатов.

Изображение выглядит как текст

Автоматически созданное описание

На консоли мы получим 3 элемента.

Изображение выглядит как текст

Автоматически созданное описание

Этот механизм удобно использовать, если нам нужно получить только уникальные элементы. Коллекции java.util.Set не будут добавлять дублирующие элементы.

**Доп. Информация**

Уникальность элементов в HashSet проверяется через методы equals и hashCode. Если мы хотим хранить в HashSet пользовательские темы данных нам нужно обеспечить эти модели методами equals и hashCode.

Запомните, всегда переопределяйте методы equals и hashCode в моделях данных. Напомню, модель данных — это класс, который является контейнером и описывает бизнес-модель вашего приложения. Например: клиент банка, ученик, студент. Эти классы будут хранить информацию, но не будут выполнять логику

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Set.html>

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/HashSet.html>

* 1. Map, HashMap.

Ранее, мы познакомились со структурами данных java.util.ArrayList, java.util.LinkedList, java.util.HashSet. Эти коллекции могут содержать однотипные плоские модели данных. Часто возникает ситуация, когда нам нужно хранить не плоскую модель, а пары ключ-значение.

Конечно, мы можем создать модель Pair(key, value) и хранить данные в ArrayList. Такое решение возможно, но поиск значения в этой структуре будет зависеть от количества элементов.

Другое решение — это использование ассоциативных карт или справочников. В Java эти коллекции описываются интерфейсом **java.util.Map**. Он имеет несколько реализаций. Мы начнем с разбора **HashMap**.

Интерфейс java.util.Map имеет два базовых метода



И метод получения значения по ключу.



**1. Вставка элементов.**

Вставить данные в карту можно используя метод **put**. В предыдущих коллекциях для вставки данных использовался метод add. Имена методов отличаются.

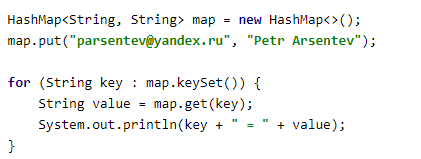
Изображение выглядит как текст

Автоматически созданное описание

Здесь мы видим, в карту мы добавляем сразу два аргумента, то есть делаем ассоциацию ключа с его значением.

**2. Вывод элементов.**

Вывести все элементы на печать или итерироваться по элементам.



В интерфейсе java.util.Map есть методы **keySet()**, который возвращает коллекцию java.util.Set состоящую только из ключей.

Через метод **get** мы получаем значение, ассоциированное с этим ключом.



Другой способ получить все значения — это использовать метод **entrySet()**. Этот метод вернет коллекцию java.util.Set состоящую из объекта Map.Entry, которая сразу содержит и ключ, и значение.

Изображение выглядит как текст

Автоматически созданное описание

**3. Обновление пары.**

Пару можно обновить тем же методом, что и вставить. Ключ пары должен быть уникальный. Если в Map уже есть такой key, то ячейка value будет заменена на новое значение.

Изображение выглядит как текст, внутренний, снимок экрана, документ

Автоматически созданное описание

В примере выше размер Map будет 1, а не 2, потому что вторая пара имеет ключ, который уже был в Map.

Проверка ключей происходит по методу equals. Важно помнить, что объект key должен правильно реализовывать метод equals, иначе Map работать не будет.

**4. Удаление.**

Удаление происходит по ключу через метод remove(key);

Изображение выглядит как текст

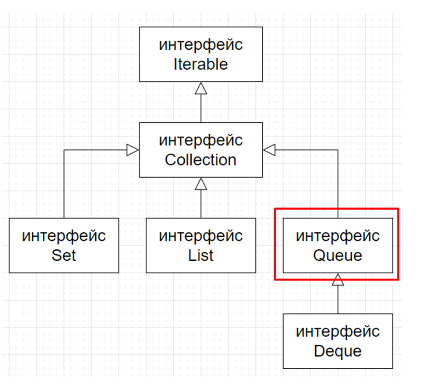
Автоматически созданное описание

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Map.html>

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/HashMap.html>

* 1. Queue и Dequeue.
     1. Queue.

Познакомимся еще с одним интерфейсом, который расширяет возможности интерфейса Collection. Это интерфейс **Queue**.



На схеме мы видим, что **Queue** является прямым наследником Collection, а тот интерфейс в свою очередь наследует интерфейс Iterable. Следовательно, помимо тех методов, которые определены в Queue, в нем также доступны все методы вышестоящих интерфейсов, которые мы рассматривали ранее.

Для начала разберемся что вообще такое очередь. Структура данных, которая в программировании называется очередью, представляет из себя хранилище однотипных элементов, в котором первым будет извлекаться тот элемент, который был туда и вставлен первым. Такой способ организации хранения данных называется FIFO, по первым буквам фразы, описывающей этот способ хранения – First-In-First-Out.

Очередь работает по тому же принципу, как и любая очередь в реальной жизни – например очередь за билетами в кино (человек, который первым встал в очередь первым, дойдет до кассы и купит билет). Следовательно, тот, кто станет в очередь последним – и купит билет последним.

Очередь – очень удобный инструмент, который используется для моделирования реальных ситуаций ожидания клиентов в банке, вылета самолетов или передачи данных по Интернету. Рассмотрим методы этого интерфейса, а потом попробуем их использовать на практике.

1. Метод **add()** – добавляет элемент в конец очереди (за исключением PriorityQueue, там элемент добавляется согласно своему приоритету).

2. Метод **offer()** – предпринимает попытку вставки элемента в конец очереди (зависит от размера очереди).

3. Методы **remove()** и **poll()** – возвращают и удаляют верхний (первый) элемент из очереди. Разница в том, что **remove()** вызовет ошибку при использовании на пустой список, а **poll()** нет.

4. Методы **element()** и **peek()**– позволяют вернуть элемент из очереди без его удаления. Разница в том, что **element()** вызовет ошибку при использовании на пустой список, а **peek()** нет.

Подведем некоторые итоги:

1. Очередь используется для вставки элементов в конец очереди и удаления из начала очереди. Она следует концепции FIFO.

2. В Java интерфейс Queue поддерживает все методы интерфейса Collection, включая вставку, удаление и т.д.

3. LinkedList , ArrayBlockingQueue и PriorityQueue — наиболее часто используемые реализации интерфейса Queue.

4. Очереди, доступные в пакете java.util, являются неограниченными очередями.

5. Очереди, доступные в пакете java.util.concurrent, являются ограниченными очередями.

Также краткое описание интерфейса Queue можем увидеть в официальной документации:

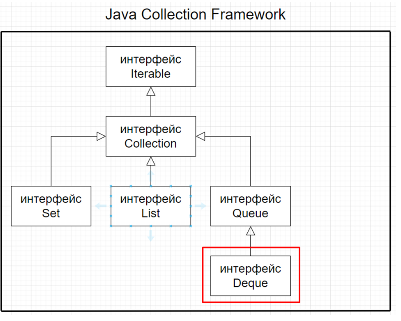
[Ссылка](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Queue.html)  
Можно познакомиться со следующими туториалами:

[Ссылка 1](https://www.programiz.com/java-programming/queue)

[Ссылка 2](https://www.baeldung.com/java-queue)

* + 1. Deque.

Ранее был рассмотрен интерфейс Queue, который в свою очередь расширяет интерфейс Collection. Как мы помним, такая очередь позволяет поместить элементы в один конец очереди, а извлекать их – из другого. Поэтому такие очереди называются односторонними. В текущей задаче мы познакомимся с интерфейсом, который расширяет интерфейс Queue – это интерфейс Deque. Изобразим на схеме этот интерфейс в иерархии наследования интерфейсов в Collection Framework:

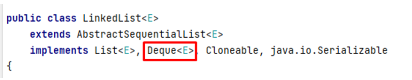


Этот интерфейс также является очередью, но ее отличительной особенностью то, что эта очередь является двухсторонней, т.е. она поддерживает удаление и добавление элементов с обоих концов очереди. Таким образом, в отличие от Queue, здесь также можно организовать способ хранения элементов по принципу LIFO (сокращение по первой букве от описания этого принципа на английском языке Last In First Out – последний пришел, первый вышел). Такую структуру данных в программировании называют стеком. Пример аналогии из жизни – стопка тетрадей на проверку у учителя, учитель проверяет их по порядку и та тетрадь, которая была положена в эту стопку первой, будет проверена последней.

Также с таким поведением мы уже могли встретиться при выполнении наших программ – если запустить приложение в режиме отладки, то слева:

Мы можем увидеть так называемый стек вызова методов – так вот, тот метод, который был вызван первым в этом стеке – будет выполнен последним.

Наиболее часто используемой реализацией интерфейса Deque является класс LinkedList. Вы спросите – как так мы же использовали эту реализацию и с Queue? Вопрос очень уместный, надо разобраться – если посмотреть на объявление класса LinkedList:



То мы увидим, что он на самом деле реализует интерфейс Deque. А тот наследует интерфейс Queue со всеми его методами. Следовательно, при создании объекта LinkedList можно использовать оба базовых типа указанных интерфейсов – следующие обе строки кода будут работать без проблем:

Изображение выглядит как текст, снимок экрана, оранжевый

Автоматически созданное описание

В этом и есть одно из самых ярких проявлений полиморфизма – вместо того, чтобы слева от знака присваивания ставить конкретный класс, мы используем интерфейс и, следовательно, можем использовать методы, определенные в нем, а также всех вышестоящих по иерархии интерфейсов.

Таким образом, разница в этих записях в том, что для первой строки будут доступны только методы интерфейса Queue и всех вышестоящих интерфейсов (Collection и Iterable). Для второй строки – будут добавлены все методы, которые определены в интерфейсе Deque.

Разберемся подробнее со всем многообразием методов, доступных нам благодаря интерфейсу Deque.

1. Добавление элементов:

Метод **add()** – позволяет добавить элемент в конец очереди. Однако тут конечно же есть особенности – в двухсторонней очереди добавление возможно с обоих ее направлений.

Методы **addFirst()** и **addLast()**. Функциональность этих методов легко определяется по их названиях – они позволяют добавить элемент в начало и конец очереди соответственно.

2. Удаление элементов.

Для удаления доступно несколько различных методов, но логика примерно так же, как было и с добавлением элементов. Методы poll() и remove() мы уже разбирали у очереди – разница в том, что первый вернет null при пустой очереди, а второй – сгенерирует исключение. Обсудим следующие метод:

- **pop()** – позволяет удалить головной удалить из очереди и вернуть его. Если очередь пуста – будет сгенерировано исключение NoSuchElementException. **//**Хз, ничем не отличается от **remove()**).

**- removeFirst()** и **pollFirst()** – методы также позволят удалить первый элемент из очереди с его возвратом. Первый сгенерирует исключение при пустой очереди, второй – вернет null.

- **removeLast()** **и pollLast()** – методы позволят удалить последний элемент из очереди с его возвратом. Первый сгенерирует исключение при пустой очереди, второй – вернет null.

Таким образом мы делаем вывод – предпочтительнее использовать методы **poll()**, **pollFirst()** и **pollLast()**, поскольку они не генерируют непроверяемых исключений, когда наша очередь пустая.

3. Чтение элементов.

К ранее рассмотренным методам **element()** **и peek()**, которые позволяют прочитать первый элемент из очереди, для чтения также добавлены методы **peekFirst()** и **peekLast()** – методы позволяют соответственно прочитать без удаления первый и последний элемент из очереди.

Более интересным является итерация по элементам очереди с помощью итератора – для этого послужит метод **iterator()** и его методы **hasNext()** и **next()**. Продемонстрируем на примере:

Изображение выглядит как текст

Автоматически созданное описание

Вывод в консоль будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

Т.е. вывод в консоль будет аналогичным с тем, если бы мы очередь перебирали в цикле for-each. Однако, наша очередь двусторонняя и поэтому с помощью итератора мы можем перебирать нашу очередь в обратном порядке. Для этого определен метод **descendingIterator()**:

Изображение выглядит как текст

Автоматически созданное описание

В выводе мы увидим обратный порядок наших элементов в очереди.

Небольшие выводы:

1. В очередь Deque элементы можно добавлять, удалять и извлекать с обоих ее сторон. Она следует как принципу FIFO, так и при правильной организации можно получить поведение по принципу LIFO.

2. В Java интерфейс Deque поддерживает все методы интерфейса **Queue**, **Collection**, включая вставку, удаление и т.д.

3. Необходимо быть предельно аккуратными при использовании методов, которые при своей работе генерируют непроверяемое исключение. Предпочитайте использовать методы **poll()**, **pollFirst()**, **pollLast()**, **peekFirst()**, **peekLast()**, **peek()**.

4. Наиболее часто используемой реализацией интерфейса Deque является **LinkedList**.

Краткое описание интерфейса можно найти в официальной документации:

[Ссылка](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Deque.html)

А также в следующих туториалах:

[Ссылка 1](https://jenkov.com/tutorials/java-collections/deque.html)

[Ссылка 2](https://www.programiz.com/java-programming/deque)

[Ссылка 3](https://www.baeldung.com/java-array-deque)

* 1. Сортировка.

Java автоматически может представить любой символ в таблице ASCII. Это свойство позволяет отсортировать любой массив с символами.

Например: {f, a, e, d} соответствуют кодам {102, 97, 101, 100}. После сортировки по возрастанию массив примет вид: {a, d, e, f}.

***Встроенная сортировка в Java***

В Java большинство коллекций поддерживает метод Collections.sort, который позволяет отсортировать коллекцию.

Давайте рассмотрим это c коллекцией java.utils.List.

Изображение выглядит как текст

Автоматически созданное описание

1. Мы создали список из чисел и передали его в статический метод Collections.sort.

2. Java производит сортировку по возрастанию. Внутри метода sort осуществляется сортировка методом слияния. В первом модуле вы реализовывали часть этого алгоритма. Чтобы использовать алгоритм слияния, сортируемый тип данных должен поддерживать интерфейс **java.util.Comparable**. Все встроенные типы данных поддерживают этот интерфейс.

Давайте откроем класс java.lang.Integer. Мы видим, что этот класс реализует интерфейс **java.util.Comparable**.

Изображение выглядит как текст

Автоматически созданное описание

Интерфейс **Comparable** имеет всего один метод **compareTo(T t)**. Этот метод возвращает меньше нуля, ноль, больше нуля.

Меньше нуля - если входящее значение больше, чем текущий объект.

Ноль - если значения равны.

Больше нуля - если текущее значение больше, чем входящее. Примечание. Текущее значение — это объект, у которого вызван метод compareTo.

Изображение выглядит как текст

Автоматически созданное описание

Метод compareTo вызван у объекта first, то есть текущее значение будет 1, входящее значение будет 2. Результат вычисления метода будет меньше нуля. Так как входящее значение больше, чем текущее. Давайте рассмотрим пример сравнения строк.

Изображение выглядит как текст

Автоматически созданное описание

При сравнении каждая строка раскладывается на массив символов, которые в свою очередь преобразуются в массив чисел.

{P, e, t, r} -> {80, 101, 116, 114}

{I, v, a, n} -> {73, 118, 100, 110}

Дальше Java по очереди сравнивает каждую ячейку массива. Если вычисление дает результат 0, то вычисление повторяется для следующей пары, пока не дойдет до конца или пока результат не станет отличен от 0. В этом примере первое вычисление вернет число 7. Так как 80 – 73 = 7. То есть больше нуля. Это значит, что слово "Petr" больше, чем слово "Ivan".

***Сортировка по убыванию***

Через интерфейс java.util.Comparable мы можем задать только один порядок сортировки. Чтобы сделать другой порядок, в Java есть интерфейс, который не привязан к модели данных. Это интерфейс **java.util.Comparator**.

В нем есть метод **int compare(T first, T second)**, который принимает сразу два параметра.

Для всех встроенных типов Integer, String и т.д. в Java есть готовые классы, реализующие этот интерфейс.

В классе java.util.Collections.

Изображение выглядит как текст

Автоматически созданное описание

Чтобы воспользоваться этим классом, в метод sort с коллекцией передается объект Comparator.

Изображение выглядит как текст

Автоматически созданное описание

Чтобы использовать интерфейс **Comparable**, его нужно переопределить в нашей модели данных (также в любом классе, где надо сравнивать значения внутри объектов) и имплементировать этот класс.



Изображение выглядит как текст

Автоматически созданное описание

Для использования интерфейса **Comparator** стоит создать отдельный класс и уже из него уже имплементировать наш интерфейс.

Использование **compare()**:

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**Дополнительно:**

**TreeSet.**

В Java есть интерфейс java.util.Set. Классы этого интерфейса — это **java.util.HashSet** и **java.util.TreeSet**. Как вы знаете, эти коллекции имеют свои особенности. Они не содержат дубликатов. Рассмотрим про **java.util.TreeSet**. Эта коллекция сразу сортирует элементы. **TreeSet** требует от входящего типа данных поддержания интерфейса **java.util.Comparable**. При каждой вставке в **TreeSet** элемент будет искать нужную ячейку через механизм сравнения элементов. Внутри **TreeSet** используется бинарное дерево поиска. Давайте рассмотрим вариант, когда нам нужно расположить элементы по убыванию. Для этого в конструктор коллекции нужно передать компаратор.

**Изображение выглядит как текст

Автоматически созданное описание**

**Комбинированная сортировка.**

Поговорим о комбинированных случаях сортировки. Например, пользователь хочет отсортировать первое поле по дате, а второе поле по важности. Другой пользователь хочет отсортировать по имени и дате. В этом случае нужно использовать компараторы. Но как быть, если у модели 10 полей? Пользователь может выбрать 100 различных комбинаций. Нужно ли нам создавать 100 классов компаратора? Конечно же нет. Нам нужно создать компаратор для каждого поля, а потом их комбинировать.

Сделаем для модели данных два компаратора по убыванию: по полям имени и приоритета.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

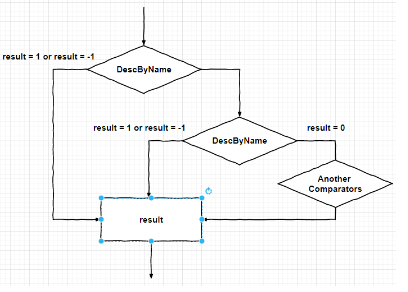
Чтобы обеспечить сортировку по убыванию, в компараторе мы сравниваем второй параметр с первым, а в случае сортировки по возрастанию - первый со вторым. Теперь давайте попробуем совместить их. В интерфейсе **Comparator** есть метод с реализацией по умолчанию – **thenComparing()**. Этот метод принимает другой компаратор. Таким образом, можно соединить или скомбинировать любую сортировку.

Изображение выглядит как текст

Автоматически созданное описание

Как работает комбинированный компаратор?

Если результат первого компаратора JobDescByName равен 0, то есть объекты равны, то проверка переходит ко второму компаратору JobDescByPriority.



**Ссылки:**

<https://yandex.ru/search/?text=comparable+java+%D0%B8%D0%B5%D1%80%D0%B0%D1%80%D1%85%D0%B8%D1%8F&lr=213>

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Comparator.html>

[https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Iterator.html](https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/Iterator.htmlИнструменты)

* 1. **Инструменты.**
     1. JaCoCo. Процент покрытия тестами.

Одним из показателей качества кода является процент покрытия тестами. Это единственный показатель, который мы можем измерить с помощью программы. В идеале надо стремиться, чтобы проект был покрыт тестами на 80%. Но в реальном мире такое бывает редко. **JaCoCo** – плагин, который позволяет собрать статистику по покрытию нашего проекта тестами.

Результат:

Изображение выглядит как стол

Автоматически созданное описание

* + 1. Генерация документации по JavaDoc.

JavaDoc — это генератор документации в HTML-формате из комментариев исходного кода Java.

На примере класса ArrayList, мы научимся генерировать документацию в HTML-формате. Если мы обратимся к реализации метода contains(Object o) класса ArrayList(), то увидим такой код:

Изображение выглядит как текст

Автоматически созданное описание

и если мы посмотрим описание этого метода в документации по ссылке ниже:

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/ArrayList.html#contains(java.lang.Object)>

то мы увидим, что текст в документации совпадает с тем текстом, который заключен в конструкцию **/\*\* какой-то текст \*/**. Задача этого и следующего задания - научиться сначала писать комментарии, а потом на их основе генерировать качественную документацию к нашему коду.

О качестве кода, который пишет программист, говорит не только его "чистота", но и наличие документации к написанному коду. Если документация и код разделены, то могут появиться проблемы, которые связанны с необходимостью внесения изменений в соответствующие разделы сопроводительной документации при изменении программного кода.

Все существующие среды разработки IDE приложений на языке Java предоставляют инструмент по связыванию кода с документацией в процессе разработки с использованием **JavaDoc**. Чтобы пользоваться этой возможностью просто необходимо соответствующим образом написать комментарий к коду. Поэтому комментарии к коду нужны не только для простого его комментирования, но и для составления и оформления документации.

Разработан специальный синтаксис для оформления документации в виде комментариев и инструмент для создания из комментариев документации. Таким инструментом является **JavaDoc**, который обрабатывает файл с исходным кодом, выделяет помеченную документацию из комментариев и связывает с именами соответствующих классов, методов и полей. Таким образом, приложив минимальные усилия для создания комментариев к коду, можно получить качественную документацию к программе.

**JavaDoc** — это генератор документации в HTML-формате из комментариев исходного кода Java. Для создания элементов, которые позволят программисту анализировать структуру приложения, **JavaDoc** предоставляет API. При этом комментарий должен находиться перед документируемым элементом. При этом важная особенность **JavaDoc** комментариев, что в отличие от комментариев, с которыми мы сталкивались ранее, в данном случае комментарии помещаются в такую конструкцию:

**/\*\* комментирование документации \*/**

т.е. в начале у нас идет две \*\*, а не одна как ранее (одна \* — это просто многострочный комментарий в коде).

С помощью утилиты **JavaDoc**, которая входит в состав JDK, комментарий документации можно извлекать и помещать в HTML файл. Утилита **JavaDoc** позволяет вставлять HTML тэги и использовать специальные ярлыки (дескрипторы) документирования. HTML тэги заголовков не используют, чтобы не нарушать стиль файла, сформированного утилитой.

**Есть 2 типа дескрипторов:**

1. Автономные - они начинаются со знака @ и должны помещаться с начала строки комментария;

2. Встроенные - начинаются с фигурной скобки и могут применяться внутри описания.

Комментарии документации применяют для документирования классов, интерфейсов, полей (переменных), конструкторов и методов. В каждом случае комментарий должен находиться перед документируемым элементом.

Давайте познакомимся с описанием основных дескрипторов, которые Вы будете применять, разделив их на три категории:

1. Применяются только для **классов или интерфейсов**:

@author - указывается имя автора;

@version - указывается версия нашего класса. Не более одного дескриптора на класс.

2. Применяются только для **методов**:

@param - приводится описание входного параметра описываемого метода;

@return - описывается возвращаемое значение документируемого метода;

@exception имя\_класса\_исключения - приводится описание исключения, которое может бросить описываемый метод.

3. Применяются для **классов, интерфейсов, полей и методов**:

@since - указывается с какой версии доступен документируемый элемент;

@see - тут можно указать ссылку на другое место в документации;

@deprecated - описываются устаревшие блоки кода;

{@link reference} - можно указать ссылку на другой класс, метод и прочее.

1. **Потоки.**
   1. **Анонимные классы.**

Нам нужно отсортировать модель по полю size. Для этого нам нужно создать класс, который реализует интерфейс java.util.Comparator. Однако, мы знаем, что в проекте есть только один случай, где нужно использовать этот компаратор. Чтобы не засорять папки проекта лишними файлами, в Java добавили возможность создать объект на основании анонимного класса.

Поле size:

Изображение выглядит как текст

Автоматически созданное описание

java.util.Comparator — это интерфейс. Как мы помним, объект из интерфейса создать нельзя, потому что в нем могут быть абстрактные методы.

Решить эту проблему позволяет анонимный класс. Обратите внимание, что после имени интерфейса мы указываем фигурные скобки. Внутри фигурных скобок содержится тело нового класса (методы, поля). В нем же мы и определяем недостающие методы. Оператор **new** перед именем класса сообщает виртуальной машине, что из полученного класса нужно сразу создать объект.

Опишем создание анонимного класса через шаблон:

Изображение выглядит как текст

Автоматически созданное описание

В анонимном классе нет имени класса. Отсюда и пошло его название.

Создание анонимного класса:

Изображение выглядит как текст

Автоматически созданное описание

Анонимный класс можно создать на основании класса или интерфейса. В примере с java.util.ArrayList мы переопределили метод **add()**, добавив вывод на консоль элемента при добавлении элемента в коллекцию.

Переопределение add():

Изображение выглядит как текст

Автоматически созданное описание

**Обобщенные типы и анонимный класс**

Во всех примерах выше из анонимных классов удалены обобщенные типы для упрощения конструкции. Желательно всегда их указывать.

В первом примере обобщенный тип будет Attachment. Напоминаю, что все обобщенные типы указываются в угловых скобках.

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

Второй пример:

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

* 1. **Lambda.**
     1. **Lambda. Общие положения.**

В программировании термин лямбда обозначает возможность передать в качестве параметра блок кода. Переданный блок кода может быть вызван в любой момент времени или не вызван вообще.

Рассмотрим пример с компаратором.

Изображение выглядит как текст

Автоматически созданное описание

Можно переделать в:



Правая сторона — это вывернутый наизнанку метод compare.

Сравните:



И



Левая часть содержит входные параметры left и right.

Здесь не указывается тип данных. Java получает тип данных из объявления переменной Comparator<Attachment>

Символы тире и больше указывают на начало тела метода.



Если выражение содержит единственный оператор return, то его стоит убрать.

**Лямбда блок.**

Лямбда удобно использовать, когда функция описывается в одну строчку.



Иногда вычисление может занять несколько строчек. В этом случае лямбда описывается через блок. Вспомним задачу, где нам нужно отсортировать названия задач по номеру. Номер содержится в названии.

Изображение выглядит как текст

Автоматически созданное описание

Чтобы написать компаратор для этого случая, нужно вырезать номер, преобразовать его в число и сравнить.

Изображение выглядит как текст

Автоматически созданное описание

Опишем лямбда-блок через шаблон.

Изображение выглядит как текст

Автоматически созданное описание

Рассмотрим пример с выводом отладочной информации на консоль.

Отладочная информация — это дополнительные данные, которые добавляются в код с целью получения информации о ходе выполнения кода. Здесь добавлена отладочная информация, которая выводит в консоль результат каждого сравнения объектов компаратором:

Изображение выглядит как текст

Автоматически созданное описание

* + 1. Зона видимости в лямбда-выражениях и исключения.

Познакомимся с зоной видимости переменных в лямбда-выражениях.

1.Переменная, объявленная внутри лямбда-выражения, доступна только в самом лямбда-выражении.

**2. Переменная, объявленная вне лямбда-выражения, доступна как внутри лямбда-выражения, так и вне его.**

3. В лямбда-выражении могут быть доступны только те переменные, которые после своей инициализации больше не изменялись. Поэтому, если вам надо использовать в лямбда-выражении внешние переменные, которые были изменены после их первоначальной инициализации, то используйте вместо них промежуточные переменные, специально объявленные и инициализированные нужным значением непосредственно перед лямбда-выражением:

**4. Исключения в лямбда-выражениях.**

В лямбда-выражениях нельзя использовать методы, которые могут выбросить исключения. Поток данных не может прерываться.

А как быть? Не использовать лямбда-выражения с исключениями.

* 1. **Функциональные интерфейсы.**

Функциональный интерфейс — это интерфейс, который содержит только **один абстрактный** метод. Абстрактный метод — это метод, в котором нет реализации.

Для того, чтобы подчеркнуть, что интерфейс является функциональным, используется аннотация @FunctionalInterface. При этом, как и аннотация @Override она не является обязательной, код будет работать и без нее. Однако, она позволяет произвести дополнительную валидацию нашего кода - если мы попытаемся добавить в такой интерфейс еще один абстрактный метод, то на стадии компиляции мы получим ошибку.

Стоит подчеркнуть следующую особенность функциональных интерфейсов - они появились в Java версии 8 и после этого структура интерфейсов претерпела некоторые изменения:

- В интерфейсе можно добавлять статические методы и методы с реализацией по умолчанию (помечаются ключевым словом default). Так вот в функциональном интерфейсе таких методов может быть сколько угодно. Такие методы уже будут содержать реализацию;

- в функциональном интерфейсе может быть также дополнительно объявлено один или несколько абстрактных методов, которые по сигнатуре совпадают с методом из класса **java.lang.Object**, в таком случае они не влияют на ограничение в один абстрактный метод для функционального интерфейса. Такое вы можете увидеть в интерфейсе **Comparator**, если обратитесь к исходникам.

Ранее мы уже использовали функциональные интерфейсы. В разделе *"Collections. Lite."* **java.util.Comparable** и **java.util.Comparator** — это функциональные интерфейсы.

Java имеет пакет **java.util.function**, который содержит все необходимые функциональные интерфейсы. Если вы создаете свой функциональный интерфейс, то что-то в вашей программе не так.

Рассмотрим применение следующих встроенных функциональных интерфейсов:

1. **Supplier;**

2. **Consumer, (BiConsumer);**

3. **Predicate, (BiPredicate);**

4. **Function, (BiFunction);**

5. **UnaryOperator и BinaryOperator.**

Прежде чем разбирать эти интерфейсы более подробно, рассмотрим один момент, который поначалу может запутывать и пугать. Посмотрим, как например объявлен интерфейс **BiFunction:**

Изображение выглядит как текст

Автоматически созданное описание

Что же означают буквы T, U, R –, это элементы обобщенного программирования. На данном этапе мы просто воспринимаем как должное и необходимо понимать, что это нужно для того, чтобы под каждый тип данных не создавать отдельный интерфейс, а просто вместо T U и R мы можем просто поставить любой тип и многократно переиспользовать один и тот же интерфейс многократно с разным набором типов данных. Например, если мы объявим и присвоим такие типы данных:



То метод **apply()** который будет выполнять всю работу, будет иметь такую сигнатуру:



Т.е. все обобщенные типы будут заменены на конкретные типы, которые мы указали в <>.

А теперь подробнее обо встроенных функциональных интерфейсах:

**1. Supplier.**

**Supplier** (поставщик) используется для создания какого-либо объекта и при этом ему не требуется входной параметр. В стандартной библиотеке Java интерфейс представлен в следующем виде:

Изображение выглядит как текст

Автоматически созданное описание

Пока все звучит туманно, давайте попробуем использовать этот интерфейс на практике. С помощью интерфейса создадим строку и выведем ее в консоль:

Изображение выглядит как текст

Автоматически созданное описание

Обратите внимание, что для того, чтобы использовать функциональные интерфейсы надо импортировать их – import java.util.function.\*. В самом начале мы объявляем интерфейс **Supplier** с обобщенным типом String и записываем его в переменную **sup**. Основное применение этого интерфейса – создание новых объектов. Давайте попробуем создать HashSet строк из списка с помощью этого интерфейса:

Изображение выглядит как текст

Автоматически созданное описание

вывод в консоль в данном случае будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

Как и полагается работе коллекции типа Set из исходной коллекции были удалены дубликаты строк.

**2. Consumer и BiConsumer.**

**Consumer** (потребитель) используется в том случае, если нам нужно применить какое-то действие или операцию к параметру (для **BiConsumer** два параметра) и при этом у метода нет возвращаемого значения. Объявляются эти интерфейсы следующим образом:

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

Два интерфейса отличаются только одним – количеством параметров, которые принимает метод в этом интерфейсе. Это относится ко всем интерфейсам – если видите приставку **Bi** – значит отличие лишь в количестве параметров. Вернемся к ранее написанному примеру и перепишем вывод в консоль на применение интерфейса **Consumer**:

Изображение выглядит как текст

Автоматически созданное описание

Перепишем пример на применение **BiConsumer**:

Изображение выглядит как текст, внутренний, оранжевый

Автоматически созданное описание

И вывод в консоль будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

Т.е. в первом случае мы вывели на экран строку, которую передавали в метод и она была одна. Во втором случае при объявлении функционального интерфейса мы указываем два типа обобщения и в лямбду, соответственно, передаем 2 параметра. При этом важно понимать – параметры в **Biconsumer** могут быть разных типов – перепишем пример с коллекцией из 1 пункта следующим образом:

Изображение выглядит как текст

Автоматически созданное описание

При этом вывод в консоль будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

**3. Predicate (BiPredicate).**

**Predicate** (утверждение) наиболее часто применяется в фильтрах и сравнении и объявляются они следующим образом:

Изображение выглядит как текст

Автоматически созданное описание

Т.е. в метод **test()** передается один или два параметра, в зависимости от функционального интерфейса и возвращает логическое значение true или false. Посмотрим, как это работает на простом примере – проверим не является ли передаваемая строка пустой:

Изображение выглядит как текст

Автоматически созданное описание

Как и в предыдущем пункте при работе с **Bi**-формой нам требуется передавать 2 параметра:

Изображение выглядит как текст

Автоматически созданное описание

Вывод в консоль будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

Т.е. в данном случае мы проверяем что в нашей строке содержится передаваемое число, которое мы предварительно преобразуем в строку.

**4. Function (BiFunction).**

**Function** используется для преобразования входного параметра или двух параметров (для **Bi**-формы этого функционального интерфейса) в какое-либо значение, тип возвращаемого значения может не совпадать с типом входных параметров. Объявляются интерфейсы следующим образом:

Изображение выглядит как текст

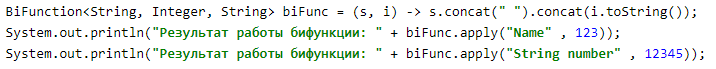
Автоматически созданное описание

Попробуем прочитать символ с определенным индексом в строке с помощью функционального интерфейса:

Изображение выглядит как текст

Автоматически созданное описание

Обратим внимание, несмотря на то что в лямбда выражение передается один параметр, в обобщенных типах мы должны указать 2 параметра – вторым параметром мы указываем тип возвращаемого значения (в нашем случае это **Character**). Давайте попробуем воспользоваться возможностями **Bi**-формы функционального интерфейса, при этом типы принимаемых значений не будут совпадать:



Вывод в консоль будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

Очень важно понимать, что все три типа в обобщениях в **BiFunction** могут быть разными: принимает на вход 2 разных типа и возвращает третий тип.

**5. UnaryOperator и BinaryOperator.**

**UnaryOperator** и **BinaryOperator** – это разновидность **Function**, в которых входные и выходные обобщенные параметры должны совпадать. Давайте посмотрим, как они объявлены в пакете java.util.function:

Изображение выглядит как текст

Автоматически созданное описание

Несложно заметить, что эти два интерфейса расширяют **Function** и **BiFunction** соответственно. Иначе говоря, использование этих интерфейсов сводится к применению **Function** и **BiFunction** в таких методах:



Используем функциональный интерфейс **BinaryOperator**, рассмотрим на примере объединения двух строк:

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

Вывод в консоль будет следующим

Изображение выглядит как текст

Автоматически созданное описание

Обратите внимание, что, несмотря на три параметра (два входных и один для возвращаемого значения) в обобщении мы указываем только один тип – *StringBuilder*. Все потому, что, как говорилось ранее, для функциональных интерфейсов **UnaryOperator** и **BinaryOperator** обобщенные типы должны совпадать и указывать три одинаковых типа обобщения не имеет смысла.

Важно отметить, что помимо абстрактных методов функциональные интерфейсы могут иметь целый набор статических методов или методы по умолчанию. Ниже по ссылке вы можете найти описание встроенных функциональных интерфейсов в документации:

<https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/function/package-summary.html>

* + 1. Встроенные функциональные интерфейсы.

Частым способом применения функциональных интерфейсов является их применение в шаблонах проектирования при рефакторинге нашего кода. В нашем курсе мы уже рассматривали такие шаблоны как синглтон, фабрика, декоратор и стратегия. В данной задачей мы рассмотрим, как можно использовать шаблон проектирования стратегия в сочетании с функциональными интерфейсами.

Давайте вспомним что такое шаблон проектирования стратегия. Стратегия – это поведенческий шаблон проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс. После этого алгоритмы можно взаимозаменять прямо во время выполнения программы. Т.е. данный шаблон предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в отдельные классы, называемые стратегиями.

Рассмотрим следующий пример – у нас есть класс, который работает со строками:

Изображение выглядит как текст

Автоматически созданное описание

Т.е. у нас есть класс и в нем определены 3 метода, которые выполняют проверки над строкой. Мы можем заметить, что у всех методов есть общее – все три метода возвращают один и тот же тип – boolean. Мы помним, что метод **test** интерфейса **Predicate** возвращает тип boolean – давайте перепишем наш код, создав метод, который будет принимать **Predicate** и выполняет проверки с нашими строками:

Изображение выглядит как текст

Автоматически созданное описание

Выходит, что ранее существовавшие методы мы можем просто удалить, а наш универсальный метод **check** - вызывать с разными экземплярами **Predicate** – сделаем это и будем вызывать наш метод **check()** в методе main:

Изображение выглядит как текст

Автоматически созданное описание

вывод в консоль в данном случае будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

Т.е. у нас получается всего один метод, и мы его можем вызывать с разными предикатами. И в данном случае это позволило избежать дублирования кода.

Представьте, что от заказчика пришло новое требование – у нас должен существовать набор методов, которые будут выполнять преобразования строк:

1. Приводить все символы строки к верхнему регистру;

2. Производит конкатенацию двух строк;

3. Удаляет из строки все пробелы в начале и конце строки.

Вспомним, что для того, чтобы выполнить преобразования над данными прекрасно подходит функциональный интерфейс **Function**. В данном случае мы работаем со строками, и на выходе работы методов мы будем также получать строку – значит использовать будем интерфейс в таком виде – **Function<String, String>**:

Изображение выглядит как текст

Автоматически созданное описание

и вызывать метод с разными экземплярами **Function**:

Изображение выглядит как текст

Автоматически созданное описание

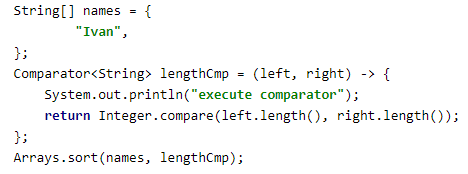
Вывод в консоль в данном случае будет следующим:

Изображение выглядит как текст, внутренний, снимок экрана, серебряный

Автоматически созданное описание

Т.е. все работает корректно. При этом у нас отсутствует дублирование кода, и мы реализуем всего один метод.

* + 1. Ленивая загрузка.

Лямбда обладает свойством отложенного выполнения. 

Что будет выведено на консоль? Ничего. Хотя мы и добавили отладочную информацию в лямбда, она не будет выведена. Лямбда вычисляется, когда вызывается метод у функционального интерфейса. В нашем примере это метод **compare**. Он не был вызван, так как массив содержит одно значение. Сортировать тут не нужно.

Добавим второе значение в массив и запустим код.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

В этом случае вычисление выполнено.

Это свойство увеличивает скорость программы. Практическое применение этого свойства будет в разделе Stream API.

* + 1. Ссылки на методы.

С лямбдами в Java связано еще одно упрощение.

Нам нужно вывести несколько элементов списка. Это можно сделать через метод forEach().

Изображение выглядит как текст

Автоматически созданное описание

Метод **forEach** принимает объект Consumer, который описан через лямбду.

Интерфейс java.function.Consumer имеет абстрактный метод.



Лямбда, которую мы описали, вызывает метод println(...)



Отличие этих методов только в имени. Если вся лямбда — это вызов другого метода, то такой код можно упростить.

Было.



Стало.



Мы убрали аргументы из лямбды и из вызываемого метода.

Вызов метода println идет через двойное двоеточие (::), а не через точку (.).

Рассмотрим пример с компаратором для **Integer**.



У Comparator есть метод compare:



У класса **Integer** есть метод:



Выражение можно упростить.

Было:



Стало:



* + 1. Ссылки на конструкторы.

Аналогично ссылкам на методы, с которыми мы познакомились, ссылки также можно объявлять и на конструкторы. Ссылки на конструкторы создаются так же, как и на методы:



Такую ссылку можно присвоить любой ссылке *на функциональный интерфейс*, где определяется метод, совместимый с конструктором. При этом "ИмяКласса" не может быть интерфейсом или абстрактным классом.

Рассмотрим простой пример использования ссылки на конструктор. Создадим функциональный интерфейс, в котором будет определен единственный метод function(), принимающий строку как параметр:

Изображение выглядит как текст

Автоматически созданное описание

Далее создадим класс Model, где определим 2 конструктора – конструктор по умолчанию и конструктор с такой же сигнатурой, как в нашем функциональном интерфейсе FuncInterface:

Теперь продемонстрируем создание ссылки на конструктор. Создадим класс ConstructorRefMain:

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

В этой строке



Вместо реализации интерфейса создается ссылка на конструктор **Model(String str){}**, так как метод **function(String s)** так же принимает единственный параметр - строку. Переменную **modelConstructor** можно использовать для создания объекта класса **Model**, то есть это еще один способ вызвать конструктор **Model(String str){}** этого класса.

В следующей строке



происходит вызов конструктора класса Model с передачей параметра-строки с помощью переменной **modelConstructor**.

**Документация:**

[Method References](https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html) (Раздел "Reference to a constructor")

* + 1. Функции высшего порядка.

Рассмотрим тему функции высшего порядка. Функции высшего порядка — это функции, зависящие от других функций.

Рассмотрим функцию.



Это функция высшего порядка. Чтобы вычислить эту функцию, сначала нужно вычислить функцию ln(x). Функция y зависит от функции ln.

В программировании функции высшего порядка описываются через композицию. Напомню. Композиция - объект А не может существовать без объекта B. Автомобиль и колеса — это композиция. Шаблон "Декоратор" - тоже композиция.

Рассмотрим создание комбинированного компаратора. Чтобы вычислить значение combine, нужно сначала вычислить значение compareName, а потом comparePriority.

Изображение выглядит как текст

Автоматически созданное описание

Объект combine — это функция высшего порядка.

* + 1. FP - функциональное программирование.

Для того, чтобы разобраться, что такое функционально программирование для начала познакомимся с таким понятием декларативное программирование и императивное.

Программную реализацию системы можно рассмотреть с двух точек зрения:

1. Сосредоточиться на способе выполнения (сделать то, после этого обновить это и т.д.). Например, нам необходимо вычислить наиболее ресурсозатратную транзакцию из списка, выполняется последовательность команд(извлечь транзакцию из списка и сравнить ее с предыдущей наиболее ресурсозатратной транзакцией: если данная транзакция является наиболее ресурсозатратной, запоминаем ее в качестве текущей наиболее ресурсозатратной; повторяем эти действия со следующей транзакцией в списке и т.д.)

Такой стиль отвечает на вопрос "как?", и отлично подходит для классического ООП (в литературе можно встретить понятие императивное программирование), поскольку такой стиль включает инструкции, имитирующие низкоуровневый лексикон компьютера (операции присваивания, условные ветвления, циклы).

Если упростить, то мы говорим компьютеру как ему работать. Даем ему последовательные инструкции: принеси, положи, подай.

2. Второй подход концентрируется на том, что нужно получить, то есть на результате, а не на инструкциях.

Например, в задании 3. Банковские переводы мы делали такой запрос карту для поиска студента по его данным:

Первый вариант сделан через циклы в императивном стиле, то есть мы говорим компьютеру, что нужно выполнить цикл, если элемент подходит, то цикл остановить и выйти.

Изображение выглядит как текст

Автоматически созданное описание

Второй вариант сделан в декларативном стиле. Мы переписали его на Stream API. Здесь мы говорим только о результате работы, если есть такой элемент, то верни его. Здесь нет указания на циклы и проверку условий.

Изображение выглядит как текст

Автоматически созданное описание

Циклы, проверка условия, поиск скрыты в библиотека Stream API.

Подобный стиль, отвечающий на вопрос "что?", часто называют декларативным программированием. Мы лишь задаем правила, описывающие, что нам нужно, а система должна сама определить, как достичь этой цели. Достоинство подобного стиля программирования - в том, что формулировка задачи сразу очевидна из кода.

Пример декларативного языка — это SQL.

Так вот, функциональное программирование служит примером идей декларативного программирования (например, использование не взаимодействующих между собой выражений, реализацию для которых выбирает система) и вычислений без побочных эффектов (модификация структуры данных без создания дополнительных структур, генерация исключения, выполнение операций ввода/вывода). Такие две идеи упрощают реализацию и сопровождение систем.

Так что же такое функциональное программирование?? Самый простой ответ звучит следующим образом - "Это программирование с помощью функций". Однако, что в данном контексте означает понятие "функция"?

Рассмотрим 2 варианта:

1. Представляем себе метод, который принимает в качестве аргументов значения типов int и double и возвращает тип double с побочным эффектом подсчета числа вызовов путем обновления изменяемой переменной:

Изображение выглядит как текст, внутренний

Автоматически созданное описание­­­­

2. В контексте функционального программирования термин функция понимается в математическом смысле: она принимает ноль или более аргументов возвращает один или более результатов и не имеет побочных эффектов. Функцию можно рассматривать как "черный ящик", получающий какие-либо входные данные и выдающий какие-то результаты, как показано ниже:



Такие функции и методы в Java отличаются разительно (сложно представить себе, что у таких математических функций как log и sin могут побочные эффекты). Математические функции всегда возвращают один и тот же результат, будучи вызванными с одними аргументами.

Говоря "функциональный", мы подразумеваем "подобный математическому", без побочных эффектов. Но вопрос - что имеется в виду в связке с понятиями программирования: имеется ли в виду, что все функции должны быть основаны на функциональных и математических идеях? Или внутри функции могут выполняться нефункциональные действия при условии, что они не оказывают влияние на остальную систему? Т.е., если программа производит невидимый для вызывающей стороны побочный эффект, то можно ли считать, что этого побочного эффекта нет? Вызывающей стороне о нем знать необязательно, поскольку он никак на нее не влияет.

И вот тут выделяется 2 понятия:

1. Чисто функциональное программирование - мы строим структуры и элементы наших программ, которые обрабатывают все вычисления в виде математических функций. Чисто функциональное программирование заключается в обеспечении того, чтобы функции внутри функциональной парадигмы зависели только от своих аргументов, независимо от какого-либо глобального или локального состояния. Чисто функциональное программирование.

2. Просто функциональное программирование или программирование в функциональном стиле.

Нас интересует больше второе понятие. Рассмотрим это понятие в применении к языку Java.

На практике невозможно программировать в чисто функциональном стиле на языке Java. Тем не менее вполне возможно написать основные компоненты системы так, как будто они чисто функциональные. Мы будем часто писать код наших программ в функциональном стиле.

Во-первых, существует нюанс относительно "невидимости" побочных эффектов и, соответственно, смысла слова «функциональный». Пусть у метода или функции нет побочных эффектов, за исключением увеличения на единицу значения определенного поля при входе в функцию и уменьшения после выхода. С точки зрения однопоточной программы у этой функции нет видимых побочных эффектов, ее можно считать функциональной. С другой стороны, если к этому полю может обращаться другой поток выполнения — или метод может вызываться конкурентно, — то считать его функциональным нельзя. Эту проблему можно скрыть с помощью адаптера с блокировкой для тела этого метода, что позволит утверждать, что он является функциональным. Но при этом мы потеряем возможность параллельного выполнения двух вызовов метода двумя ядрами многоядерного процессора. Побочный эффект может быть незаметен программе, но он проявляется для программиста в виде снижения быстродействия.

Рекомендуется считать функциональными только функции/методы, которые изменяют лишь значения локальных переменных. Кроме того, должны быть неизменяемыми и объекты, на которые они ссылаются, — то есть все поля должны быть final, а все поля ссылочного типа должны ссылаться на другие неизменяемые объекты. Далее можно разрешить модификацию полей объектов, только что созданных в данном методе, поскольку они ниоткуда не видимы и не сохраняются, так что не могут повлиять на последующие вызовы этого метода.

Однако, вышеприведенные рекомендации неполны. Чтобы быть функциональными, функция/метод должны удовлетворять дополнительному требованию: не генерировать никаких исключений. Обосновывается это требование тем, что генерация исключения означает сигнал о результате выполнения функции, отличный от возврата значения.

Чтобы считаться функциональной, функция (метод) должна вызывать только те библиотечные функции с побочными эффектами, которые позволяют скрыть нефункциональное поведение (то есть гарантировать, что любые изменения структур данных скрыты от вызывающей стороны, например, с помощью предварительного их копирования и путем перехвата всех исключений).

Надеемся, что в общих чертах Вам ясна идея функционального программирования. Укажем лишь в чем плюсы и минусы его использования:

**Плюсы:**

*- повышение надежности кода.*

Это достигается за счёт чёткой структуризации и отсутствия необходимости отслеживания побочных эффектов. Любая функция работает только с локальными данными и работает с ними всегда одинаково, независимо от того, где, как и при каких обстоятельствах она вызывается.

*- удобство организации модульного тестирования.*

Поскольку функция в функциональном программировании не может порождать побочные эффекты, менять объекты нельзя как внутри области видимости, так и снаружи. Единственным эффектом от вычисления функции является возвращаемый ей результат, и единственный фактор, оказывающий влияние на результат — это значения аргументов.

Таким образом, имеется возможность протестировать каждую функцию в программе, просто вычислив её от различных наборов значений аргументов. При этом можно не беспокоиться ни о вызове функций в правильном порядке, ни о правильном формировании внешнего состояния. Если любая функция в программе проходит модульные тесты, то можно быть уверенным в качестве всей программы.

*- возможности оптимизации при компиляции.*

Функциональное программирование позволяет описывать программу в так называемом «декларативном» виде, когда жесткая последовательность выполнения многих операций, необходимых для вычисления результата, в явном виде не задаётся, а формируется автоматически в процессе вычисления функций. Это обстоятельство, а также отсутствие состояний даёт возможность применять к функциональным программам достаточно сложные методы автоматической оптимизации.

*- возможности параллелизма.*

Ещё одним преимуществом функциональных программ является то, что они предоставляют широчайшие возможности для автоматического распараллеливания вычислений. Поскольку отсутствие побочных эффектов гарантировано, в любом вызове функции всегда допустимо параллельное вычисление двух различных параметров — порядок их вычисления не может оказать влияния на результат вызова.

**Недостатки:**

*- некоторое падение производительности приложений.*

Отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный сборщик мусора.

*- проблемы, которые могут возникнуть из-за порядка вызова функций.*

С этим связано такое понятие, как ленивая загрузка (вы уже познакомились с этим ранее). Нестрогая модель вычислений приводит к непредсказуемому порядку вызова функций, что создаёт проблемы при вводе-выводе, где порядок выполнения операций важен.

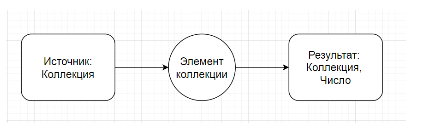
* 1. **Stream API.**

В JDK 1.8 ввели интерфейс Stream.

- Этот интерфейс позволяет гибко работать с коллекциями.

- Stream API работает совместно с лямбда-выражениями.

- Stream API — это поток данных.



**Важно!** Основа написания кода через Stream – перенос каждого метода на новую строку.

Каждый элемент коллекции проходит 3 стадии.

1. Фильтрация;

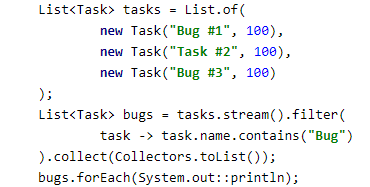
2. Преобразование;

3. Упрощение или аккумуляция.

Каждая стадия может использоваться отдельно или совместно.

**1. Фильтрация.**

Отфильтруем все задачи, где в имени есть слово BUG и сохраним ее в коллекцию List.



Рассмотрим детали:



мы получаем из коллекции Stream или поток элементов.

Каждый элемент передается в метод **filter()**. Если метод возвращает true, то элемент передается дальше.



Схема:



следующая строка:



Позволяет полученный результат сохранить в коллекции типа List.

Давайте посмотрим аналог этого решения через for-each:

Изображение выглядит как текст

Автоматически созданное описание

Если провести аналогию между циклами и **stream**, то получим:

**for()** - это stream().

**if()** - это Predicate.

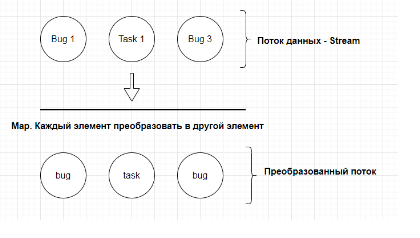
**2. Преобразование.**

Допустим, что нам нужно получить только имена задач. Для этого нужно применить метод **map()**.

Изображение выглядит как текст

Автоматически созданное описание

Cхема:



**3. Упрощение.**

Давайте теперь посчитаем общее время, потраченное на все задачи.

Изображение выглядит как текст

Автоматически созданное описание

Метод **reduce()** в следующей строке складывает каждое значение **task.spent** с начальным значением - 0L.

Схема.



**4. Аккумуляция.**

Аккумуляция — это запись элементов потока в коллекцию.

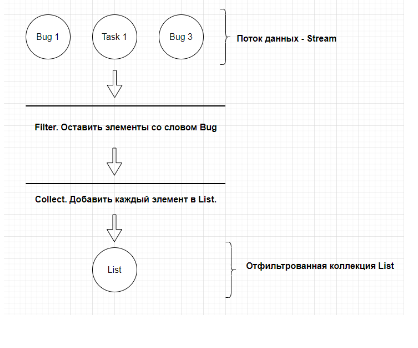
Пример.

Изображение выглядит как текст

Автоматически созданное описание

Прошедшие фильтрацию элементы записываются в коллекцию List.

Схема.



Stream:

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>

* + 1. Типы методов Stream API.

В Java Stream API представлены 2 типа методов:

- *Промежуточные (конвейерные)*. Преобразовывают элементы потока, возвращая новый преобразованный поток. Методов данного типа может быть сколько угодно в цепочке преобразований элементов потока. Данные методы ленивы, то есть отрабатывают только когда для потока вызван конечный метод. Промежуточных методов у потока может и не быть. Промежуточные операции не выполняются без конечных! Промежуточные методы ленивы - они начинают вычисляться только когда начнется терминальная операция, то есть вычисление происходит только тогда, когда оно нужно.

- *Конечные (терминальные).* Метод данного типа всегда один, располагается в конце цепочки промежуточных методов (если они есть). Данный метод возвращает другой тип объекта (например, *Optional*, коллекцию и т.д.). То есть конечный метод собирает результаты обработки элементов потока и возвращает единый результат. Конечный метод для завершения потока обязателен.

Ниже перечислены основные промежуточные и конечные методы с коротким описанием. Далее в этом разделе многие из этих методов мы разберем на практических примерах.

**Промежуточные (конвейерные) методы:**

- **filter()** - фильтрует элементы потока, возвращая только элементы, удовлетворяющие условию.

- **map()** - преобразует каждый элемент потока.

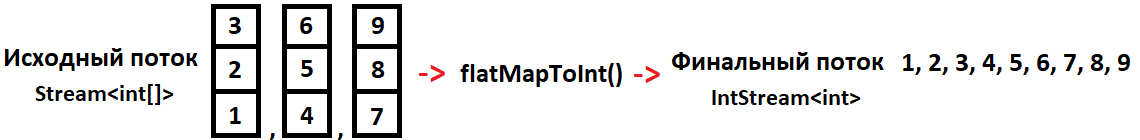
- **mapToInt()** - тот же **map()**, но возвращает поток примитивов int (также есть соответствующие **mapToDouble()** и **mapToLong()**).

- **peek()** - применяет функцию Consumer к каждому элементу потока.

*Также метод* ***peek()*** *похож на метод* ***map()****. Разница между этими методами в том, что* ***peek()*** *просто преобразует элементы потока и возвращает поток из измененных элементов того же типа, а метод* ***map()*** *в процессе преобразования элементов может возвращать поток из элементов другого типа после изменения. Поэтому метод* ***peek()*** *чаще используется в отладочных целях для вывода состояния элемента в потоке, нежели для каких-либо преобразований.*

- **flatMap()** - может преобразовывать один элемент в несколько (также поддерживает возврат потоков примитивов с помощью методов **flatMapToInt()**, **flatMapToDouble()**, **flatMapToLong()**).

*Пример:*

**

- **sorted()** - сортирует элементы потока по возрастанию. Возможна сортировка по убыванию при передаче соответствующего компаратора.

- **skip()** - пропускает указанное число элементов с начала потока.

- **limit()** - делает выборку первых элементов из родного потока в указанном количестве (отбирает элементы из потока, пока не достигнет указанного количества).

- **distinct()** - убирает дубликаты из потока.

- **mapToObj()** - трансформирует числовой поток в объектный.

**Конечные (терминальные) методы:**

- **forEach()** - применяет функцию к каждому элементу потока.

- **collect()** - собирает все элементы потока в структуру данных.

- **toArray()** - собирает элементы потока в массив.

- **count()** - возвращает количество элементов в потоке.

- **min()** - возвращает минимальный элемент (условие передается в компараторе).

- **max()** - возвращает максимальный элемент (условие передается в компараторе).

- **sum()** - возвращает сумму всех элементов потока (только для числовых потоков).

- **average(**) - возвращает среднее арифметическое всех элементов потока (только для числовых потоков).

- **allMatch()** - возвращает true, если все элементы удовлетворяют условию.

- **noneMatch()** - возвращает true, если все элементы не удовлетворяют условию.

**Конечные (терминальные) методы, делающие обход потока только до нахождения элемента, удовлетворяющего условию:**

- **findFirst()** - возвращает *Optional* с первым элементом потока (если он есть), иначе возвращает пустой *Optional*.

- **findAny()** - возвращает *Optional* со случайным элементом потока.

- **anyMatch()** - возвращает true, если хотя бы один элемент удовлетворяет условию.

**Преобразование List в Map.**

Карты являются наиболее удобной структурой данных. Часто возникают ситуации, когда необходимо выполнить преобразование списка в карту.

Для этого нужно использовать **collect** c аргументом **Collectors.toMap**.

Пример для списка чисел.

Изображение выглядит как текст

Автоматически созданное описание

Вывод в консоль будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

В следующей части кода:

Изображение выглядит как текст

Автоматически созданное описание

- ***e*** — это элемент списка.

- Первый аргумент метода — это ключ карты.

- Второй аргумент метода — это значение. В качестве примера в значение карты мы записываем квадрат элемента ключа.

- Третий аргумент позволяет обработать вариант добавления элемента с дублирующимся ключом. В данном случае мы оставляем существующее значение, хотя это поведение можно и изменить.

**Выборка элементов из потока.**

В потоке нет индексов. Процесс работы потока — это последовательный перебор всех элементов указанной структуры данных с применением к ним промежуточных операций после того, как запущена конечная операция. Из-за отсутствия индексов мы не можем таким образом выборочно работать с элементами, но в **Java Stream API** существуют методы, которыми можно вручную ограничить диапазон перебираемых элементов - **skip()** и **limit()**.

- **skip(n)** - промежуточная операция, пропускает первые n элементов с начала потока. То есть задает начальную границу диапазона перебираемых элементов. Параметром нельзя задать отрицательное число, а если заданное значение параметра равно или превышает число элементов в исходной структуре данных, то будет возвращен пустой поток.

Изображение выглядит как текст

Автоматически созданное описание

В строке ниже метод **skip(2)** пропускает 2 первых элемента с начала потока, возвращая новый поток, состоящий из оставшихся элементов.

При выводе видим, что первые 2 элемента были отсеяны:



**Важно!** Числа, передающиеся в параметрах методов потоков, начинают отсчет с 1, так как исчисляется количество элементов. Не путать с индексом, где нумерация начинается с 0.

- **limit(n)** - промежуточная операция, возвращает новый поток, содержащий только первые n элементов из исходного потока, то есть задает конечную границу диапазона перебираемых элементов. Нельзя задать отрицательное значение параметра. Этот метод перебирает элементы, пока не накопит указанное количество элементов, после чего завершает перебор исходной структуры данных и возвращает новый поток из собранных элементов.

Изображение выглядит как текст

Автоматически созданное описание

В строке ниже метод **limit(3)** оставляет только 3 первых элемента из исходного потока:

При выводе видим, что остались только 3 первых элемента:

Изображение выглядит как текст

Автоматически созданное описание

Получение первого элемента из потока.

- **findFirst()** - терминальная операция, получает первый элемент из потока.

Данный пример аналогичен предыдущему, но терминальной операцией в нем является метод **findFirst()**, который получает первое значение из получившегося после исполнения методов **skip()** и **limit()** потока. Метод **findFirst()** возвращает значение, обернутое в тип *Optional*. *Optional* — это просто обертка для значения, чтобы не возвращать *null* при отсутствии значения. Подробно *Optional* разберется позже.

В строке ниже метод **Optional.orElse()** проверяет возвращенное методом **findFirst()** значение типа *Optional* на его наличие, и если оно есть, то возвращает его, а если его нет, то возвращает значение, указанное по умолчанию. Обратите внимание, что терминальная операция в этом примере - метод **findFirst()**, в результате чего возвращается значение в типе *Optional*, из которого мы сразу же достаем значение с помощью метода **orElse()**, то есть сам поток фактически завершается после исполнения метода **findFirst()**.

Вывод:



**Методы преобразования потоков объектов в потоки примитивы и наоборот.**

Познакомимся с методами преобразования потока объектов в поток примитивов (числовой поток) и преобразования потока примитивов обратно в поток объектов.

Рассмотрим методы **mapToInt()**, **flatMapToInt()** и **mapToObj()**.

- **mapToInt()** - преобразовывает поток объектов в поток примитивных чисел типа *int*. Является промежуточной операцией. Применяется, если исходящий поток состоит из элементов-объектов, а в результате их обработки будет получен примитивный тип int. Метод **mapToInt()** возвращает объект интерфейса *IntStream* (числовой поток), который в дальнейшем обрабатывается как поток примитивных чисел.

Изображение выглядит как текст

Автоматически созданное описание

В строке ниже метод **mapToInt()** преобразовывает объекты типа *String* в примитивное значение *int*, создавая новый поток *int (IntStream)*:

Интерфейс *IntStream* описывает поток примитивных чисел *int*. То есть это такой же поток, как и Stream, только из int.

В результате отбора фильтром только четных чисел, получается такой вывод:



Рассмотрим более сложный пример.

Изображение выглядит как текст

Автоматически созданное описание

создадим класс *MapToIntMain*, в котором демонстрируется отбор людей с возрастом более 25 лет с последующим выводом суммы их возрастов:

Изображение выглядит как текст

Автоматически созданное описание

В строке ниже метод **mapToInt()** преобразует поток *Person* в поток *int*:



Далее суммируем элементы полученного числового потока(Метод **sum()** возвращает только тип *int*):



Вывод:

Изображение выглядит как текст

Автоматически созданное описание

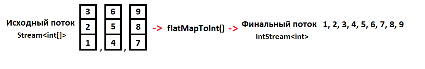
Класс ***Stream*** помимо метода **mapToInt()** также имеет аналогичные ему методы **mapToDouble()** и **mapToLong()**, работающие с *Double* и *Long* типами соответственно.

- **flatMapToInt()** - трансформирует поток массивов в поток примитивных чисел int. Работает как **mapToInt()**, но возвращает не одно преобразованное в *int* значение, а поток примитивов *int*. Является промежуточной операцией.

Разберем этот пример построчно. В строке ниже мы запускаем поток, состоящий из 3 массивов:



Далее метод **flatMapToInt()** преобразует каждый элемент потока (массив) в массив примитивных чисел *int*. Визуализируем с помощью картинки:



То есть метод **flatMapToInt()** последовательно запускает поток из каждого массива исходящего потока, в результате чего все элементы этих массивов объединяются в один общий поток примитивных чисел *int*.

Как мы знаем, примитивные числа не являются объектами, поэтому для того, чтобы собрать в *List* наш поток элементов типа *int*, нужно обернуть их в тип *Integer*. Это можно сделать с помощью метода **boxed()**, что и выполняется в следующей строке:



Вывод:



Класс *Stream* помимо метода **flatMapToInt()** также имеет аналогичные ему методы **flatMapToDouble()** и **flatMapToLong()**, работающие с *Double* и *Long* типами соответственно.

- **mapToObj()** - преобразует поток примитивных чисел в поток объектов. Является промежуточной операцией.

Изображение выглядит как текст

Автоматически созданное описание

В строке ниже создается поток из примитивных чисел int от 1 до 5 включительно:



В следующей строке метод **mapToObj()** преобразует все элементы потока примитивного тип *int* в строки типа *String*.



Вывод листа строк:



**Агрегатные методы min(), max(), count(), average(), sum().**

Познакомимся с агрегатными методами потоков - **min()**, **max()**, **count()**, **average()** и **sum()**.

Агрегация — это вычисления, строящиеся на всех элементах исходной структуры данных, в результате которых получается единое значение. Например, среднее арифметическое, сумма чисел и т.д. Агрегацию еще называют сведением. Все эти методы являются терминальными и сводят все элементы потока к одному значению.

- **min()** - возвращает минимальный элемент потока. Наименьший элемент определяется с помощью компаратора, который передается в параметр метода.

Изображение выглядит как текст

Автоматически созданное описание

В строке ниже метод **min()** вычисляет минимальное значение с помощью переданного нами компаратора сравнения объектов по возрастанию.

Компаратор нужно передавать, чтобы метод понимал по какому принципу сравнивать объекты. Если метод **min()** применяется к потоку примитивных чисел, то можно использовать его без компаратора:

Изображение выглядит как текст

Автоматически созданное описание

В данном примере мы запустили поток примитивных чисел типа *int* с помощью метода **rangeclosed()** и нашли минимальный из них с помощью метода **min()** без передачи компаратора, так как он не нужен для арифметических операций над примитивными типами. Обратите внимание, что метод **min()** здесь возвращает тип *OptionalInt*. *Optional* — это обертка для значения, чтобы в случае его отсутствия вернулся не *null*, а пустая обертка *Optional*. В нашем случае *OptionalInt* — это обертка для примитивного типа int, из этой обертки мы получаем само значение с помощью метода **getAsInt()**.

- **max()** - возвращает максимальный элемент потока. Наибольший элемент определяется с помощью компаратора, который передается в параметр метода. Работает аналогично методу **min()**.

- **count()** - возвращает число элементов в потоке.

- **average()** - возвращает среднее арифметическое всех элементов потока. Применяется только к числовым потокам. Метод возвращает *OptionalDouble* (Значение *double*, обернутое в *Optional*).

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

- **sum()** - суммирует все элементы потока. Применяется только к числовым потокам.

**Агрегатный метод reduce().**

Метод **reduce()** применяет операцию ко всем элементам потока. В этом методе понятие агрегации обобщается, так как в нем можно самостоятельно задать критерий для получения значения. Метод является *терминальным*.

Метод **reduce()** имеет 3 формы, каждую из которых мы рассмотрим в этом уроке.

Изображение выглядит как текст

Автоматически созданное описание

**reduce(BinaryOperator<T> accumulator)** - стандартный вариант метода **reduce()**. Применяет функцию *BinaryOperator* ко всем элементам потока. Метод работает так: например, есть поток из чисел 1, 2, 3, 4. Нам нужно сложить все эти числа. Метод **reduce()** сначала сложит первые 2 элемента - (1 + 2 = 3), далее к этому результату будет прибавлен следующий элемент (3 + 3 = 6) и так далее, пока не будут сложены все элементы потока. В результате весь поток будет сведен к единому результату. В нашем случае это сумма всех чисел. Данная форма метода возвращает *Optional<T>* (значение в обертке).

Изображение выглядит как текст

Автоматически созданное описание

В строке ниже мы передаем *BinaryOperator* в метод **reduce()**.



*(a, b)* — это первые 2 элемента потока, *a + b* — это действие, которое нужно применить к этим элементам. На первом шаге эта операция будет выглядеть так: (1, 2) -> 1 + 2. Сумма этих элементов становится первым элементом функции на следующем шаге: (3, 3) -> 3 + 3 и так далее, пока метод не сложит все элементы потока в единую сумму.

Вывод:



**- reduce(T identity, BinaryOperator<T> accumulator)** - эта версия метода похожа на предыдущую, только в *BinaryOperator* первым параметром будет *identity* - значение, к которому будет применяться второй элемент в функции. Если поток будет пуст, то *identity* останется значением по умолчанию. Данная форма метода возвращает само значение (T).

Изображение выглядит как текст

Автоматически созданное описание

В строке ниже мы так же передаем *BinaryOperator* в метод **reduce()**. На этот раз первым параметром будем выступать указанный *identity*, который складывается с первым элементом *(a + b) -> identity + b*, на следующем шаге первым параметром в *BinaryOperator* будет сумма этих элементов, которая будет складываться уже со вторым элементом потока, дальше все шаги аналогичны предыдущей форме метода **reduce()**.

**- reduce (U identity, BiFunction<U, ? super T,U> accumulator, BinaryOperator<U> combiner)** - расширенная версия второй формы. *accumulator* здесь позволяет выполнить промежуточное действие над элементами потока, после чего к ним будет применена агрегатная операция *combiner*. Данная форма метода аналогична комбинации методов **map()** и второй формы **reduce()**, то есть в этой форме метода можно возвращать значение, отличного по типу от исходного.

Вернемся к примеру из предыдущего урока по агрегатным методам, посчитаем через метод **reduce()** сумму возрастов людей, которым более 25 лет.

Изображение выглядит как текст

Автоматически созданное описание

В строке ниже задаем первый параметр метода **reduce()** - *identity*, равный 0, так как суммировать будем с нуля. Если в потоке не будет элементов, заданный *identity* будет выведен как значение по умолчанию.

Изображение выглядит как текст

Автоматически созданное описание

Следующий блок кода описывает второй параметр метода **reduce()**, здесь мы получаем из объектов *Person* промежуточное значение - возраст людей, которым более 25 лет. На этом шаге будет создан новый поток с отобранными по условию числами 35, 26 и 29.

Изображение выглядит как текст

Автоматически созданное описание

В следующей строке выполняем бинарную операцию по суммированию всех элементов. На этом шаге складываются числа из предыдущего шага:



Вывод:



Таким образом, с помощью этих трех форм метода **reduce()** мы можем самостоятельно задавать критерий для получения значения, в отличие от методов **sum()**, **average()** и тд, которые возвращают значение конкретных функций.

**Документация:**

[Stream.reduce(java.util.function.BinaryOperator)](https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/stream/Stream.html#reduce(java.util.function.BinaryOperator)).

[Stream.reduce(T,java.util.function.BinaryOperator)](https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/stream/Stream.html#reduce(T,java.util.function.BinaryOperator)).

[Stream.reduce(U,java.util.function.BiFunction,java.util.function.BinaryOperator)](https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/stream/Stream.html#reduce(U,java.util.function.BiFunction,java.util.function.BinaryOperator)).

**Методы проверки элементов потока на соответствие условию.**

Познакомимся с методами проверки элементов на соответствие заданному условию - **noneMatch()**, **anyMatch()** и **allMatch()**.

Все эти методы принимают *Predicate* в качестве параметра (условия для проверки). Также все 3 метода являются *конечными (терминальными).*

- **noneMatch()** - возвращает true, если ни один из элементов потока не удовлетворяет условию.

Изображение выглядит как текст

Автоматически созданное описание

Метод **noneMatch()** возвращает true, так как такое значение в нашем листе отсутствует, иначе вернул бы false:

- **anyMatch()** - возвращает true, если как минимум один из элементов потока удовлетворяет условию.

Изображение выглядит как текст

Автоматически созданное описание

Метод **anyMatch()** возвращает true, так как такое значение в нашем листе присутствует ("Пять"), иначе вернул бы false:

- **allMatch()** -возвращает true, если все элементы потока удовлетворяют условию.

Изображение выглядит как текст

Автоматически созданное описание

Метод **allMatch()** возвращает true, так как все значения в нашем листе начинаются с "Три", то есть удовлетворяют условию, иначе было бы возвращено false:

**Документация:**

[Stream.noneMatch()](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#noneMatch(java.util.function.Predicate)).

[Stream.anyMatch()](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#anyMatch(java.util.function.Predicate)).

[Stream.allMatch()](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#allMatch(java.util.function.Predicate)).

* + 1. Шаблоны проектирования в Stream API.

Если мы посмотрим на интерфейс Stream, то в нем определен вложенный интерфейс *Builder*, который аннотирован следующим *JavaDoc* комментарием:

Изображение выглядит как текст

Автоматически созданное описание

Если обратимся к официальной документации, то данный интерфейс описывается следующим образом:

<https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/stream/Stream.Builder.html>

И вот мы уже в самом начале получаем ответ - для построения потоков при работе со Stream используется шаблон проектирования *Builder* (в переводе на русский - Строитель). Разберемся, что это за шаблон, как он реализуется и для чего может быть применен.

**Строитель** — это порождающий шаблон проектирования, который позволяет создавать сложные объекты пошагово. **Строитель** позволяет использовать один и тот же код построения объекта для получения разных представлений объектов. Пока звучит очень туманно, но постепенно вы поймете, о чем идет речь.

Например, у нас есть сложный объект, который для создания требует сложной пошаговой инициализации множества полей, и, что еще хуже и сложнее, вложенных объектов. Чтобы проинициализировать такой объект, необходимо использовать очень большой конструктор с множеством параметров, который при использовании клиентом становится абсолютно нечитаемым, и чтобы контролировать вводимые параметры в том месте, где определено их место, требуется постоянно возвращаться к объявленному конструктору. Это определенно неудобно и приводит к проблемам у потребителя такого кода. И еще, например, *checkstyle* по умолчанию не позволяет определять в конструкторе более 7 параметров - поэтому, если мы хотим вводить больше параметров - требуется менять это значение по умолчанию. Как видите, многовато проблем для казалось бы несложной ситуации.

Создадим следующую модель данных - User:

Изображение выглядит как текст

Автоматически созданное описание

Сюда не были добавлены геттеры и сеттеры (это 14 методов), перегруженные конструкторы (вариаций с 7 полями, как вы понимаете, может быть немало), а также методы **equals()** и **hashcode()** - если все это добавить, строк в классе будет в разы больше.

В данной модели мы пошли по пути наименьшего сопротивления - просто создали конструктор, который принимает уйму параметров, который на выходе нам выдаст объект с проинициализированными полями, и при этом все параметры надо контролировать, чтобы поставить на свое место, и чтобы, например, строка, обозначающая пол пользователя, не попала в его имя. А что если нам для потребления объекта не нужно инициализировать все поля - напрашивается очевидный ответ перегрузить конструктор, но так мы можем расплодить бесчисленное количество конструкторов, которые могут потребляться один раз, т.е. простаивать.

Именно для таких случаев используется шаблон **Строитель**. Он предлагает решение такой проблемы путем вынесения конструирования объекта за пределы его собственного класса (или за счет вложенного статического класса), поручив это дело отдельным объектам, которые и называются **строителями**.

Шаблон предлагает разбить процесс конструирования объекта на отдельные шаги (например, для нашего класса - присвоитьИмя, присвоитьВозраст и т.д.). Чтобы создавать объекты таким образом, нам нужно поочередно вызывать методы строителя. При этом нам не нужно запускать все шаги, а только те, которые нужны для производства объекта определенной конфигурации.

Вернемся к нашему классу и реализуем шаблон **Строитель** на его основе. Для этого мы убираем конструктор и добавляем вложенный статический класс *Builder* (такое именование является общепринятой практикой):

Изображение выглядит как текст

Автоматически созданное описание

Копируем в наш вложенный класс поля, которые объявлены внутри класса *User* - и наш вложенный класс приобретет вид.

Следующий этап - добавление методов, которые будут инициализировать поля, объявленные внутри вложенного класса и при этом возвращать тип *Builder* (о чем идет речь, вы поймете ниже из кода):

Изображение выглядит как текст

Автоматически созданное описание

Обратите внимание, что все методы возвращают один и тот же тип *Builder*, и при этом инициализируют одно поле (если сравнивать это с поведением конвейерных методов в *Stream API*, то вы без проблем найдете сходство). При этом имена методов стараются именовать по шаблону - ставится слово *build* и к нему в соответствии с *CamelCase* добавляется имя поля, которое мы инициализируем в этом методе.

Последний элемент реализации шаблона - метод, который построит нам готовый объект и вернет его экземпляр:

Изображение выглядит как текст

Автоматически созданное описание

Метод **build()** мы уже будем вызывать для того, чтобы вернуть готовый объект. Его поведение схоже с терминальными операциями в **Stream API**, которые возвращают результат из потока, а не поток, как это делают конвейерные методы. Давайте попробуем использовать эту реализацию шаблона:

Изображение выглядит как текст

Автоматически созданное описание

Запустим этот код, вывод в консоль будет следующим:

Изображение выглядит как текст

Автоматически созданное описание

При этом, вызывая методы, мы не сможем запутаться, какие поля мы инициализируем, поскольку мы знаем это из имен методов, которые мы вызываем.

Когда стоит использовать шаблон проектирования **Builder (Строитель)**:

1. Когда мы хотим избавиться от конструктора с большим количеством параметров (случай описанный в задании);

2. Когда код должен создавать разные представления какого-то объекта. Например, когда нам надо инициализировать лишь несколько полей из всего набора;

3. Когда нам необходимо собирать сложные объекты, т.е. когда поля в классе являются экземплярами других классов.

**Преимущества использования шаблона Строитель:**

1. Позволяет создавать объекты пошагово;

2. Позволяет использовать один и тот же код для создания различных объектов - если использовать не класс *Builder*, а *interface* с таким именем, тогда можно создавать различные имплементации этого интерфейса (в данном задании мы это не рассматривали);

3. Сложный код сборки объекта от основной бизнес логики будет изолирован.

**Недостатки использования шаблона Строитель:**

1. Усложняет код программы из-за введения дополнительных классов.

* + 1. Дополнительное Java API.

**Stream API.**

**Stream.takeWhile()** - этот метод позволяет получать поток данных до тех пор, пока условие истина.

Пример:

Изображение выглядит как текст

Автоматически созданное описание

Вывод:

Изображение выглядит как текст, снимок экрана, экран, стопка

Автоматически созданное описание

Когда поток обрабатывает число 3 условие принимает значение "ложь" поток данных завершается. Эту конструкцию удобно использовать с отсортированными данными.

**Stream.dropWhile()**- этот метод позволяет получать поток данных после того, как условие стало ложным.

Пример:

Изображение выглядит как текст

Автоматически созданное описание

Вывод:

Изображение выглядит как текст

Автоматически созданное описание

Когда обрабатывается число 3 условие v -> v < 3 становится "истина". Все дальнейшие элементы проходят дальше.

**Stream.ofNullable()** - сделает из элемента поток, если элемент равен **null**, то элемент пропускается. Этот метод позволяет фильтровать поток с **null** элементами.

Изображение выглядит как текст, внутренний, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

**Local-Variable Type Inference.**

**var** — это ограниченный идентификатор, который упрощает создание переменных. Он имеет особое значение как тип объявления локальной переменной то стоит разобраться с этим подробнее.

Пример:



**var** может использоваться только в сочетании с данными.

Изображение выглядит как текст

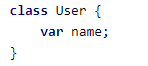
Автоматически созданное описание



Важно, что мы не можем использовать **var** без самого значения, потому что JDK нужно знать тип переменной, а тип можно извлечь только из значения.

**Где нельзя использовать var:**

1. Поля класса.



2. Параметры метода.



3. Переменные без значения.



1. Дополнительно:

**1) Работа с символами:**

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/regex/Pattern.html>.

Пример:

<https://job4j.ru/profile/topics/42/task_code/16/152043>.

**2) Работа с агрегацией информации:**

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/stream/Collector.html>.

Пример:

<https://job4j.ru/profile/topics/3/task_code/192/151516>.

* 1. S.O.L.I.D.
     1. Single responsibility principle.

Single responsibility principle (принцип единой ответственности) говорит нам о том, что нужно стремиться к такому написанию кода, чтобы 1 класс решал одну задачу (Декомпозиция классов).

Пример:

Было:

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

Стало:

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описание

Плюсы:

1. Появились модули, следовательно вырастет читаемость кода.

2. Вносить изменения стало проще.

3. Нет антипаттерна GodObject.

4. Класс инкапсулирует решение одной задачи.

* + 1. Open – closed principle.

Open – closed principle (Принцип открытости/закрытости) – программные сущности должны быть открыты для расширения и закрыты для изменения (Добавление новых сущностей, а не изменение имеющихся).

Пример:

Было:

Изображение выглядит как диаграмма, линия, зарисовка, Прямоугольник

Автоматически созданное описание

Стало:

Изображение выглядит как диаграмма, Прямоугольник, линия, прямоугольный

Автоматически созданное описание

Плюсы:

1. Нет необходимости регрессии (Регрессионное тестирование).

2. Меньше вероятность допущения ошибки.

* + 1. Liskov Substitution principle.

Liskov Substitution principle (Принцип подстановки Барбары Лисков) – функции и сущности, которые работают с родительскими классами должны также работать и с дочерними (не должна нарушаться логика и ничего не должно ломаться). Или : наследуемый класс должен дополнять а не замещать работу базового класса.

Пример:

Изображение выглядит как текст, диаграмма, снимок экрана, Параллельный

Автоматически созданное описание

Учитывая принцип подстановки Барбары Лисков, данное наследование классов является ошибочным, ведь один из дочерних классов не может выполнять базовые инструкции.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Автоматически созданное описание Также неверно.

* + 1. Interface segregation principle.

Interface segregation principle (Принцип разделения интерфейса) – Программные сущности не должны зависеть от методов, которые они не реализуют (Нельзя заставлять пользователя реализовывать интерфейс, которым он не пользуется).

Пример:

Неверно относительно ISP.

Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание

Верно:

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описание

Плюсы:

1. Избавляем программные сущности от методов, которые они не используют.

2. Получаем более предсказуемую работу.

3. Код становится менее связным.

* + 1. Dependency inversion principle.

Dependency inversion principle (Принцип инверсии зависимости) – Модули высокого уровня не должны зависеть от модуля более низкого уровня, все они должны зависеть от абстракций, а абстракции не должны зависеть от деталей, что говорит о том, что детали должны зависеть от абстракций.

Было:

Изображение выглядит как текст, диаграмма, снимок экрана, Параллельный

Автоматически созданное описание

Оптимизация:

Изображение выглядит как текст, снимок экрана, диаграмма, Прямоугольник

Автоматически созданное описание

Финальная оптимизация:

Изображение выглядит как текст, снимок экрана, диаграмма, Параллельный

Автоматически созданное описание

1. Работа с файлами, а также, в принципе, получением информации из вне, и ее обработка.

В Java существует два способа работы с вводом – выводом: IO API (Input - Output) и NIO API (New Input - Output). Существенная разница их использования приведена ниже:

Java IO и NIO — это два разных подхода к работе с вводом-выводом данных.

|  |  |
| --- | --- |
| **Java IO** | **Java NIO** |
| Блокирующая модель ввода-вывода: поток ожидает завершения операции, прежде чем продолжить выполнение кода (блокирует файл и доступ к нему до момента окончания выполнения своей задачи). | Неблокирующая модель ввода-вывода: поток может продолжать работу, не дожидаясь завершения операции |
| Потоко-ориентированный: данные считываются и записываются в поток байтов | Буферо-ориентированный: данные считываются и записываются в буфер |
| Имеет только одно соединение на один поток | Может обрабатывать несколько соединений на поток |
| Базируется на потоке: передаёт данные в одну сторону (one-directional) | Базируется на канале:  передаёт данные в обе стороны (bi-directional) |
| Использует Heap — основной вариант работы с памятью в JVM | Использует ресурсы вне JVM, что позволяет снизить накладные расходы памяти |
| Подходит для простой синхронной обработки данных, например для парсинга XML-документов небольшого объёма в одном потоке | Подходит для обработки больших объёмов данных с использованием множества сетевых соединений одновременно, например для многопоточной асинхронной обработки файлов |
| **Итог:**  + простота использования, + надёжность, - производительность. | **Итог:**  + производительность, + гибкость, - сложности в настройке и оптимизации. |

У каждого API свои преимущества и недостатки. Важно иметь о них представление, чтобы понимать, какой API в каких ситуациях использовать.

* 1. IO API.

IO API очень удобный инструмент с работой ввода-вывода, поэтому разберем, как с ним работать, и какие именно возможности, он предоставляет пользователю.

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описаниеИерархия:

Как мы уже знаем, IO API работает с потоками данных, поэтому прямые наследники от класса Object идут InputStream и OutputStream, все остальные вариации потоков – их дочерние классы.

Пример:

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Из-за того, что все эти классы имеют ограниченный функционал (работать можно только с байтами, следовательно только работа с символами не строчками и т.д.), были придуманы два других класса для более удобного и расширенного использования данных потоков: Reader и Writer.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описаниеБолее подробно:

Reader и Writer расширяют возможности, позволяя работать например со строчками, преобразуя в поток в буфер, что обеспечивает эффективное чтение символов, массивов и строк. При всем при этом, поток преобразуется в контейнер типа Reader, таким образом объекты классов семейства Reader принимают в себя классы типа Input/OutputStream (Не все, зависит от реализации, BufferReader принимает Reader классы). Следующая страница с парой примеров.

Пример:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Пример2:



* 1. NIO API.

Все про NIO API ниже по ссылке:

<https://1drv.ms/f/s!AiunBd2rVeExrkQqu4xI4BZWx5T4?e=chL3ME>

[Файл NIO API.](NIO_API.docx)

Файл NIO API.