**Классы** — это, можно сказать, основа основ программирования на Java.

Java — это объектно-ориентированный язык программирования. Все программы состоят из объектов, которые как-то связываются между собой. Класс — это, по сути, шаблон для объекта. Он определяет, как объект будет выглядеть и какими функциями обладать. **Каждый объект является объектом какого-то класса**. Рассмотрим самый простой пример:

**public** **class** Cat {

String name;

**int** age;

}

Допустим, мы пишем программу, и в этой программе нам для чего-то нужны кошки (например, у нас ветеринарная клиника с возможностью онлайн-записи). Мы создали класс Cat, и указали для него две переменные — строку name, и число age. Такие переменные классов называются **полями**. По сути — это шаблон для всех кошек, которых мы создадим в будущем. У каждой кошки (объекта класса Cat) будет две переменные — имя и возраст.

**public** **class** Cat {

String name;

**int** age;

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat();

barsik.age = 3;

barsik.name = "Барсик";

System.out.println("Мы создали кота по имени " + barsik.name + ", его возраст - " + barsik.age);

}

}

Вот так это работает! Мы создали кота, присвоили ему имя и возраст и вывели все это в консоль. Ничего сложного :) Классы чаще всего описывают предметы и явления окружающего мира. Кошка, стол, человек, молния, страница книги, колесо — все это в твоей программе будет создаваться с помощью отдельных классов. Теперь давай обратим внимание на переменные, которые мы создали в классе Cat. Они называются полями, или **переменными экземпляров**. В названии, собственно, раскрыта вся их суть. Эти переменные будут **у каждого экземпляра** (объекта) класса Cat. У каждого кота, которого мы создадим, будет своя переменная name и своя переменная age. Логично, в общем-то: с настоящими котами все так и есть :) Кроме переменных экземпляров существуют и другие — **переменные классов**, или статические. Дополним наш пример:

**public** **class** Cat {

String name;

**int** age;

**static** **int** count = 0;

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat();

barsik.age = 3;

barsik.name = "Барсик";

count++;

Cat vasia = **new** Cat();

vasia.age = 5;

vasia.name = "Вася";

count++;

System.out.println("Мы создали кота по имени " + barsik.name + ", его возраст - " + barsik.age);

System.out.println("Мы создали кота по имени " + vasia.name + ", его возраст - " + vasia.age);

System.out.println("Общее количество котов = " + count);

}

}

Вывод в консоль:

Мы создали кота по имени Барсик, его возраст - 3

Мы создали кота по имени Вася, его возраст - 5

Общее количество котов = 2

Теперь у нас в классе появилась новая переменная — count (количество). Она отвечает за подсчет созданных котов. Каждый раз, когда в методе main мы создаем кота, мы увеличиваем эту переменную на 1. Эта переменная обозначена ключевым словом **static**. Это значит, что **она принадлежит классу**, а не конкретному объекту класса. Что, конечно, логично: если имя у каждого кота должно быть свое, то счетчик котов нам нужен один на всех. Именно этого позволяет добиться слово **static** — переменная count одна для всех котов. Обрати внимание: когда мы выводим ее в консоль, мы не пишем barsik.count или vasia.count. Она не принадлежит ни Барсику, ни Васе — она принадлежит всему классу Cat. Поэтому — просто count. Можно также написать Cat.count — это тоже будет правильно. С выводом в консоль переменной name у нас бы такое не прошло:

**public** **class** Cat {

String name;

**int** age;

**static** **int** count = 0;

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat();

barsik.age = 3;

barsik.name = "Барсик";

count++;

System.out.println("Мы создали кота по имени " + name + ", его возраст - " + barsik.age);

System.out.println("Общее количество котов = " + count);

}

}

Ошибка! name у каждого кота свое. В этом месте компилятор сбивается с толку. "Вывести в консоль имя? А чье имя-то? :/"

**Методы**

Кроме переменных, у каждого класса есть методы. О них мы поговорим в отдельной лекции более подробно, но общие моменты довольно просты. **Методы** — это функционал твоего класса; то, что объекты этого класса умеют делать. С одним из методов ты уже знаком — это метод main(). Но метод main, как ты помнишь, является статическим — то есть он принадлежит всему классу (логика такая же, как с переменными). А обычные, нестатические методы, можно вызывать только на конкретных объектах, которые мы создали. Например, если мы хотим написать класс для кошки, нам надо понять, какими функциями кошка должна обладать в нашей программе. Исходя из этого, напишем для нее пару методов:

**public** **class** Cat {

String name;

**int** age;

**public** **void** sayMeow() {

System.out.println("Мяу!");

}

**public** **void** jump() {

System.out.println("Прыг-скок!");

}

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat();

barsik.age = 3;

barsik.name = "Барсик";

barsik.sayMeow();

barsik.jump();

}

}

Ну вот, теперь наш класс гораздо больше похож на описание настоящей кошки! У нас теперь не просто кот Барсик с именем и возрастом. Он еще умеет мяукать и прыгать! Какой же кот без такого "функционала" :) Берем конкретный объект — barsik, и вызываем у него методы sayMeow() и jump(). Смотрим в консоль:

Мяу!

Прыг-скок!

Настойщий кот! :)

**Создание собственных классов. Абстракция**

В будущем тебе придется писать собственные классы. На что нужно обратить внимание при их написании? Если мы говорим о переменных, тебе нужно пользоваться такой вещью как **абстракция**. **Абстракция** — один из четырех основных принципов объектно-ориентированного программирования. Он подразумевает **выделение главных, наиболее значимых характеристик предмета**, и наоборот — отбрасывание второстепенных, незначительных. Например, мы создаем картотеку работников компании. Для создания объектов "работник" мы написали класс **Employee**. Какие характеристики важны для описания работника в картотеке компании? ФИО, дата рождения, номер социального страхования, ИНН. Но вряд ли в карточке работника компании нам нужны его рост, цвет глаз и волос. Компании эта информация ни к чему. Поэтому для класса Employee мы зададим переменные String name, int age, int socialInsuranceNumber и int taxNumber, а от лишней для нас информации (типа цвета глаз) откажемся, **абстрагируемся**. А вот если мы создаем картотеку фотомоделей для модельного агентства, ситуация резко меняется. Для описания фотомодели нам как раз очень важны рост, цвет глаз и цвет волос, а вот номер ИНН для нее нам абсолютно не важен. Поэтому в классе Model нам нужно создать переменные int height, String hair, String eyes. Вот так и работает абстракция, все просто! :)

**Конструкторы**

Давай вернемся к нашему примеру с кошками.

**public** **class** Cat {

String name;

**int** age;

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat();

System.out.println("Тут в программе в течение 2-х часов что-то происходит...");

barsik.age = 3;

barsik.name = "Барсик";

}

}

Посмотри на этот код и попробуй догадаться, что с нашей программой не так. На протяжении двух часов в нашей программе существовал кот без имени и возраста! Конечно, это в корне неверно. В базе данных ветеринарной клиники не должно быть котов без информации о них. Сейчас мы отдаем это на откуп программиста. Не забудет он указать имя и возраст — все будет ок. Забудет — в базе будет ошибка, неизвестные коты. Как нам решить эту проблему? Надо каким-то образом запретить создавать котов без имени и возраста. Здесь нам на помощь приходят **функции-конструкторы**. Приведем пример:

**public** **class** Cat {

String name;

**int** age;

//конструктор для класса Cat

**public** Cat(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat("Barsik", 5);

}

}

**Конструктор** — это, по сути, шаблон для объектов класса. В данном случае мы указывем, что для каждого объекта cat должны быть указаны два аргумента — строка и число. Если мы теперь попытаемся создать безымянного кота — у нас это не получится.

**public** **class** Cat {

String name;

**int** age;

**public** Cat(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat(); //ошибка!

}

}

Теперь, когда в классе есть конструктор, компилятор Java знает, как должны выглядеть объекты, и не позволяет создавать объекты без указанных в нем аргументов. Теперь давай разберемся с ключевым словом **this**, которое ты видишь внутри конструктора. С ним тоже все просто. "this" по-английски — "этот, этого". То есть это слово указывает на конкретный предмет. Код в конструкторе

**public** Cat(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

можно перевести почти дословно: "**name** для этого кота (которого мы создаем) = аргументу **name**, который указан в конструкторе. **age** для этого кота (которого мы создаем) = аргументу **age**, который указан в конструкторе." После срабатывания конструктора можешь проверить, что нашему коту присвоились все нужные значения:

**public** **class** Cat {

String name;

**int** age;

**public** Cat(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat("Барсик", 5);

System.out.println(barsik.name);

System.out.println(barsik.age);

}

}

Вывод в консоль:

Барсик

5

Когда конструктор отработал:

Cat barsik = **new** Cat("Барсик", 5);

Внутри него по факту произошло следующее:

**this**.name = "Барсик";

**this**.age = 5;

И объекту barsik (он и является **this**) присвоились значения из аргументов конструктора. На самом деле, если не указать в классе конструкторы **у него все равно будет срабатывать конструктор**! Но как такое возможно? О\_О Дело в том, что в Java у всех классов есть так называемый **конструктор по умолчанию**. У него нет никаких аргументов, но он срабатывает каждый раз при создании любого объекта любого класса.

**public** **class** Cat {

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat(); //вот здесь сработал конструктор по умолчанию

}

}

На первый взгляд это не заметно. Ну создали объект и создали, где тут работа конструктора? Чтобы это увидеть, давай прямо руками напишем для класса Cat пустой конструктор, а внутри него выведем какую-нибудь фразу в консоль. Если она выведется, значит конструктор отработал.

**public** **class** Cat {

**public** Cat() {

System.out.println("Создали кота!");

}

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat(); //вот здесь сработал конструктор по умолчанию

}

}

Вывод в консоль:

Создали кота!

Вот и подтверждение. Конструктор по умолчанию всегда незримо присутствует в твоих классах. Но тебе нужно знать еще одну его особенность. **Дефолтный конструктор исчезает из класса, когда ты создаешь какой-то конструктор с аргументами.** Доказательство этого, на самом деле, мы уже видели выше. Вот в этом коде:

**public** **class** Cat {

String name;

**int** age;

**public** Cat(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat(); //ошибка!

}

}

Мы не смогли создать кота без имени и возраста, потому что определили конструктор для Cat: строка + число. Дефолтный конструктор сразу после этого исчез из класса. Поэтому обязательно запомни: если тебе в твоем классе нужно несколько конструкторов, включая пустой, **его нужно создать отдельно**. Например, наша ветеринарная клиника хочет делать добрые дела и помогать бездомным котикам, чьих имен и возраста мы не знаем. Тогда наш код должен выглядеть так:

**public** **class** Cat {

String name;

**int** age;

//для домашних котов

**public** Cat(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

//для уличных котов

**public** Cat() {

}

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat("Barsik", 5);

Cat streetCat = **new** Cat();

}

}

Теперь, когда мы явно указали конструктор по умолчанию, мы можем создавать котов обоих типов. В конструкторе можно присваивать значения и явно, а не только брать их из аргументов. Например, мы можем записывать всех уличных котов в базу данных под именем "Уличный кот номер..." :

**public** **class** Cat {

String name;

**int** age;

**static** **int** count = 0;

**public** Cat() {

count++;

**this**.name = "Уличный кот номер " + count;

}

**public** Cat(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

Cat streetCat1 = **new** Cat();

Cat streetCat2 = **new** Cat();

System.out.println(streetCat1.name);

System.out.println(streetCat2.name);

}

}

У нас есть переменная count, которая является счетчиком уличных котов. Каждый раз при выполнении конструктора по умолчанию мы увеличиваем ее на 1 и присваем этот номер в качестве имени кота. **Для конструктора очень важен порядок следования аргументов.** Поменяем в нашем конструкторе аргументы имени и возраста местами.

**public** **class** Cat {

String name;

**int** age;

**public** Cat(**int** age, String name) {

**this**.name = name;

**this**.age = age;

}

**public** **static** **void** main(String[] args) {

Cat barsik = **new** Cat("Барсик", 10); //ошибка!

}

}

Ошибка! Конструктор четко описывает: при создании объекта Cat ему должны быть переданы **число и строка, именно в таком порядке**. Поэтому наш код не срабатывает. Обязательно запомни это и учитывай при создании своих собственных классов:

**public** Cat(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

**public** Cat(**int** age, String name) {

**this**.age = age;

**this**.name = name;

}

**Это два абсолютно разных конструктора!** Теперь реши несколько задач на закрепление материала :)

* Музей древностей.

Твоя задача — спроектировать класс Artifact. Артефакты, которые хранятся в музее, бывают трех видов. Первый — о которых неизвестно ничего, кроме порядкового номера, присвоенному музеем (например: 212121). Второй — о которых известен порядковый номер и культура, которой он был создан (например: 212121, "Ацтеки"). Третий вид — о которых известен порядковый номер, культура, которой он был создан, и точный век его создания (например: 212121, "Ацтеки", 12). Создай класс Artifact, описывающий хранящиеся в музее древности, и напиши необходимое количество конструкторов для него. В методе main() создай по одному артефакту каждого вида.

**public** **class** Artifact {

**public** **static** **void** main(String[] args) {

}

}

* Сайт знакомств

Вы создаете базу данных пользователей для сайта знакомств. Да вот беда — вы подзабыли в каком порядке их нужно указывать, а технического задания под рукой нет. Спроектируйте класс User, у которого будут поля — имя (String) возраст (short) и рост (int). Создайте для него необходимое количество конструкторов, чтобы имя, возраст и рост можно было указывать в любом порядке.

**public** **class** User {

String name;

**short** age;

**int** height;

**public** **static** **void** main(String[] args) {

}

}

**Типы данных**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тип** | **Размер, байт** | **Диапазон значений** | **Значение по умолчанию** | **Описание** |
| byte | 1 | -128 .. 127 | 0 | Самое маленькое целое — один байт |
| short | 2 | -32,768 .. 32,767 | 0 | Короткое целое, два байта |
| int | 4 | — 2\*109 .. 2\*109 | 0 | Целое число, 4 байта |
| long | 8 | — 9\*1018 .. 9\*1018 | 0L | Длинное целое, 8 байт |
| float | 4 | -10127 .. 10127 | 0.0f | Дробное, 4 байта |
| double | 8 | -101023 .. 101023 | 0.0d | Дробное, двойной длины, 8 байт |
| boolean | 1 | true, false | false | Логический тип (только true & false) |
| char | 2 | 0..65,535 | ‘\u0000 ‘ | Символы, 2 байта, все больше 0 |
| Object | 4 | Любая ссылка или null. | null | Хранит ссылки на объекты типа Object или классов наследников |

— Тип byte – это самый маленький тип целых чисел. Каждая переменная этого типа занимает всего один байт памяти, поэтому он может хранить значения в диапазоне от -128 до 127.

— А зачем такой маленький тип нужен. Почему нельзя везде использовать int?

— Можно. Но если ты создаешь очень большие массивы, и тебе не нужно хранить там значения больше 100, то почему бы не воспользоваться таким типом. Так ведь?

— Тип **short** – ровно в два раза длиннее типа byte и тоже хранит только целые числа. Самое большое число, которое в него вмещается – это 32767. Самое большое отрицательное число – это -32768.

— Тип **int** ты уже знаешь. Он может хранить целые числа до двух миллиардов, как положительные, так и отрицательные.

— Тип **float** – создан для хранения вещественных (дробных) чисел. Его размер 4 байта.

— Все дробные числа хранятся в памяти в очень интересной форме.

— Например, число 987654.321 можно представить как 0.987654321\*106. Поэтому в памяти оно будет представлено как два числа «0.987654321» (**мантисса — значащая часть числа**) и «6» (**экспонента — степень десятки**)

— А зачем так сделано?

— Такой подход позволяет хранить гораздо большие числа, чем int, используя всего 4 байта. Но при этом мы жертвуем точностью. Часть памяти расходуется на хранение мантиссы, поэтому такие числа хранят всего 6-7 знаков после запятой, остальные отбрасываются.

—Такие числа еще называют «числа с плавающей запятой» или «числа с плавающей точкой (**float**ing point number)». Отсюда, кстати, и название типа – **float**.

— Тип **double** – это такой же тип, как и **float**, только в два раза длиннее – он занимает восемь байт. (**double** – двойной, по-английски). И предельный размер мантиссы и количество значащих цифр в нем больше. Если тебе нужно хранить вещественные числа – старайся использовать именно этот тип.

— Тип **char** – гибридный тип. Его значения можно интерпретировать и как числа (их можно складывать и умножать) и как символы. Так было сделано потому, что хоть символы и имеют визуальное представление, для компьютера они в первую очередь просто числа. И работать с ними как с числами гораздо удобнее. Тут еще есть одно замечание: тип **char** строго положительный – отрицательных значений он хранить не может.

— Тип **boolean** – логический тип и может хранить всего два значения: **true** (истина) и **false**(ложь).

— Тип **Object**, хоть и представлен в таблице, примитивным типом не является. Это базовый класс для всех классов в Java. Во-первых, все классы считаются унаследованными от него, а значит, содержат его методы. А во-вторых, ему можно присваивать ссылки на объекты любого типа. В том числе и **null** – пустую ссылку.

**Модификаторы методов**

### 1. Модификаторы доступа

Перед каждым методом программист может указать так называемые модификаторы доступа. К ним относятся ключевые слова: public, protected, private.

Эти «модификаторы доступа» позволяют ограничивать доступ к методу из других классов.

Например, если перед объявлением метода написать ключевое слово private, этот метод можно будет вызывать только из того же класса, в котором он объявлен. Ключевое слово public разрешает обращение к помеченному методу из любого метода любого класса.

Всего таких модификаторов 3, а типов доступа к методу — 4, ведь отсутствие модификатора доступа тоже что-то значит.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Доступ из...** | | | |
| Модификаторы | Любого класса | Класса-наследника | Своего пакета | Своего класса |
| public | **Есть** | **Есть** | **Есть** | **Есть** |
| protected | **Нет** | **Есть** | **Есть** | **Есть** |
| без модификатора | **Нет** | **Нет** | **Есть** | **Есть** |
| private | **Нет** | **Нет** | **Нет** | **Есть** |

**1. Модификатор public.**

К методу (переменной или классу), помеченному модификатором public, можно обращаться из любого места программы. Это самая высокая степень открытости – никаких ограничений нет.

**2. Модификатор private.**

К методу (переменной или классу), помеченному модификатором private, можно обращаться только из того же класса, где он объявлен. Для всех остальных классов помеченный метод (или переменная) – невидимы и «как бы не существуют». Это самая высокая степень закрытости — только свой класс.

**3. Без модификатора (модификатор по умолчанию).**

Если метод (или переменная) не помечены никаким модификатором, считается, что они помечены «модификатором по умолчанию». Переменные или методы с таким модификатором (т.е. вообще без какого-нибудь) видны всем классам пакета, в котором они объявлены. И только им. Этот модификатор еще иногда называют package, намекая, что доступ к переменным и методам открыт для всего пакета, в котором находится их класс

**4. Модификатор protected.**

Если метод помечен модификатором protected, к нему можно обращаться из того же класса, того же пакета и классов наследников (классов, унаследованных от класса, в котором объявлен метод). Эту тему разберем подробнее в квесте Java Core.

До конца квеста Java Syntax вы можете всем своим методам (а также классам и переменным класса) указывать модификатор public. Другие модификаторы вам понадобятся, когда мы начнем активно изучать ООП.

**Зачем нужны модификаторы доступа?**

Все дело в больших проектах, которые пишут десятки и сотни программистов одновременно.

Иногда бывают ситуации, когда программист хочет разделить слишком большой метод своего класса на части и вынести часть кода во вспомогательные методы. Но вместе с тем он не хочет, чтобы другие программисты вызывали эти вспомогательные методы, т.к. тогда код его класса может работать неправильно.

Вот и придумали такие модификаторы доступа. Пометил вспомогательный метод словом **private**, и никакой код кроме вашего класса не сможет увидеть ваш вспомогательный метод.

### 2. Ключевое слово static

Ключевое слово static превращает метод в статический. Что это такое, мы рассмотрим позднее, а сейчас просто запомним пару фактов про статические методы.

**Факт 1. Статический метод не привязан ни к какому объекту**, но относится к классу, в котором объявлен. Чтобы вызвать статический метод, нужно написать:

ИмяКласса.имяМетода()

Примеры статических методов:

|  |  |  |
| --- | --- | --- |
|  | **Имя класса** | **Имя статического метода** |
| Thread.sleep() | Thread | sleep() |
| Math.abs() | Math | abs() |
| Arrays.sort() | Arrays | sort() |

Имя класса перед именем статического метода можно не писать, если вы вызываете статический метод изнутри — из того же класса. Именно поэтому вам не нужно каждый раз писать Solution перед именами вызываемых статических методов.

**Факт 2. Статический метод не может обращаться к нестатическим методам** его же класса. Статический метод может обращаться только к статическим методам. Поэтому все методы, которые мы хотим вызвать из метода main, мы объявляем статическими.

### 3. Ключевое слово throws

Есть еще одно ключевое слово, которое вы, вероятно, видели рядом с объявлением метода — это слово throws. Оно, в отличие от модификаторов доступа и ключевого слова static, размещается после параметров метода:

**public** **static** тип имя(параметры) **throws** Exception

{

код метода

}

Какой именно смысл в нем заключается, мы разберем немного позднее, когда будем изучать исключения.

Но если поверхностно коснуться его, можно сказать, что в методе, помеченном словом throws, могут возникать ошибки (исключения) типа Exception (и унаследованных от него классов). Если в классе могут возникнуть ошибки нескольких типов, нужно перечислить все типы через запятую.

### 4. Метод main

Строка, в которой объявляется метод, со всеми модификаторами, будет влиять на то, как этот метод будет вызываться из других классов и методов. Результат какого типа он будет возвращать и какие возможны ошибки при его работе.

Такая строка называется объявлением метода и имеет общий вид:

модификаторы доступа **static** тип имя(параметры) **throws** исключения

*Общий вид объявления метода*

Где модификаторы доступа — это public, protected, private или ничего;

static ставится, если метод статический (отсутствует для нестатических методов)

тип — тип результата функции (void если результат отсутствует)

Теперь вы скорее всего понимаете, что значат все ключевые слова при объявлении метода main:

**public** **static** **void** main(String[] args) **throws** Exception

*Объявление метода main*

Доступ к методу main() возможен из любого класса, об этом говорит слово public.

Метод статический (его можно явно вызвать в виде Solution.main()).

Метод main не возвращает никакого результата. Тип результата — void (пустой тип/заглушка)

Метод main принимает параметры(!): массив строк. И название args намекает на «аргументы программы». При запуске программы в нее можно передать параметры — массив строк. Они и будут содержаться в массиве args метода main().

В методе main() могут возникать необработанные ошибки типа Exception (или его наследники).

**Переменные**

## 1. Переменные и коробки

*Переменные* — это такие специальные штуки для хранения данных. Любых данных. Все данные в Java хранятся с помощью переменных. Больше всего переменная по смыслу напоминает коробку: самую обычную *коробку*.

Допустим, вы написали на листке число 13 и положили этот лист в коробку. Теперь мы можем говорить, что «коробка хранит значение 13».

Каждая переменная в Java имеет три важных свойства: *тип*, *имя* и *значение*.

*Имя* используется для того, чтобы отличать одну переменную от другой. Это как *надпись на коробке*.

*Тип* переменной определяет тип *значений/данных*, которые в ней можно хранить. В коробке для торта храним торт, в коробке для обуви — туфли, и т.д.

*Значение* — это некий объект, данные или информация, которая хранится в переменной.

Каждый объект в языке Java имеет свой *тип*. Например, могут быть такие типы данных: *целое число*, *дробное число*, *текст*, *тип Кот*, *тип Дом* и т.д.

У каждой *переменной* (коробки) тоже есть свой *тип*. Переменная может хранить значения только того же типа, что и она сама. Разные коробки используются для хранения различных вещей: коробка конфет, коробка для десятка яиц и т.п. Все как в жизни.

## 2. Создание переменной

В языке Java для того, чтобы создать переменную, используется команда вида:

тип имя;

*Объявление переменной*

Где *тип* — это тип переменной (совпадает с типом значений, которые переменная сможет хранить). А *имя* — это имя переменной.

Примеры:

|  |  |
| --- | --- |
| **Создание переменной: сначала тип, затем имя.** | **Описание** |
| int a; | Создаем переменную по имени a типа int. |
| String s; | Создаем переменную по имени s типа String. |
| double c; | Создаем переменную по имени c типа double. |

Два самых часто используемых типа — это *целые числа* (обозначается словом int) и *текст* (обозначается словом String). Так же популярен тип double — это дробные (вещественные) числа.

## 3. Присваивание

Как уже говорилось выше, у переменной есть имя, тип и значение. Имя и тип мы уже разобрали, а что насчет значения? Как занести значение в переменную?

Чтобы занести значение в переменную, существует специальная операция — *операция присваивания*. Она копирует значение из одной переменной в другую. Не переносит, а именно копирует. Как файл на диске. Выглядит это присваивание так:

имя = значение;

*Операция присваивания*

Где имя — это имя переменной, а значение — это значение, которое будет занесено в переменную. В качестве значения может фигурировать конкретное значение, имя другой переменной или даже какое-нибудь выражение с использованием переменных.

Примеры:

|  |  |
| --- | --- |
| **Код** | **Описание** |
| **int** i;  **int** a, b;  **int** x; | Создается переменная i Создаются переменные a, b Создается переменная x |
| i = 3; | В переменную i заносится значение 3. |
| a = 1;  b = a + 1; | В переменную a заносится значение 1. В переменную b заносится значение 2. |
| x = 3;  x = x + 1; | В переменную x заносится значение 3. В следующей строке значение x увеличивается на 1, x теперь равен 4. |

Для операции присваивания используется символ равно =. Это не сравнение. Это именно команда копирования значения справа от знака *равно* в *переменную*, которая слева. Для *сравнения* в языке Java используется двойное равно ==.

**Типы переменных**

## 1. Объявление переменных

Давайте еще раз разберем, как создавать переменные.

Чтобы создать переменную, нужно написать такую команду: тип имя;.

Примеры:

|  |  |
| --- | --- |
| **Команда** | **Пояснение** |
| String s; | Создается переменная s типа String. Такая переменная может хранить текст. |
| **int** x; | Создается переменная x типа int. Такая переменная может хранить в себе целые числа. |
| **int** a, b, c;  **int** d; | Создаются переменные a, b, c, d типа int. Такие переменные могут хранить целые числа. |

**Важно!**

Нельзя создать две переменные с одинаковыми именами в одном методе. А вот в разных методах — можно. Это как коробки, стоящие в разных домах.

Еще есть ограничения на *имя переменной*. С одной стороны, оно может быть любым, с другой – оно не может содержать *пробелов*, символов +, - и т.д. Лучше всего в имени переменной использовать только *латинские буквы* и *цифры*.

Обратите внимание, что в **языке Java** играет роль, какие буквы вы пишите — **большие** или **маленькие**. int a — это не то же самое, что и Int a.

Кстати, в Java можно одновременно создавать *переменную* и присваивать ей *значение*. Это экономит время и место:

|  |  |
| --- | --- |
| **Компактный код** | **Длинный код, эквивалентный коду слева** |
| int a = 5; | int a;  a = 5; |
| int b = 6; | int b;  b = 6; |
| int c = 7; | int c;  c = 7; |
| int d = c + 1; | int d;  d = c + 1; |
| String s = "I'm Amigo"; | String s;  s = "I'm Amigo"; |

Так гораздо компактнее и понятнее.

Ну а раз с созданием переменных мы уже разобрались, давайте немного познакомимся с двумя самыми используемыми типами в языке Java. Это типы int (целые числа) и String (текст/строки).

## 2. Тип int

В переменной типа int можно хранить целые числа. Также над типом int можно выполнять различные операции: сложение, вычитание, умножение, деление и другие. Примеры:

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| **int** x = 1;  **int** y = x\*2;  **int** z = 5\*y\*y + 2\*y + 3; | x будет равен 1 y будет равен 2 z будет равен 20+4+3, будет равен 27 |
| **int** a = 5;  **int** b = 1;  **int** c = (a-b) \* (a+b); | a будет равно 5 b будет равно 1 c будет равно 4\*6, будет равно 24 |
| **int** a = 64;  **int** b = a/8;  **int** c = b/4;  **int** d = c\*3; | a будет равно 64 b будет равно 8 c будет равно 2 d будет равно 6 |

## 3. Тип String

Тип String позволяет хранить текстовые строки.

Чтобы задать какую-то текстовую строку, в Java надо написать *текст строки*, а с обеих сторон поставить *двойные кавычки*. Пример:

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| String s = "Amigo"; | s будет содержать текст Amigo |
| String s = "123"; | s будет содержать текст 123 |
| String s = "Bond 007"; | s будет содержать текст Bond 007 |

Выглядит несложно, правда? Ну если так, тогда вот еще один интересный факт.

Строки в Java можно склеивать с помощью знака плюс — +. Пример:

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| String s = "Amigo" + " the best"; | s будет содержать Amigo the best |
| String s = ""; | s будет содержать пустую строку — вообще без символов. |
| **int** x = 333;  String s = "Amigo" + x; | s будет содержать Amigo333 |

Обратите внимание на последний пример: мы сложили *строку* и *число*. Тут тоже все просто: число будет преобразовано в строку, затем две строки будут склеены. При *сложении* *строк* и *чисел*, всегда получается *строка*.

## 4. Вывод переменной на экран

Вроде все так очевидно и элементарно. Ну тогда, может, вы сразу догадаетесь, какой *командой* можно вывести переменную на экран?

На самом деле все просто. Чтобы вывести что-то на экран, мы используем команду System.out.println(), куда в качестве параметра передаем то, что мы выводим.

|  |  |
| --- | --- |
| **Код** | **Вывод на экран** |
| System.out.println("Amigo"); | Amigo |
| System.out.println("Ami" + "go"); | Amigo |
| String s = "Amigo";  System.out.println(s); | Amigo |
| String s = "Am";  System.out.println(s + "igo"); | Amigo |

**Базовые операторы.**

## Какие бывают операторы в Java

Для любой операции нам нужно как минимум две вещи:

* оператор;
* операнд.

Примером оператора может быть простой плюс в операции сложения двух чисел. А складываемые друг с другом числа будут в этом случае операндами. Итак, с помощью операторов мы выполняем операции над одним или несколькими операндами. Операторы, которые осуществляют операции над двумя операндами, называются бинарными. Например, сложение двух чисел. Операторы, которые осуществляют операции над одним операндом, называются унарными. Например, унарный минус.

## Операции над числами в Java

Самая частая операция, которую программисты производят над числами — присвоение числового значения какой либо переменной. Она, как и оператор = тебе знакомы:

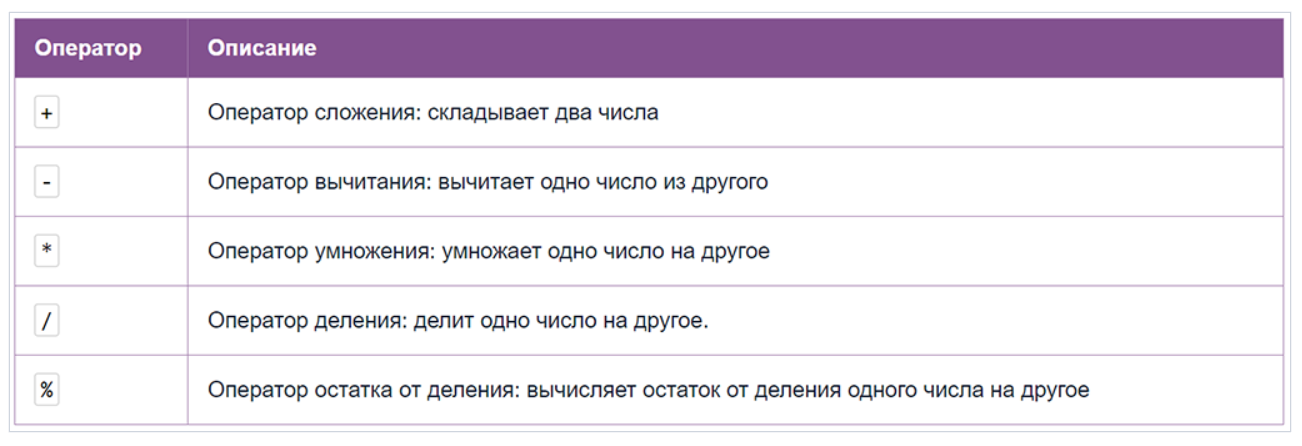
**int** a = 1;

**int** b = 2;

**int** c = 3;

Есть также арифметические операции. Они осуществляются с помощью бинарных арифметических операторов:

Таблица 1. Бинарные арифметические операторы



Первые четыре оператора не должны вызывать вопросов: все так же, как в математике. Последний оператор, остаток от деления, также не делает ничего сверхсложного. К примеру, если разделить 24 на 7, мы получим 3 целых и 3 в остатке. Именно остаток и вернет данный оператор:

System.out.println(24 % 7); // выведет 3

Java позволяет комбинировать: например, операторы присваивания и арифметические операторы. Рассмотрим пример:

**int** x = 0;

x = x + 1; // x = 0 + 1 => x = 1

x = x + 1; // x = 1 + 1 => x = 2

x = x + 1; // x = 2 + 1 => x = 3

Здесь мы задали переменную x и присвоили ей нулевое значение. Далее в каждой строке мы присваиваем значению x сумму текущего значения переменной x и единицы. В комментариях к каждой строке есть пояснения. Эту процедуру называют наращиванием или инкрементированием переменной. Операцию инкрементирования из примера выше можно заменить на аналогичную с использованием комбинации операторов:

**int** x = 0;

x += 1; // x = 0 + 1 => x = 1

x += 1; // x = 1 + 1 => x = 2

x += 1; // x = 2 + 1 => x = 3

Комбинировать оператор присваивания можно с любым арифметическим оператором:

**int** x = 0;

x += 10; // x = 0 + 10 => x = 10

x -= 5; // x = 10 - 5 => x = 5

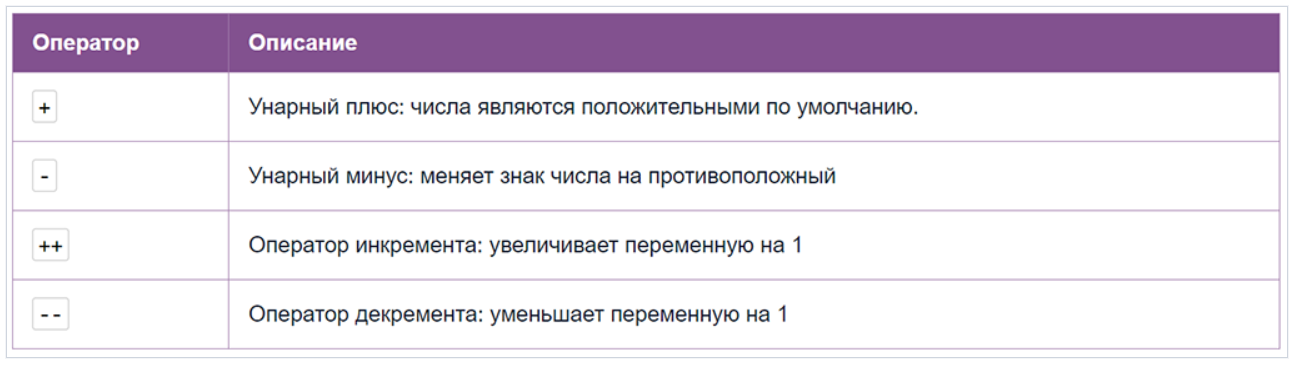
x \*= 5; // x = 5 \* 5 => x = 25

x /= 5; // x = 25 / 5 => x = 5

x %= 3; // x = 5 % 3 => x = 2;

Помимо бинарных, в Java есть унарные арифметические операторы.

*Таблица 2. Унарные арифметические операторы:*



Пример унарных плюса и минуса:

**int** x = 0;

x = (+5) + (+15); //Скобки для наглядности, можно и без них

System.out.println("x = " + x);

**int** y = -x;

System.out.println("y = " + y);

Операции инкремента и декремента по сути просты. В первом случае происходит увеличение переменной на 1, во втором — уменьшение переменной на 1. Пример ниже:

**int** x = 9;

x++;

System.out.println(x); // 10

**int** y = 21;

y--;

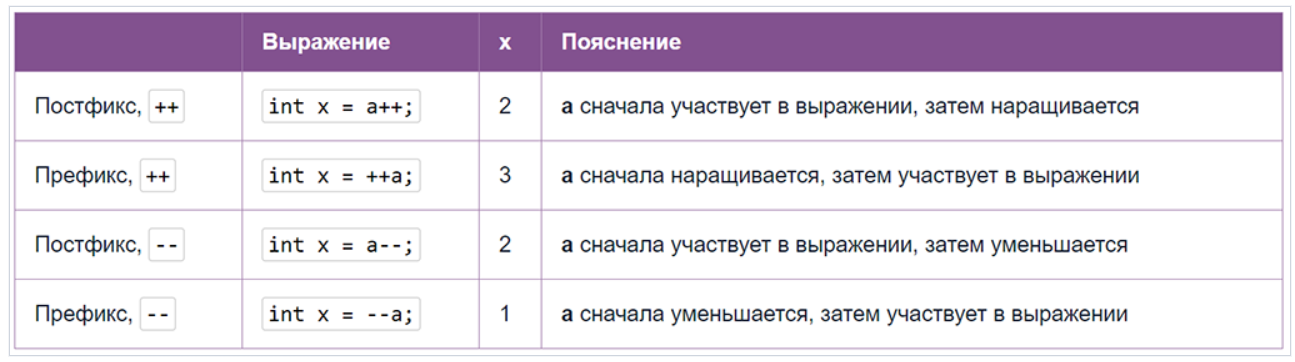
System.out.println(y); // 20

Есть два типа данных операций — постфиксная и префиксная. В первом случае оператор пишется после переменной, во втором случае — перед переменной. Разница лишь в том, когда выполнится операция инкрементирования или декрементирования. Пример и описание в таблице ниже. Предположим, у нас есть переменная:

**int** a = 2;

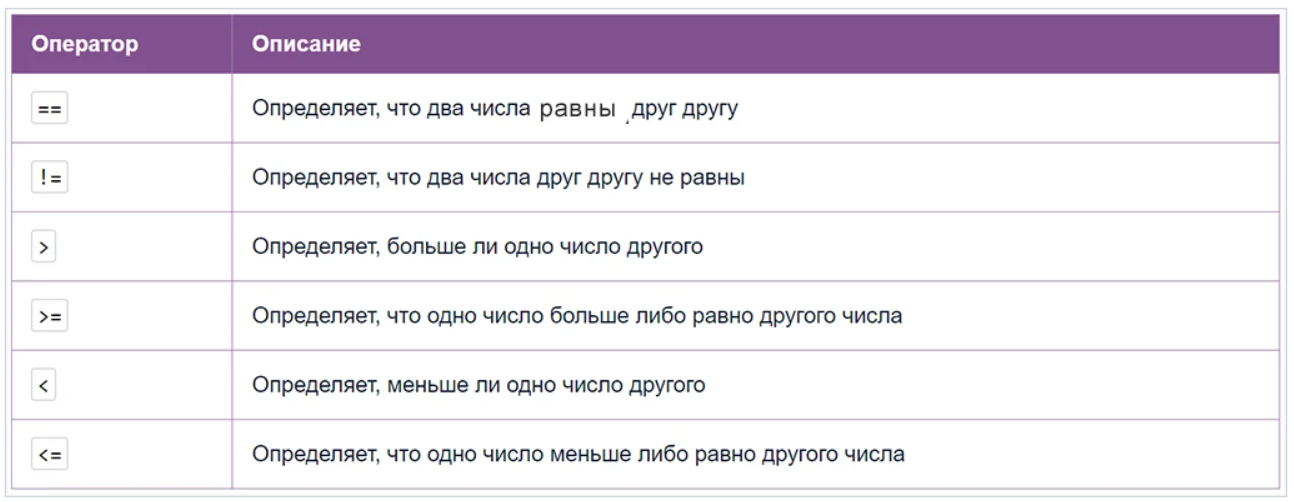
Тогда:

*Таблица 3. Операторы инкремента-декремента:*



Помимо арифметических, существуют операции сравнения (двух чисел). Результатом всегда будет истина либо ложь (*true* / *false*).

*Таблица 4. Операторы сравнения*



Примеры:

**int** a = 1;

**int** b = 2;

**boolean** comparisonResult = a == b;

System.out.println("a == b :" + comparisonResult);

comparisonResult = a != b;

System.out.println("a != b :" + comparisonResult);

comparisonResult = a > b;

System.out.println("a > b :" + comparisonResult);

comparisonResult = a >= b;

System.out.println("a >= b :" + comparisonResult);

comparisonResult = a < b;

System.out.println("a < b :" + comparisonResult);

comparisonResult = a <= b;

System.out.println("a <= b :" + comparisonResult);

## Логические операции в Java

Рассмотрим логические операции и таблицы истинности каждой из них:

* операция отрицания (NOT);
* операция конъюнкции, логическое И (AND);
* операция дизъюнкции, логическое ИЛИ (OR);
* операция сложения по модулю, исключающее ИЛИ (XOR).

Операция отрицания унарная и применяется к одному операнду. Все остальные операции — бинарные. Рассмотрим таблицы истинности данных операций. Здесь 0 — аналог значения false в Java, а 1 — значения true.

Таблица 5. Таблица истинности оператора отрицания (NOT)



Таблица 6. Таблица истинности оператора конъюнкции (AND)

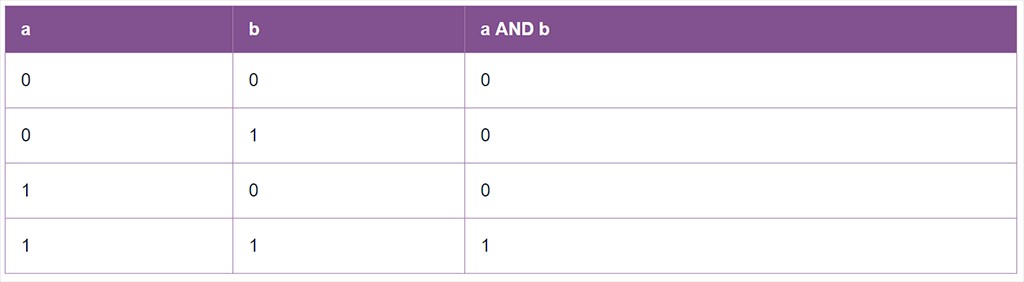


Таблица 7. Таблица истинности оператора дизъюнкции (OR)

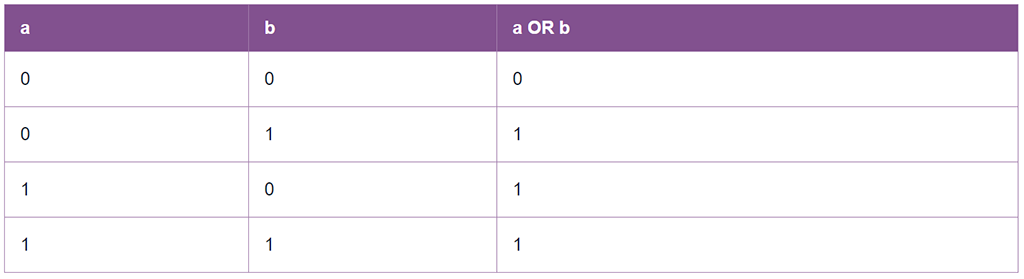


Таблица 8. Таблица истинности оператора сложения по модулю (XOR)

В Java есть те же логические операции:

* ! — оператор отрицания;
* && — оператор логическое И (сокращенный);
* || — оператор логическое ИЛИ (сокращенный);
* & — оператор побитовое И;
* | — оператор побитовое ИЛИ;
* ^ — оператор побитовое исключающее ИЛИ.

Разницу между побитовыми и сокращенными операторами рассмотрим чуть ниже, пока давай преобразуем все таблицы истинности в Java код:

**public** **class** LogicDemo {

**public** **static** **void** main(String[] args) {

notExample();

andExample();

orExample();

xorExample();

}

**public** **static** **void** notExample() {

System.out.println("NOT EXAMPLE:");

System.out.println("NOT false = " + !false);

System.out.println("NOT true = " + !true);

System.out.println();

}

**public** **static** **void** andExample() {

System.out.println("AND EXAMPLE:");

System.out.println("false AND false = " + (false & false));

System.out.println("false AND true = " + (false & true));

System.out.println("true AND false = " + (true & false));

System.out.println("true AND true = " + (true & true));

System.out.println();

}

**public** **static** **void** orExample() {

System.out.println("OR EXAMPLE:");

System.out.println("false OR false = " + (false | false));

System.out.println("false OR true = " + (false | true));

System.out.println("true OR false = " + (true | false));

System.out.println("true OR true = " + (true | true));

System.out.println();

}

**public** **static** **void** xorExample() {

System.out.println("XOR EXAMPLE:");

System.out.println("false XOR false = " + (false ^ false));

System.out.println("false XOR true = " + (false ^ true));

System.out.println("true XOR false = " + (true ^ false));

System.out.println("true XOR true = " + (true ^ true));

System.out.println();

}

}

Данная программа выведет на экран: NOT EXAMPLE: NOT false = true NOT true = false AND EXAMPLE: false AND false = false false AND true = false true AND false = false true AND true = true OR EXAMPLE: false OR false = false false OR true = true true OR false = true true OR true = true XOR EXAMPLE: false XOR false = false false XOR true = true true XOR false = true true XOR true = false Логические операторы применимы только к boolean переменным. В нашем случае мы применяли их сразу к значениям, но можно их использовать и с boolean переменными

Теперь, у нас есть сокращенные операторы (&&, ||) и аналогичные побитовые операторы (&, |). В чем между ними разница? Во-первых, побитовые можно применять к целым числам. Об этом мы поговорим чуть позже. А во-вторых, одни сокращенные, а другие — нет. Чтобы понять, как выглядит сокращенность, посмотрим на выражение:

false AND x = ?

true OR x = ?

Здесь x может принимать любое булево значение. И в целом, согласно законам логики и таблицам истинности, независимо от того, будет x *true* или *false*, результатом первого выражения будет *false*, а результатом второго будет *true*.

Иногда результат выражения можно вычислить уже по первому операнду. Этим и отличаются сокращенные операторы && и ||. В выражениях, аналогичных описанным выше, они не вычисляют значение второго операнда.

В случае с сокращенными операторами не вычисляется вторая часть выражения. Но происходит это только тогда, когда результат выражения очевиден уже по первому операнду.

## Побитовые операции в Java

Ну вот мы и подобрались к самому интересному: побитовым операциям. Как можно понять из названия, это операции, которые производятся над битами. Но прежде чем мы погрузимся в эту тему, стоит поговорить о смежных областях.

### Представление чисел в двоичной системе счисления

Числа, как и любая другая информация в программе, хранятся в памяти компьютера в двоичном коде. Двоичный код — набор нулей и единиц. Каждый ноль или единица представляют собой единицу измерения информации, которая называется бит.

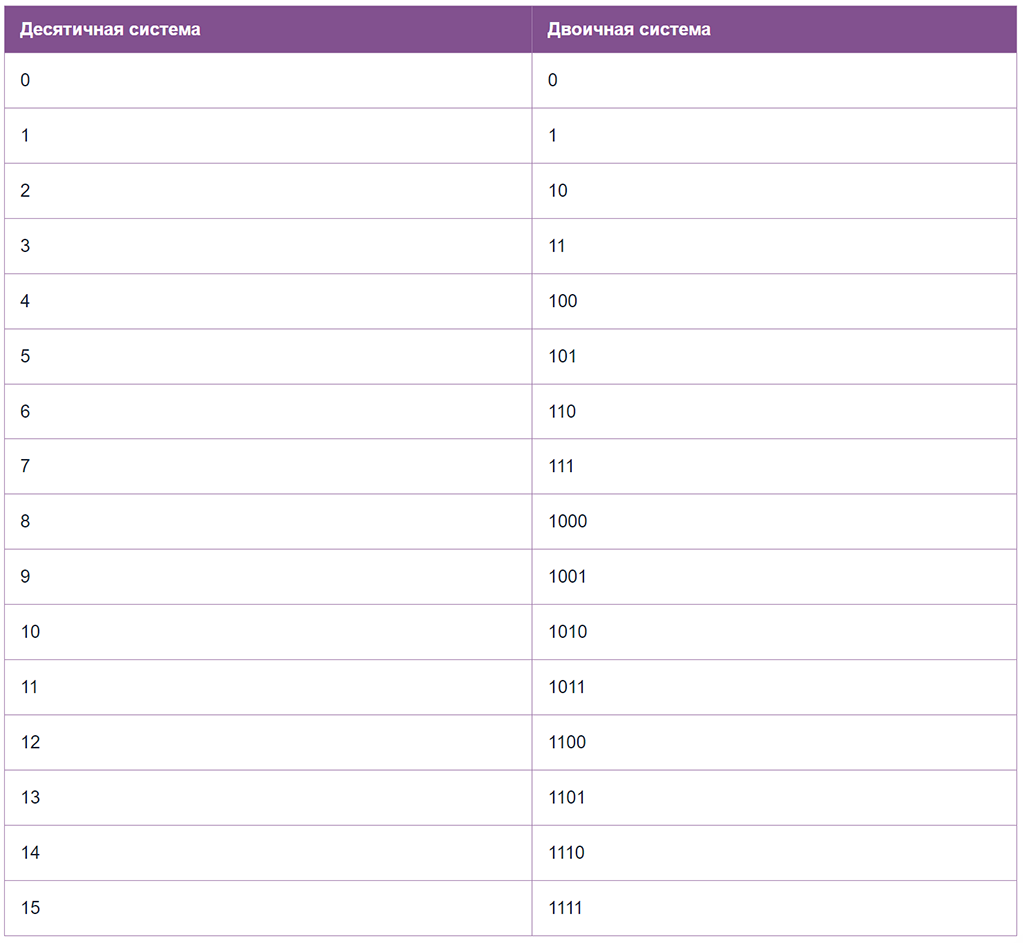
#### Согласно Википедии:

Бит (от англ. binary digit — двоичное число; также игра слов: англ. bit — кусочек, частица) — единица измерения количества информации. 1 бит информации — это символ или сигнал, который может принимать два значения: включено или выключено, да или нет, высокий или низкий, заряженный или незаряженный; в двоичной системе исчисления это 1 (единица) или 0 (ноль).

### С какими данными работают побитовые операторы?

Побитовые операции в Java осуществляются только над целыми числами. А целые числа хранятся в памяти компьютера в виде набора битов. Можно сказать, что компьютер переводит любую информацию в двоичную систему счисления (в набор битов) и только потом взаимодействует с ней. Но как устроена двоичная система счисления? В десятичной системе счисления у нас есть всего 10 символов: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. С помощью этих символов мы ведем счет. После 9 идет 10, после 19 — 20, после 99 — 100, после 749 — 750. То есть мы используем комбинацию имеющихся 10 символов и можем с их помощью считать «от нуля и до обеда». В двоичной системе счисления вместо десяти символов есть всего два — 0, 1. Но комбинируя эти символы по тому же принципу, что и в десятичной системе, можем считать бесконечно долго.

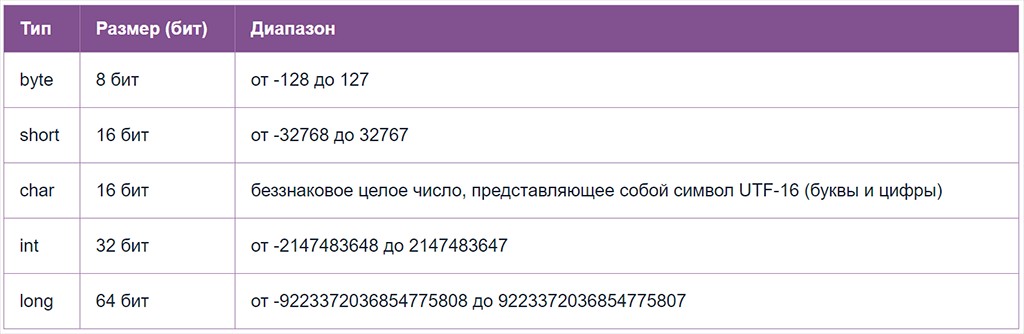
Продемонстрируем счет от 0 до 15 в десятичной системе и в двоичной:

Как видим, все не так уж и сложно. Помимо битов, есть другие знакомые единицы измерения информации — байты, килобайты, мегабайты, гигабайты и тд. Ты, наверно, знаешь, что в 1 байте — 8 бит. Что это значит? Это значит, что 8 битов подряд занимают 1 байт. Вот примеры, какими могут быть байты:

00000000 - 1 байт

10110010 - 1 байт

01011011 - 1 байт

Количество возможных неповторяющихся комбинаций битов в одном байте — 256 (28 = 256). Но вернемся ближе к Java. Есть такой целочисленный тип данных — byte. Данный тип может принимать значения от -128 до 127 и одно число в памяти компьютера занимает ровно 8 бит, или 1 байт. Одно число этого типа занимает ровно 1 byte памяти компьютера. И здесь названия совпадают не случайно. Как мы помним, 1 байт может хранить 256 различных значений. И одно число типа byte может принимать 256 различных значений (128 отрицательных, 127 положительных и 1 ноль). Каждому значению числа byte соответствует уникальный набор из восьми битов. Так обстоят дела не только с типом byte, но и со всеми целочисленными типами. Тип byte приведен в пример как самый маленький. Ниже в таблице представлены все целочисленные типы Java и занимаемое ими место в памяти: Рассмотрим тип int. Он может хранить 2147483648 отрицательных, 2147483647 положительных значений и один ноль. Итого:

2147483648 + 2147483647 + 1 = 4294967296.

Данный тип занимает в памяти компьютера 32 бита. Количество возможных комбинаций из набора 32-ух нулей и единиц равно:

232 = 4294967296.

То же число, что и у количества значений, вмещаемых в тип int. Это всего лишь демонстрация взаимосвязи между диапазоном значений типа данных и его размером (количество бит в памяти). Любое число любого типа в Java можно перевести в двоичную систему счисления. Давай посмотрим, как легко это можно сделать это с помощью Java языка. Будем учиться на примере типа int. У данного типа есть свой класс-обертка — Integer. А у него — toBinaryString, который и сделает за нас всю работу

int число занимает 32 бита. Но когда мы выводим число 10 в примере выше, мы видим в консоли 1010. Это потому, что ведущие нули не выводятся на печать. Если бы они выводились, вместо 1010 мы бы видели в консоли 00000000000000000000000000001010. Но для удобства восприятия все ведущие нули опускаются. Не так уж и сложно до тех пор, пока не задашься вопросом: а что с отрицательными числами? Он воспринимает информацию только в двоичной системе. Получается, что знак минус также необходимо прописывать двоичным кодом. Это можно сделать с помощью прямого или дополнительного кода.

### Прямой код

Способ представления чисел в двоичной системе счисления, при котором старший разряд (крайний левый бит) отводится под знак числа. Если число положительное, в крайний левый бит записывается 0, если отрицательное — 1.

Рассмотрим это на примере 8-ми битного числа:

Подход несложный и в принципе понятный. Однако у него есть недостатки: возникают трудности с выполнением математических операций. К примеру со сложением отрицательных и положительных чисел. Их нельзя складывать, если не провести дополнительные манипуляции.

### Дополнительный код

Используя дополнительный код, можно избежать недостатков прямого кода. Для получения дополнительного кода числа есть несложный алгоритм. Попробуем получить дополнительный код числа -5. Представим это число с помощью дополнительного кода в двоичной системе счисления. Шаг 1. Получаем представление отрицательного числа с помощью прямого кода. Для -5 это будет 10000101. Шаг 2. Инвертируем все разряды, кроме разряда знака. Заменим все нули на единицы, а единицы на нули везде, кроме крайнего левого бита.

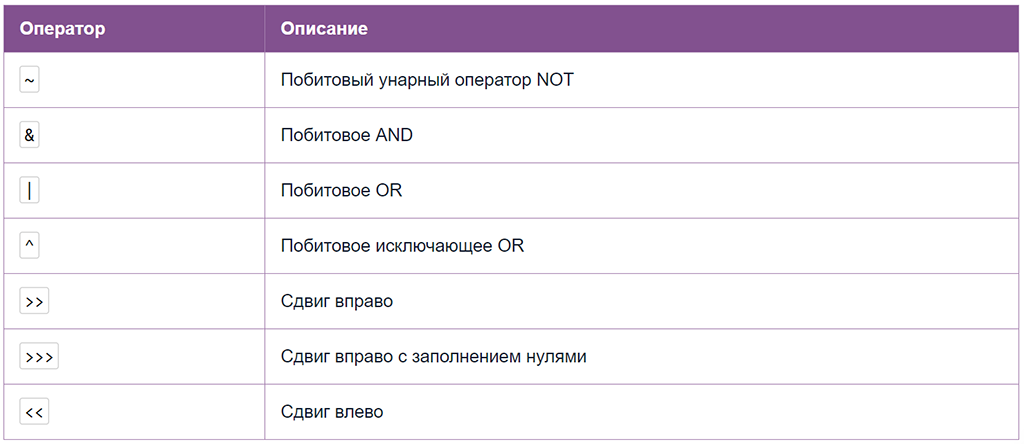
10000101 => 11111010

Шаг 3. К полученному значению прибавим единицу:

11111010 + 1 = 11111011

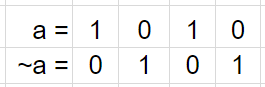
Готово. Мы получили значение числа -5 в двоичной системе счисления с использованием дополнительного кода. Это важно для понимания дальнейшего материала, так как в Java для хранения отрицательных чисел в битах используется дополнительный код.

## Типы побитовых операций

Теперь, когда мы разобрались со всеми вводными, поговорим о побитовых операциях в Java. Побитовая операция осуществляется над целыми числами, и ее результатом будет целое число. В процессе число переводится в двоичную систему, над каждым битом выполняется операция, и результат приводится обратно в десятичную систему. Список операций — в таблице ниже: 

Как мы уже выяснили числа можно представить в виде набора битов. Побитовые операции осуществляют операции как раз над каждым битом такого представления. Возьмем NOT, AND, OR, XOR. Вспомним, что недавно мы рассматривали таблицы истинности, только для логических операндов. В данном случае те же операции применяются к каждому биту целого числа.

### Побитовый унарный оператор NOT ~

Данный оператор заменяет все нули на единицы, а единицы — на нули. Предположим, у нас есть число 10 в десятичной системе исчисления. В двоичной системе это число равно 1010. Если применить к данному числу унарный побитовый оператор отрицания, мы получим примерно следующее: 

Давай взглянем как это выглядит в Java коде:

**public** **static** **void** main(String[] args) {

**int** a = 10;

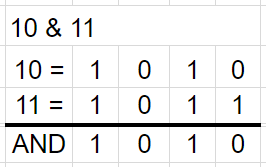
System.out.println(" a = " + a + "; binary string: " + Integer.toBinaryString(a));

System.out.println("~a = " + ~a + "; binary string: " + Integer.toBinaryString(~a));

}

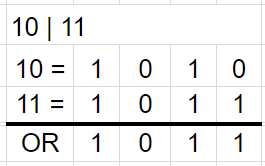
В первой строке мы получили значение в двоичной системе счисления без ведущих нулей. Хоть мы их не видим, они есть. Об этом говорит вторая строка, в которой все биты трансформировались в обратные. Именно поэтому мы видим так много ведущих единиц. Это бывшие ведущие нули, которые игнорировались компилятором при выводе в первой строки.

### Побитовый оператор AND

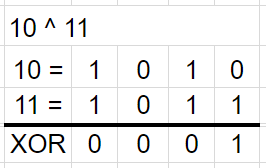
Данный оператор применим к двум числам. Он производит операцию AND между битами каждого числа. Рассмотрим пример: 

Данная операция осуществляется над двумя числами.

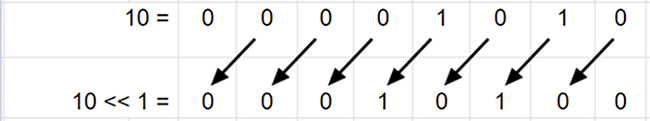
### Побитовый оператор OR

OR применим к двум числам. Он производит операцию OR между битами каждого числа: 

### Побитовая операция, исключающее ИЛИ (XOR)

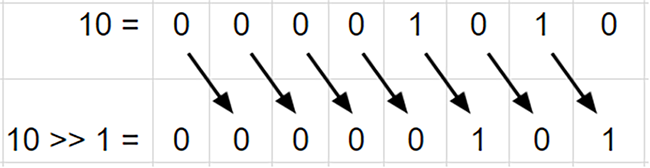
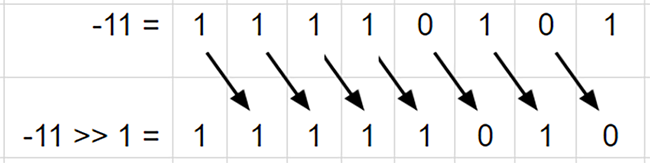
Взглянем на тот же пример, но с новой операцией: 

### Побитовый сдвиг влево

Данный оператор применим к двум операндам, то есть в операции x << y, биты числа x сдвинутся на y позиций влево. Что это значит? Рассмотрим на примере операции 10 << 1 Результатом операции будет число 20 в десятичной системе. Как видно из схемы выше, все биты сдвигаются влево на 1. При этой операции значение старшего бита (крайнего левого) теряется. А самый младший бит (крайний правый) заполняется нулем. Что можно сказать об этой операции?

1. Сдвигая биты числа X на N битов влево мы умножаем число X на 2N.
2. Но! У нас может измениться знак числа, если бит со значением 1 займет крайнее левое положение.
3. Если осуществлять сдвиг влево бесконечно долго, число просто превратится в 0.

### Побитовый сдвиг вправо

Данный оператор применим к двум операндам. Т.е. в операции x >> y, биты числа x сдвинутся на y позиций вправо. Рассмотрим другой пример. Схематично разберем операцию 10 >> 1. Сдвинем все биты числа 10 на одну позицию вправо: При операции сдвига мы теряем правые биты. Они попросту исчезают. Крайний левый бит — показатель знака числа (0 — число положительное, 1 — отрицательное). Поэтому в итоговом значении он ставится таким же, как и в исходном числе. Пример с отрицательным числом: Крайний правый бит потерялся, а крайний левый бит скопирован из исходного числа, как почетный показатель знака числа.

Теперь. Что можно сказать о числах, над которыми осуществляется сдвиг вправо? Они делятся на 2. Каждый раз, осуществляя сдвиг на один бит вправо мы делим исходное число на 2. Если число нацело на 2 не делится, результат будет округлен в сторону минус бесконечности (в меньшую сторону). Но это работает, только если мы сдвигаем биты ровно на 1. А если на 2 бита, делим на 4. На 3 бита — делим на 8. На 4 бита — на 16. Видишь? Степени двойки… При сдвиге числа X на N битов вправо, мы делим число X на 2 в степени N. Демонстрация:

**public** **class** BitOperationsDemo {

**public** **static** **void** main(String[] args) {

**for** (**int** i = 1; i <= 10; i++) {

**int** shiftOperationResult = 2048 >> i;

**int** devideOperationResult = 2048 / (**int**) Math.pow(2, i);

System.out.println(shiftOperationResult + " - " + devideOperationResult);

}

}

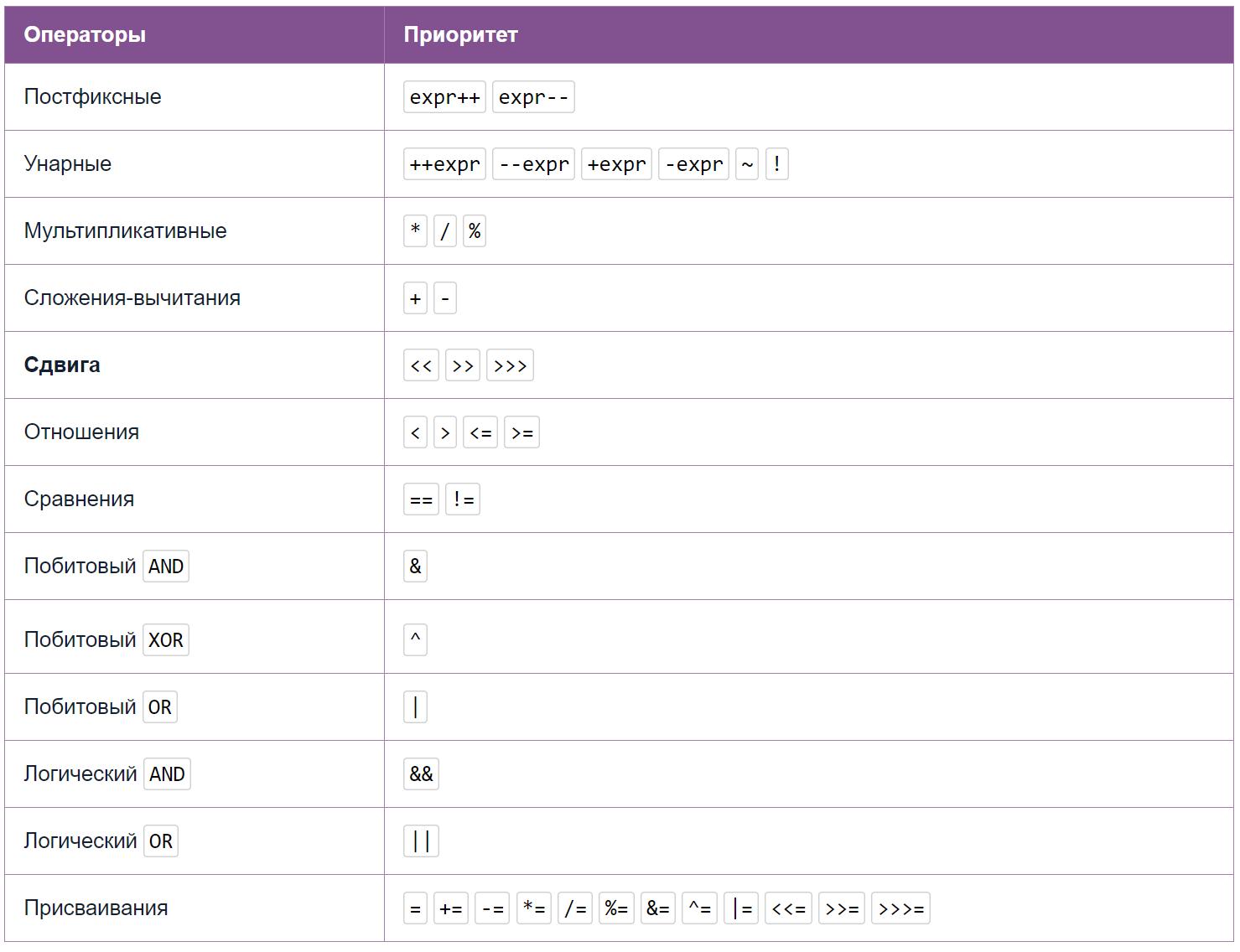
}

Что тут происходит?

1. Цикл, в котором переменная i наращивается от 1 до 10.
2. Каждую итерацию мы вычисляем 2 значения:
   * в переменную shiftOperationResult записываем результат сдвига числа 2048 на i битов вправо;
   * в переменную devideOperationResult записываем результат деления числа 2048 на 2 в степени i.
3. Попарно выводим два полученных значения.

Результат выполнения программы таков: *1024 - 1024 512 - 512 256 - 256 128 - 128 64 - 64 32 - 32 16 - 16 8 - 8 4 - 4 2 – 2*

### Приоритет операций в Java

Как и в математике, в Java есть приоритет операций. В таблице ниже приведен приоритет (от высшего к низшему) рассмотренных нами операций. 

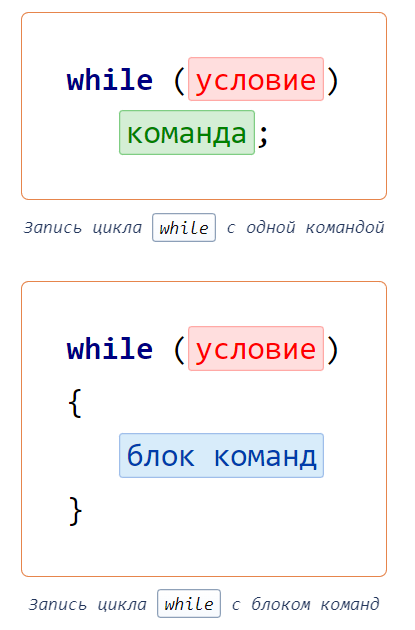
# Циклы.

## Цикл while

Оператор if-else существенно расширил наши возможности программирования: теперь можно писать программы, которые выполняют разные действия в разных ситуациях. Но есть еще одна вещь, которая сделает наши программы на порядок мощнее — это **циклы**.

В Java есть 4 вида циклов: **while**, **for**, **for-each** и **do-while**. Сейчас мы разберем самый первый из них.

Оператор цикла **while** очень простой и состоит всего из двух частей: *условие* и *тело цикла*. Тело цикла выполняется снова и снова, пока условие равно true. Общий вид цикла while такой:



Все очень просто. *Команда* или *блок команд* выполняются снова и снова, пока *условие цикла* истинно — равно true.

Это работает так: сначала проверяется *условие*, и если оно истинно, выполняется *тело цикла* (*команда* или *блок команд*), затем снова проверяется *условие* и снова выполняется *тело цикла*. И так до тех пор, пока *условие* не станет ложным.

Если условие **всегда** будет **истинно**, значит программа никогда не прекратит работать и будет всегда выполнять цикл.

Если же в **самый первый раз** условие было **ложно**, тело цикла не выполнится ни разу.

## 3. Примеры циклов

Вот несколько практических примеров использования циклов.

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| **int** n = 5;  **while** (n > 0)  {  System.out.println(n);  n--;  } | На экран будут выведены 5 строк:  5  4  3  2  1 |
| **Код** | **Пояснение** |
| **int** n = 0;  **while** (n < 10)  {  System.out.println(n);  n++;  } | На экран будут выведены 10 строк:  0  1  ...  8  9 |

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| Scanner console = **new** Scanner(System.in);  **while**(console.hasNextInt())  {  **int** x = console.nextInt();  } | Программа считывает с клавиатуры числа до тех пор, пока вводят именно числа. |
| **Код** | **Пояснение** |
| **while** (true)  System.out.println("C"); | Программа будет бесконечно печатать на экран букву С. |

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| Scanner console = **new** Scanner(System.in);  **boolean** isExit = false;  **while** (!isExit)  {  String s = console.nextLine();  isExit = s.equals("exit");  } | Программа будет вводить строки с клавиатуры,  пока не будет введена строка exit. |

Функция equals() в предыдущем примере используется для сравнения строк. Если строки равны, функция вернет результат — true, если строки не равны, то вернет false.

## 1. Команда break

Давайте разберем пример из предыдущего урока:

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| Scanner console = **new** Scanner(System.in);  **boolean** isExit = false;  **while** (!isExit)  {  String s = console.nextLine();  isExit = s.equals("exit");  } | Программа будет считывать строки с клавиатуры, пока не будет введена строка exit. |

Программа читает с консоли строки, пока не будет введено слово exit. Если ввести это слово, переменная isExit станет true, а условие цикла !isExit примет значение false, и цикл завершится.

В Java есть специальный оператор break, который позволяет упростить написание такой логики. Если выполнить команду break внутри цикла, он немедленно завершается: программа начнет выполнять команды, которые идут после цикла. Команда очень лаконична:

**break**;

Вот как можно переписать только что разобранный пример с использованием break:

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| Scanner console = **new** Scanner(System.in);  **while** (true)  {  String s = console.nextLine();  **if** (s.equals("exit"))  **break**;  } | Программа будет считывать строки с клавиатуры, пока не будет введена строка exit. |

## 2. Команда continue

Но break — не единственный оператор, который позволяет управлять поведением цикла. В Java еще один оператор — continue. Если выполнить команду continue внутри цикла, текущий виток цикла завершится досрочно.

Выполнение тела цикла один раз называется витком цикла. Команда continue прерывает текущий виток цикла, но не сам цикл, в отличие от команды break. Команда тоже лаконична:

**continue**;

Этот оператор continue очень удобно использовать в цикле, если мы хотим «пропускать» выполнение тела цикла для некоторых ситуаций.

Задача: мы хотим написать программу, которая выводит на экран 20 чисел от 1 до 20, и при этом пропускает числа, которые делятся на 7. Вот как мог бы выглядеть этот код:

|  |  |
| --- | --- |
| **Код** | **Пояснение** |
| **int** i = 1;  **while** (i <= 20)  {  **if** ( (i % 7) == 0 )  **continue**;  System.out.println(i);  i++;  } | Программа выводит на экран числа с 1 по 20. Если число делится на 7 (остаток от деления на 7 равен 0), вывод на экран пропускается. |

На самом деле такой код работать не будет, потому что i навсегда застынет на цифре 7. Ведь вызов continue привел к тому, что пропустились две команды — System.out.println(i) и i++. Поэтому достигнув значения 7, переменная i перестанет меняться, и цикл будет выполняться вечно.

Мы специально написали такой код, т.к. это очень распространенная ошибка. Как же его исправить?

Тут есть два варианта:

**Вариант 1:** изменить i перед вызовом continue, но после вызова i % 7

**Вариант 2:** увеличивать i всегда в начале цикла. Но тогда стартовое значение i должно быть 0.

|  |  |
| --- | --- |
| **Вариант 1** | **Вариант 2** |
| **int** i = 1;  **while** (i <= 20)  {  **if** ( (i % 7) == 0 )  {  i++;  **continue**;  }    System.out.println(i);  i++;  } | **int** i = 0;  **while** (i < 20)  {  i++;  **if** ( (i % 7) == 0)  **continue**;  System.out.println(i);  } |

# Цикл for

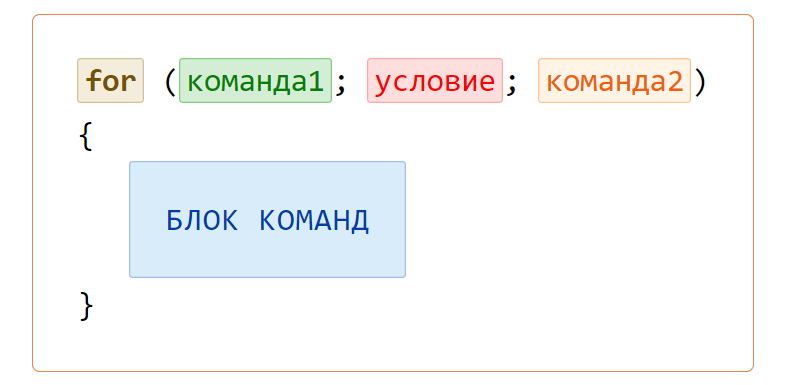
## 1. Сравнение циклов: for vs while

Цикл while можно использовать во всех случаях, когда команду или группу команд нужно выполнить несколько раз. Но из всех случаев один случай стоит выделить особо.

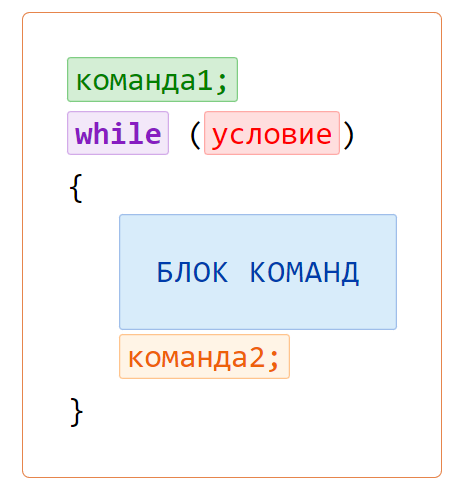
Это случай, когда программист (автор программы) заранее знает, сколько раз должен выполниться его цикл. Обычно это решается введением специальной переменной-счетчика, и каждую итерацию цикла переменная увеличивается (или уменьшается) на 1.

Вроде бы все работает как надо, но не очень удобно. Перед циклом мы задаем стартовое значение переменной-счетчика, затем в условии проверяем, не достигла ли она уже финального значения. А вот меняем ее обычно в самом конце тела цикла.

А если тело цикла большое? Или у нас несколько вложенных циклов? В общем, хотелось бы в таких случаях собрать всю эту информацию о переменной-счетчике в одном месте. Так в Java появился цикл for. Выглядит он тоже не очень сложно:

****Если у цикла while в круглых скобках было только условие, тут еще добавились две команды, разделенные точкой с запятой.

Все на самом деле проще, чем кажется: компилятор превращает цикл for в обычный цикл while таким образом:



Или лучше продемонстрируем на примере: два примера ниже идентичны.

|  |  |
| --- | --- |
| **Вариант 1** | **Вариант 2** |
| **for** (**int** i = 0; i < 20; i++)  {  System.out.println(i);  } | **int** i = 0;  **while** (i < 20)  {  System.out.println(i);  i++;  } |

Мы просто взяли и собрали в одном месте весь код, который касается переменной-счетчика — i.

команда1 в цикле for выполняется только один раз перед самим циклом (это видно на втором примере)

команда2 выполняется столько же раз, сколько и тело цикла, но будет выполняться каждый раз уже после того, как выполнится все тело цикла

## 2. Где используется цикл for

Цикл for, наверное, самый используемый вариант цикла в Java. Он применяется везде, т.к. это просто более понятная и удобная форма записи цикла while для программистов. Практически любой цикл while можно преобразовать в цикл for.

Примеры:

|  |  |
| --- | --- |
| **Цикл while** | **Цикл for** |
| **int** i = 3;  **while** (i >= 0)  {  System.out.println(i);  i--;  } | **for** (**int** i = 3; i >= 0; i--)  {  System.out.println(i);  } |
| **int** i = 0;  **while** (i < 3)  {  System.out.println(i);  i++;  } | **for** (**int** i = 0; i < 3; i++)  {  System.out.println(i);  } |
| **boolean** isExit = false;  **while** (!isExit)  {  String s = console.nextLine();  isExit = s.equals("exit");  } | **for** (**boolean** isExit = false; !isExit; )  {  String s = console.nextLine();  isExit = s.equals("exit");  } |
| **while** (true)  System.out.println("C"); | **for** (; true; )  System.out.println("C"); |
| **while** (true)  {  String s = console.nextLine();  **if** (s.equals("exit"))  **break**;  } | **for** (; true; )  {  String s = console.nextLine();  **if** (s.equals("exit"))  **break**;  } |

Обратите внимание на последний пример. Там пропущены команды работы со счетчиком цикла. Нет счетчика — нет команд.

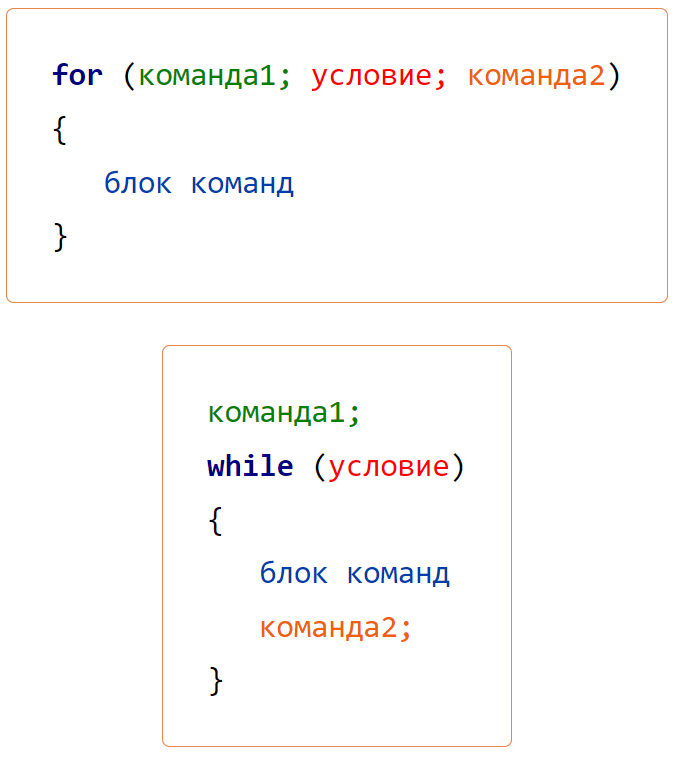
Java разрешает не писать в цикле for «команду по инициализации счетчика» и «команду по изменению счетчика». И даже выражение-условие цикла можно не писать.

## 3. Нюансы использования цикла for

Важный момент по использованию цикла for и операторов break и continue.

Оператор break в цикле for работает так же, как и в цикле while — приводит к немедленному завершению цикла. А вот continue приводит к пропуску тела цикла, но не команды2 (которая меняет счетчик цикла).

Давайте еще раз посмотрим, как связаны for и while.



Если в блоке команд цикла for вызвать оператор continue, остаток блока команд будет пропущен, а вот команда2 (которая работает с переменной-счетчиком цикла for) все равно выполнится.

Вернемся к нашему примеру с пропуском чисел, которые делятся на 7.

|  |  |
| --- | --- |
| **Этот код зациклится** | **Этот код будет отлично работать** |
| **int** i = 1;  **while** (i <= 20)  {  **if** ( (i % 7) == 0) **continue**;  System.out.println(i);  i++;  } | **for** (**int** i = 1; i <= 20; i++)  {  **if** ( (i % 7) == 0) **continue**;  System.out.println(i);  } |

Код с использованием цикла while работать не будет — i никогда не превысит 7. А вот код с циклом for отлично будет работать.

# Цикл do-while

## 1. Обратный цикл

В Java есть еще одна разновидность цикла while — цикл do-while. Он очень похож на while и тоже состоит всего из двух частей: «условие» и «тело цикла». Тело цикла выполняется снова и снова, пока условие равно true. Общий вид цикла do-while такой:

# 

Но если в цикле while последовательность выполнения будет такой: *условие*, *тело цикла*, *условие*, *тело цикла*, *условие*, *тело цикла*, ...

То в do-while она будет немного другой: *тело цикла*, *условие*, *тело цикла*, *условие*, *тело цикла*, ...

Фактически разница между while и do while только в том, что *тело цикла* в цикле do-while выполнятся как минимум один раз.

## 2. Выгоды от использования цикла do-while

Собственно, единственное отличие цикла do-while от цикла while как раз и состоит в том, что *тело цикла* в цикле do-while выполнятся как минимум один раз.

Цикл do-while обычно используют именно тогда, когда нет смысла проверять условие, если тело цикла не выполнилось. Например, в *теле цикла* проходят какие-нибудь вычисления, и их результаты используются в *условии*.

Пример:

Программа вводит строки с клавиатуры, пока не введено слово exit

|  |  |
| --- | --- |
| **while** | **do while** |
| String s;  **while** (true)  {  s = console.nextLine();  **if** (s.equals("exit"))  **break**;  } | String s;  **do**  {  s = console.nextLine();  }  **while** (!s.equals("exit")); |

операторы break и continue в цикле do-while работают точно так же, как в цикле while.

# Цикл for-each

## 1. Предыстория появления итератора

Вы уже познакомились со множеством HashSet. И если вы действительно разбирались с ним, а не просто читали лекцию, должны были задаться вопросом:

**А как вывести список всех элементов HashSet на экран?** Ведь методов get() и set() у множества нет!

И HashSet в своей проблеме не одинок. Кроме HashSet, есть еще много различных коллекций, у которых нельзя получить элемент по номеру, ведь четкий порядок элементов отсутствует.

Программисты в свое время изобрели много сложных структур данных, таких как граф, дерево. Или, например, список списков.

Многие контейнеры меняют порядок своих элементов при добавлении новых или удалении существующих элементов. Например, есть список, который хранит элементы в отсортированном порядке, и при добавлении нового элемента он практически всегда вставляется в середину списка.

Вот мы и получаем ситуацию, что контейнер, содержащий элементы, есть, сами элементы тоже есть, а фиксированного порядка нет.

Но допустим, мы хотим скопировать все элементы из такой коллекции в массив или список. Нам нужно получить все элементы. Нам все равно в каком порядке мы их обойдем, главное — не повторяться. Как нам это сделать?

## 2. Итератор в коллекции

В качестве решения вышеописанной проблемы было предложено решение — *итератор*.

Итератор — это специальный объект у коллекции, который помогает обойти все элементы коллекции и не повторяться.

Получить итератор у любой коллекции можно с помощью кода:

Iterator<Тип> it = имя.iterator();

Где имя — это имя переменной-коллекции, Тип — это тип элементов коллекции. iterator() — это метод коллекции. it — это имя переменной-объекта-итератора.

У объекта-итератора есть 3 метода:

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| Тип next() | Возвращает очередной элемент коллекции |
| **boolean** hasNext() | Проверяет, есть ли еще не пройденные элементы |
| **void** remove() | Удаляет текущий элемент коллекции |

Эти методы чем-то похожи на методы класса Scanner: nextInt() и hasNextInt().

Метод next() возвращает очередной элемент коллекции, у которой мы получили итератор.

Метод hasNext() проверяет, есть ли еще элементы в коллекции, которые итератор не вернул.

Вот как можно вывести на экран все элементы множества HashSet:

|  |  |
| --- | --- |
| **Код** | **Примечания** |
| HashSet<String> set = **new** HashSet<String>();  set.add("Привет");  set.add("Hello");  set.add("Hola");  set.add("Bonjour");  set.add("Cialo");  set.add("Namaste");  Iterator<String> it = set.iterator();  **while** (it.hasNext())  {  String str = it.next();  System.out.println(str);  } | Создаем объект типа HashSet, который хранит элементы типа String.   Заносим в set приветствия на разных языках.     Получаем объект-итератор у множества set. Пока есть еще элементы  Получаем следующий элемент Выводим элемент на экран |

## 3. Цикл for-each

Основной минус итератора в том, что с его использованием код получается еще более громоздким, чем с использованием цикла for.

Давайте для сравнения выведем на экран список с помощью цикла for и с помощью итератора:

|  |  |
| --- | --- |
| **Итератор** | **Цикл for** |
| ArrayList<String> list = **new** ArrayList<String>();  Iterator<String> it = list.iterator();  **while** (it.hasNext())  {  String str = it.next();  System.out.println(str);  } | ArrayList<String> list = **new** ArrayList<String>();  **for** (**int** i = 0; i < list.size(); i++)  {  String str = list.get(i);  System.out.println(str);  } |

Да, элементы списка ArrayList гораздо лучше обходить с помощью цикла — все короче получается.

Однако разработчики Java вновь решили подсыпать нам сахарку. И на наше счастье, это опять был *синтаксический сахар*.

Они добавили в Java новый вид циклов и назвали его for-each. Вот как выглядит его использование в общем случае:

**for**(Тип имя:коллекция)

Где коллекция — это имя переменной коллекции, Тип — это тип элементов коллекции, а имя — это имя переменной, которая на каждом витке цикла принимает очередное значение из коллекции.

Этот цикл обходит все элементы коллекции с помощью скрытого итератора. Вот как он на самом деле работает:

|  |  |
| --- | --- |
| **Цикл for-each** | **Что видит компилятор: Цикл с итератором** |
| ArrayList<String> list = **new** ArrayList<String>();  **for** (String str: list)  {  System.out.println(str);  } | ArrayList<String> list = **new** ArrayList<String>();  Iterator<String> it = list.iterator();  **while** (it.hasNext())  {  String str = it.next();  System.out.println(str);  } |

Когда компилятор встретит в вашем коде цикл for-each, он просто заменит его на код справа: добавит метод получения итератора и все недостающие вызовы методов.

Программисты очень любят цикл for-each и практически всегда используют его, когда нужно обойти все элементы коллекции.

Даже обход списка ArrayList с помощью цикла for-each выглядит короче:

|  |  |
| --- | --- |
| **Цикл for-each** | **Цикл for** |
| ArrayList<String> list = **new** ArrayList<String>();  **for** (String str: list)  {  System.out.println(str);  } | ArrayList<String> list = **new** ArrayList<String>();  **for** (**int** i = 0; i < list.size(); i++)  {  String str = list.get(i);  System.out.println(str);  } |

## 4. Удаление элемента в цикле for-each

У цикла for-each есть один минус: он не умеет правильно удалять элементы. Если вы напишете такой код, получите ошибку.

|  |  |
| --- | --- |
| **Код** | **Примечание** |
| ArrayList<String> list = **new** ArrayList<String>();  list.add("Привет");  list.add("Hello");  list.add("Hola");  list.add("Bonjour");  list.add("Cialo");  list.add("Namaste");  **for** (String str: list)  {  **if** (str.equals("Hello"))  list.remove(str);  } | При удалении возникнет ошибка! |

Это очень красивый и понятный код, только работать он не будет.

**Важно!**

Нельзя менять коллекцию, пока вы обходите ее с помощью итератора.

Есть три способа обойти это ограничение.

**1 Использование другого цикла**

Если вы обходите коллекцию ArrayList, можете воспользоваться обычным циклом со счетчиком i.

|  |
| --- |
| **Код** |
| **for** (**int** i = 0; i < list.size(); i++)  {  String str = list.get(i);  **if** (str.equals("Hello"))  {  list.remove(str);  i--; // нужно уменьшить i, т.к. после удаления элементы сдвинулись  }  } |

Однако этот вариант не подходит для коллекций HashSet и HashMap

**2 Явное использование итератора**

Можно использовать итератор явно и задействовать его метод remove().

|  |  |
| --- | --- |
| **Рабочий вариант** | **Нерабочий вариант** |
| Iterator<String> it = set.iterator();  **while** (it.hasNext())  {  String str = it.next();  **if** (str.equals("Hello"))  it.remove();  } | **for** (String str: list)  {  **if** (str.equals("Hello"))  list.remove(str);  } |

Обратите внимание, что метод remove() мы вызываем у объекта-итератора! Итератор «будет знать» об удалении элемента и сможет правильно обработать эту ситуацию.

**3 Использование копии коллекции**

Также вы можете создать копию коллекции и использовать в цикле for-each коллекцию-копию, а удалять элементы из оригинальной коллекции.

|  |  |
| --- | --- |
| **Код** | **Примечание** |
| ArrayList<String> listCopy = **new** ArrayList(list);  **for** (String str: listCopy)  {  **if** (str.equals("Hello"))  list.remove(str);  } | Копию коллекции создать очень легко    Цикл использует итератор коллекции-копии. Элементы удаляются из коллекции list. |

Копия коллекции создается довольно быстро: элементы при копировании коллекции не дублируются, в новой коллекции будут храниться ссылки на те же элементы, что и в старой.

# Операторы ветвления и выбора.

Поговорим о таких управляющих конструкциях, как:

* if-then (или же if)
* if-then-else (или же if-else)
* switch-case

## Ветвление

Начнем с базовых понятий. Любая программа представляет собой набор команд, выполняемых компьютером. Чаще всего команды выполняются последовательно, друг за другом. Чуть реже (но все еще довольно часто) возникают ситуации, когда нужно изменить последовательный ход выполнения команд программы. Иногда, в зависимости от некоторых условий, бывает необходимо исполнять один блок команд вместо другого. А когда эти условия меняются, поступать наоборот. К примеру, есть ряд сайтов, посещение которых запрещено лицам, не достигшим возраста 18 лет. Обычно, при первом посещении подобного ресурса пользователя встречает некоторая форма, в которой пользователя предупреждают о возрастном ограничении и просят ввести дату его рождения. Тогда, в зависимости от данных, которые ввел пользователь, ему либо позволят войти на ресурс, либо нет. Такой функционал обеспечивается тем, что принято называть ветвлением. Приведем еще одну аналогию. Представим себя на перекрестке семи дорог. Перед нами стоит выбор: свернуть налево или направо, либо же пойти прямо. Наш выбор опирается на некоторые условия. Также у нас нет возможности идти несколькими дорогами одновременно. Т.е. в зависимости от некоторых условий, нам придется выбрать одну дорогу. Такой же принцип и с ветвлением. Теперь попробуем привести определение ветвлению. Ветвление — это алгоритмическая конструкция, в которой в зависимости от истинности некоторого условия, выполняется одна из нескольких последовательностей действий. Ветвление реализовано (скорее всего) абсолютно во всех языках программирования. В языке программирования Java есть несколько так называемых управляющих конструкций, благодаря которым можно реализовать ветвление в своей программе. Таких конструкций в языке программирования 3:

* Оператор if-then
* Оператор if-then-else
* Тернарный оператор ? :

В данной статье мы рассмотрим операторы if-else и switch-case.

### if-then

Оператор if-then, или же просто if пожалуй самый распространенный оператор. Выражение “да там 1 if написать” уже стало крылатым. Оператор if имеет следующую конструкцию:

**if** (bool\_condition) {

statement

}

В данной конструкции:

* bool\_condition — boolean выражение, результатом которого является true или false. Данное выражение называют условием.
* statement — команда (может быть не одна), которую необходимо исполнить, в случае, если условие истинно (bool\_statement==true)

Данную конструкцию можно прочитать так:

Если (bool\_condition), то {statement}

Приведем примеры:

**public** **static** **void** main(String[] args) {

Scanner scanner = **new** Scanner(System.in);

System.out.print("Сколько процентов заряда батареи осталось на вашем смартфоне?");

**int** a = scanner.nextInt();

**if** (a < 10) {

System.out.println("Осталось менее 10 процентов, подключите ваш смартфон к зарядному устройству");

}

}

В данной программе пользователю предлагается ввести количество процентов заряда батареи на его смартфоне. В случае, если осталось менее 10 процентов заряда, программа предупредит пользователя о необходимости зарядить смартфон. Это пример простейшей конструкции if. Стоит заметить, что если переменная `а` будет больше либо равна 10, то ничего не произойдет. Программа продолжит выполнять код, который следует за конструкцией if. Также заметим, что в данном случае, у конструкции if есть только одна последовательность действий для исполнения: напечатать текст, либо не делать ничего. Эта вариация ветвления с одной “ветвью”. Такое порой бывает необходимо. Например когда мы хотим обезопасить себя от неправильных значений. К примеру, мы не можем узнать количество букв в строке, если строка равна null. Примеры ниже:

**public** **static** **void** main(String[] args) {

String x = **null**;

printStringSize(x);

printStringSize("Не представляю своей жизни без ветвлений...");

printStringSize(**null**);

printStringSize("Ифы это так захватывающе!");

}

**static** **void** printStringSize(String string) {

**if** (string != **null**) {

System.out.println("Кол-во символов в строке `" + string + "`=" + string.length());

}

}

В результате выполнения метода main в консоль будет выведено:

Количество символов в строке `Не представляю своей жизни без ветвлений...`=43

Количество символов в строке `Ифы это так захватывающе!`=25

Благодаря проверке на то, что string != null, нам удалось избежать ошибок в программе. И не делать ничего в тех случаях, когда переменная string была равна null.

### if-then-else

Если при обычном if у программы есть выбор: “сделать что-то, либо не делать ничего”, то при if-else выбор программы сводится тому, чтобы “сделать либо одно, либо другое”. Вариант “не делать ничего” отпадает. Вариантов исполнения (или количество ветвей) при таком виде ветвления бывает от двух и более. Рассмотрим случай, когда варианта два. Тогда управляющая конструкция имеет следующий вид:

**if** (bool\_condition) {

statement1

} **else** {

statement2

}

Здесь:

* bool\_statement — boolean выражение, результатом которого является true или false. Данное выражение называют условием.
* statement1 — команда (может быть не одна), которую необходимо выполнить, если условие истинно (bool\_statement==true)
* statement2 — команда (может быть не одна), которую необходимо выполнить, если условие ложно (bool\_statement==false)

Данную конструкцию можно прочитать так:

Если (bool\_condition), то {statement1}

Иначе {statement2}

Приведем пример:

**public** **static** **void** main(String[] args) {

Scanner scanner = **new** Scanner(System.in);

System.out.print("Сколько процентов заряда батареи осталось на вашем смартфоне?");

**int** a = scanner.nextInt();

**if** (a < 10) {

System.out.println("Осталось менее 10 процентов, подключите ваш смартфон к зарядному устройству");

} **else** {

System.out.println("Заряда вашей батареи достаточно для того, чтобы прочитать статью на Javarush");

}

}

Тот же пример об уровне заряда батареи на смартфоне. Только если в прошлый раз программа только лишь предупреждала о необходимости зарядить смартфон, то в этот раз у нас появляется дополнительное уведомление. Разберем этот if:

**if** (a < 10) {

System.out.println("Осталось менее 10 процентов, подключите ваш смартфон к зарядному устройству");

} **else** {

System.out.println("Заряда вашей батареи достаточно для того, чтобы прочитать статью на Javarush");

}

Если a < 10 истинно (уровень заряда батареи меньше 10), программа выведет на печать один текст. Иначе, если условие a < 10 не выполняется, то программа выведет уже совсем другой текст. Доработаем также и второй наш пример, в котором мы выводили на экран количество букв в строке. В прошлый раз программа не выводила ничего, если переданная строка была равна null. Исправим этом, превратив обычный if в if-else:

**public** **static** **void** main(String[] args) {

String x = **null**;

printStringSize(x);

printStringSize("Не представляю своей жизни без ветвлений...");

printStringSize(**null**);

printStringSize("Ифы это так захватывающе!");

}

**static** **void** printStringSize(String string) {

**if** (string != **null**) {

System.out.println("Кол-во символов в строке `" + string + "`=" + string.length());

} **else** {

System.out.println("Ошибка! Переданная строка равна null!");

}

}

В методе printStringSize в конструкцию if мы добавили блок else. Теперь, если мы запустим программу, она выведет в консоль уже не 2 строки, а 4, хотя вводные (метод main) остались такими же, как и в прошлый раз. Текст, который выведет программа:

Ошибка! Переданная строка равна null!

Кол-во символов в строке `Не представляю своей жизни без ветвлений...`=43

Ошибка! Переданная строка равна null!

Кол-во символов в строке `Ифы это так захватывающе!`=25

Допустимы ситуации, когда после else следуют не команды на исполнение, а еще один if. Тогда конструкция принимает следующий вид:

If (bool\_condition1) {

statement1

} **else** **if** (bool\_condition2) {

statement2

} **else** **if** (bool\_conditionN) {

statementN

} **else** {

statementN+1

}

В данной конструкции есть несколько условий:

* bool\_condition1
* bool\_condition2
* bool\_conditionN

Количество таких условий не ограничено. К каждому условию “привязаны” свои команды:

* statement1
* statement2
* statementN

Каждый такой statement может содержать в себе 1 или более строк кода. Условия проверяются одно за другим. Как только определится первое истинное условие, будет выполнен ряд команд, “привязанных” к истинному условию. После выполнения этих команд программа выйдет из блока if, даже если впереди были еще проверки. Выражение “statementN+1” будет выполнено в случае, если ни одно из определенных выше условий не окажется истинным. Данную конструкцию можно прочитать так:

Если (bool\_condition1) то {statement1}

Иначе если (bool\_condition2) то {statement2}

Иначе если (bool\_conditionN) то {statementN}

Иначе {statementN+1}

Последняя строка в данном случае опциональна. Можно обойтись и без последнего одинокого else. И тогда конструкция примет следующий вид:

If (bool\_condition1) {

statement1

} **else** **if** (bool\_condition2) {

statement2

} **else** **if** (bool\_conditionN) {

statementN

}

И читаться будет так:

Если (bool\_condition1) то {statement1}

Иначе если (bool\_condition2) то {statement2}

Иначе если (bool\_conditionN) то {statementN}

Соответственно, в случае, если ни одно из условий не окажется истинным, то ни одна команда не будет исполнена. Перейдем к примерам. Вернемся к ситуации с уровнем заряда на смартфоне. Напишем программу, которая будет более детально информировать владельца об уровне заряда его девайса:

**public** **static** **void** main(String[] args) {

String alert5 = "Я скоро отключусь, но помни меня бодрым";

String alert10 = "Я так скучаю по напряжению в моих жилах";

String alert20 = "Пора вспоминать, где лежит зарядка";

String alert30 = "Псс, пришло время экономить";

String alert50 = "Хм, больше половины израсходовали";

String alert75 = "Всё в порядке, заряда больше половины";

String alert100 = "Я готов к приключениям, если что..";

String illegalValue = "Такс, кто-то ввел некорректное значение";

Scanner scanner = **new** Scanner(System.in);

System.out.print("Сколько процентов заряда батареи осталось на вашем смартфоне?");

**int** a = scanner.nextInt();

**if** (a <= 0 || a > 100) {

System.out.println(illegalValue);

} **else** **if** (a < 5) {

System.out.println(alert5);

} **else** **if** (a < 10) {

System.out.println(alert10);

} **else** **if** (a < 20) {

System.out.println(alert20);

} **else** **if** (a < 30) {

System.out.println(alert30);

} **else** **if** (a < 50) {

System.out.println(alert50);

} **else** **if** (a < 75) {

System.out.println(alert75);

} **else** **if** (a <= 100) {

System.out.println(alert100);

}

}

К примеру в данном случае, если пользователь введет 15, то программа выведет на экран: “Пора вспоминать, где лежит зарядка”. Несмотря на то, что 15 меньше и 30 и 50 и 75 и 100, вывод на экран будет только 1. Напишем еще одно приложение, которое будет печатать в консоль, какой сегодня день недели:

**public** **static** **void** main(String[] args) {

// Определим текущий день недели

DayOfWeek dayOfWeek = LocalDate.now().getDayOfWeek();

**if** (dayOfWeek == DayOfWeek.SUNDAY) {

System.out.println("Сегодня воскресенье");

} **else** **if** (dayOfWeek == DayOfWeek.MONDAY) {

System.out.println("Сегодня понедельник");

} **else** **if** (dayOfWeek == DayOfWeek.TUESDAY) {

System.out.println("Сегодня вторник");

} **else** **if** (dayOfWeek == DayOfWeek.WEDNESDAY) {

System.out.println("Сегодня среда");

} **else** **if** (dayOfWeek == DayOfWeek.THURSDAY) {

System.out.println("Сегодня четверг");

} **else** **if** (dayOfWeek == DayOfWeek.FRIDAY) {

System.out.println("Сегодня пятница");

} **else** **if** (dayOfWeek == DayOfWeek.SATURDAY) {

System.out.println("Сегодня суббота");

}

}

Удобно конечно, но в глазах немного рябит от обилия однообразного текста. В ситуациях, когда у нас имеются большое количество вариантов лучше использовать оператор, речь о котором пойдет ниже.

### switch-case

Альтернативой жирным if с большим количеством ветвей служит оператор switch-case. Данный оператор как бы говорит “Так, у нас есть вот такая вот переменная. Смотрите, в случае, если её значение равно `x`, то делаем то-то и то-то, если ее значение равно `y`, то делаем по-другому, а если ничему не равно из вышеперечисленного, просто делаем вот так… ” Данный оператор обладает следующей структурой.

**switch** (argument) {

**case** value1:

statement1;

**break**;

**case** value2:

statement2;

**break**;

**case** valueN:

statementN;

**break**;

**default**:

default\_statement;

**break**;

}

Рассмотрим данную структуру более подробно. argument —это некоторая переменная, значение которой мы будем сравнивать с гипотетическими различными вариантами. Переменная должна быть final. Также стоит сказать, что оператор switch поддерживает в качестве аргумента далеко не любой тип данных. Допустимые типы перечислены ниже:

* byte and Byte
* short and Short
* int and Integer
* char and Character
* enum
* String

case value1 (value2, valueN) — это значения, конкретные константы, с которыми мы сравниваем значение переменной argument. Также, каждый кейс определяет под собой набор команд, которые нужно исполнить. statement1, statement2, statementN —это команды, которые необходимо будет исполнить, если argument окажется равен одному из value. К примеру, если argument будет равен value2, то программа выполнит statement2. default и default\_statement — это “значения по умолчанию”. Если argument не будет равен ни одному из представленных value, то сработает ветка default и выполнится команда default\_statement. default и default\_statement являются опциональными атрибутами оператора switch-case. break — можно заметить, что в конце каждого case блока следует оператор break. Данный оператор также является опциональным и служит для того, чтобы разграничивать код различных кейсов. Порой бывает необходимо выполнять одинаковые действия в разных кейсах: тогда данные кейсы можно объединить. В таком случае ключевое слово break — опускают, и структура оператора switch-case будет выглядеть следующим образом:

**switch** (argument) {

**case** value1:

statement1;

**break**;

**case** valueX:

**case** valueY:

statementXY;

**break**;

}

Стоит обратить внимание: между `case valueX:` и `case valueY:` нет оператора break. Здесь, если argument будет равен value1, выполнится statement1. А если argument будет равен valueX либо valueY, выполнится statementXY. Разбавим тяжелую для восприятия теорию на легкую для восприятия практику. Перепишем пример с днями недели с использованием оператора switch-case.

**public** **static** **void** main(String[] args) {

// Определим текущий день недели

DayOfWeek dayOfWeek = LocalDate.now().getDayOfWeek();

**switch** (dayOfWeek) {

**case** SUNDAY:

System.out.println("Сегодня воскресенье");

**break**;

**case** MONDAY:

System.out.println("Сегодня понедельник");

**break**;

**case** TUESDAY:

System.out.println("Сегодня вторник");

**break**;

**case** WEDNESDAY:

System.out.println("Сегодня среда");

**break**;

**case** THURSDAY:

System.out.println("Сегодня четверг");

**break**;

**case** FRIDAY:

System.out.println("Сегодня пятница");

**break**;

**case** SATURDAY:

System.out.println("Сегодня суббота");

**break**;

}

}

Теперь напишем программу, которая выводит информацию о том, является ли сегодняшний день будним днем или выходным, используя оператор switch-case.

**public** **static** **void** main(String[] args) {

// Определим текущий день недели

DayOfWeek dayOfWeek = LocalDate.now().getDayOfWeek();

**switch** (dayOfWeek) {

**case** SUNDAY:

**case** SATURDAY:

System.out.println("Сегодня выходной");

**break**;

**case** FRIDAY:

System.out.println("Завтра выходной");

**break**;

**default**:

System.out.println("Сегодня рабочий день");

**break**;

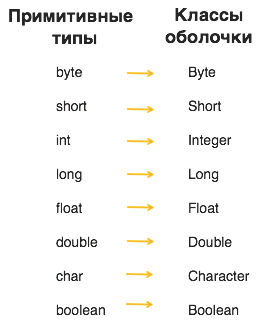
}

}

Немного поясним. В данной программе мы получаем enum DayOfWeek, который обозначает текущий день недели. Далее мы смотрим, равняется ли значение нашей переменной dayOfWeek значениям SUNDAY либо SATURDAY. В случае, если это так, программа выводит “Сегодня выходной”. Если же нет, то мы проверяем, равняется ли значение переменной dayOfWeek значению FRIDAY. В случае, если это так, программа выводит “Завтра выходной”. Если же и в этом случае нет, то вариантов у нас немного, любой оставшийся день является будним днем, поэтому по умолчанию, если сегодня НЕ пятница, НЕ суббота и НЕ воскресение программа выведет “Сегодня рабочий день”.

# Классы – обёртки

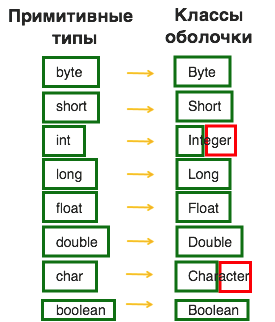
 у примитивных типов есть **объекты-аналоги** - так называемые **"классы оболочки"**, или **"wrapper"** (с англ. "обертка, упаковка"):



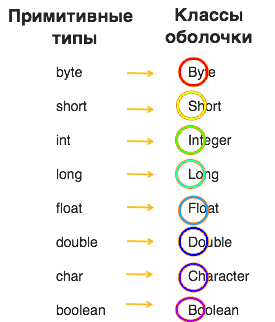
Класс называется "оболочкой" потому, что он, по сути, копирует то, что уже существует, **но добавляет новые возможности для работы с привычными типами**.

###### **О названиях**

Имена классов нетрудно запомнить - они повторяют имена примитивных типов данных:



Обратите внимание - все классы оболочки пишутся с большой буквы:



**Примитивы** и их аналоги, **классы оболочки**, существуют параллельно, потому что у каждого есть преимущества.

Например, обычный **int**занимает меньше места, и если нет необходимости проводить над ним особые операции, Ваш компьютер будет работать быстрее.

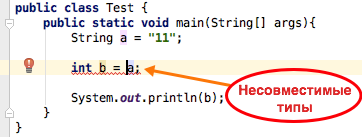
В свою очередь, с помощью класса-оболочки **Integer** можно выполнять специальные операции - например, перевести текст в число (с помощью метода **.parseInt()**для Integer-а ).

###### **Метод .parseInt()**

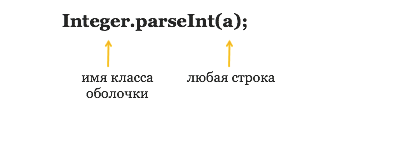
Иногда в объекте типа **String** содержится число, и Вам нужно с ним работать дальше - умножать, делить, в степень возводить. Но нельзя! Строка же. Что делать?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Test {      public static void main(String[] args){          String a = "11";            int b = a;            System.out.println(b);      }  } |

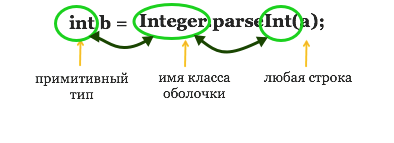
Нам видно, что **11** - это **число**, но выдает ошибку:



Можно исправить это с помощью **.parseInt()**(для int; для long будет .parseLong(), и так далее). Он объявляется так:



Название класса оболочки зависит от типа примитива, которому присваивается значение:



Пример:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class Test {      public static void main(String[] args){          String a = "11";            int b = Integer.parseInt(a);            System.out.println(b);      }  } |

Теперь все работает и на экран выводится число 11

У примитивов в программировании, и в Java в частности, есть множество преимуществ: они занимают мало памяти, за счет чего повышается эффективность работы программы, и четко разделены по диапазонам значений. Однако в процессе изучения Java мы уже не раз, словно мантру, повторяли — “**в Java все является объектом**”. А ведь примитивы — прямое опровержение этих слов. Объектами они не являются. Получается, принцип “все является объектом” является ложным? На самом деле нет. В Java у каждого примитивного типа есть свой брат-близнец — **класс-обертка** (**Wrapper**). Что такое обертка? **Обертка — это специальный класс, который хранит внутри себя значение примитива.** Но поскольку это именно класс, он может создавать свои экземпляры. Они будут хранить внутри нужные значения примитивов, при этом будут являться настоящими объектами. Названия классов-оберток очень похожи на названия соответствующих примитивов, или полностью с ними совпадают. Поэтому запомнить их будет очень легко.

| **Wrapper Classes for Primitive Data Types** | |
| --- | --- |
| **Primitive Data Types** | **Wrapper Classes** |
| int | Integer |
| short | Short |
| long | Long |
| byte | Byte |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

Объекты классов оберток создаются так же, как и любые другие:

**public** **static** **void** main(String[] args) {

Integer i = **new** Integer(682);

Double d = **new** Double(2.33);

Boolean b = **new** Boolean(false);

}

Классы-обертки позволяют нивелировать недостатки, которые есть у примитивных типов. Самый очевидный из них — **примитивы не имеют методов**. Например, у них нет метода **toString()**, поэтому ты не сможешь, например, преобразовать число int в строку. А вот с классом-оберткой Integer — запросто.

**public** **static** **void** main(String[] args) {

Integer i = **new** Integer(432);

String s = i.toString();

}

Возникнут сложности и с обратным преобразованием. Допустим, у нас есть строка, про которую мы точно знаем, что она содержит число. Тем не менее, в случае с примитивным типом **int** мы никак не сможем это число из строки достать и превратить, собственно, в число. Но благодаря классам-оберткам такая возможность у нас появилась.

**public** **static** **void** main(String[] args) {

String s = "1166628";

Integer i = Integer.parseInt(s);

System.out.println(i);

}

**Вывод: 1166628** Мы успешно получили число из строки и присвоили его в переменную-ссылку Integer i. Кстати, по поводу ссылок. Ты уже знаешь, что параметры передаются в методы по-разному: примитивы — по значению, а объекты — по ссылке. Ты можешь использовать это знание при создании своих методов: если твой метод работает, например, с дробными числами, но тебе нужна логика именно передачи по ссылке, ты можешь передать в метод параметры Double/Float вместо double/float. Кроме того, помимо методов в классах-обертках есть очень удобные для использования статические поля. Например, представь, что перед тобой сейчас стоит задача: вывести в консоль максимально возможное число int, а после — минимально возможное. Задачка вроде элементарная, а все равно — без гугла вряд ли справишься. А классы-обертки легко позволяют решать такие “бытовые задачи”:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

System.out.println(Integer.MAX\_VALUE);

System.out.println(Integer.MIN\_VALUE);

}

}

Такие поля позволяют не отвлекаться от выполнения более серьезных задач. Не говоря уж о том, что в процессе печати числа 2147483647 (это как раз MAX\_VALUE) не мудрено и опечататься:) Кроме того, в одной из прошлых лекций мы уже обращали внимание на то, что **объекты классов-оберток являются неизменяемыми (Immutable)**.

**public** **static** **void** main(String[] args) {

Integer a = **new** Integer(0);

Integer b = **new** Integer(0);

b = a;

a = 1;

System.out.println(b);

}

**Вывод: 0** Объект, на который изначально указывала ссылка **а**, не изменил свое состояние, иначе значение b тоже изменилось бы. Как и в случае со String, вместо изменения состояния объекта-обертки в памяти создается абсолютно новый объект. Почему же создатели Java, в конечном итоге, приняли решение оставить в языке примитивные типы? Раз уж все должно являться объектом, и у нас уже есть классы-обертки, которыми можно выразить все, что выражают примитивы, почему вообще не оставить в языке только их, а примитивы удалить? Ответ прост — производительность. Примитивные типы потому и называют примитивными, потому что они лишены многих “тяжеловесных” особенностей объектов. Да, у объекта есть много удобных методов, но ведь они не всегда тебе нужны. Иногда тебе нужно просто число 33, или 2,62, или значение true/false. В ситуациях, когда все преимущества объектов не имеют значения и не нужны для работы программы, примитивы справятся с задачей гораздо лучше.

## Автоупаковка/автораспаковка

Одной из особенностей примитивов и их классов-оберток в Java является автоупаковка/автораспаковка (Autoboxing/Autounboxing) Давай разберемся с этим понятием. Как мы с тобой уже узнали ранее, Java — объектно-ориентированный язык. Это значит, что все программы, написанные на Java, состоят из объектов. Примитивы не являются объектами. Но при этом переменной класса-обертки можно присваивать значение примитивного типа. Этот процесс называется **автоупаковкой** (**autoboxing**). Точно так же переменной примитивного типа можно присваивать объект класса-обертки. **Этот процесс называется автораспаковкой (autounboxing)**. Например:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**int** x = 7;

Integer y = 111;

x = y; // автораспаковка

y = x \* 123; // автоупаковка

}

}

В строке 5 мы присваиваем примитиву x значение y, который является объектом класса-обертки Integer. Как видишь, никаких дополнительных действий для этого не нужно: **компилятор знает что int и Integer, по сути, одно и то же**. Это и есть автораспаковка. Так же происходит и автоупаковка в строке 6: объекту y легко присваивается значение примитивов (х\*123). Это пример автоупаковки. Именно поэтому добавляется слово "авто": **для присваивания ссылок-примитивов объектам их классов-оберток (и наоборот) не требуется ничего делать, все происходит автоматически**. Удобно, да? :) Еще одно очень большое удобство автоупаковки/автораспаковки проявляется в работе методов. Дело в том, что **параметры методов тоже подлежат автоупаковке и автораспаковке**. И, например, если какой-то из них принимает на вход два объекта Integer — мы легко можем передать туда обычные примитивы int!

**public** **class** Main {

**public** **static** **void** main(String[] args) {

printNumber(7);//обычный int, даже без переменной

}

**public** **static** **void** printNumber(Integer i) {

System.out.println("Вы ввели число " + i);

}

}

**Вывод: Вы ввели число 7** Точно так же работает и наоборот:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

printNumber(**new** Integer(632));

}

**public** **static** **void** printNumber(**int** i) {

System.out.println("Вы ввели число " + i);

}

}

Важный момент, о котором нужно помнить: **автоупаковка и распаковка не работают для массивов**!

**public** **class** Main {

**public** **static** **void** main(String[] args) {

**int**[] i = {1,2,3,4,5};

printArray(i);//ошибка, не компилируется!

}

**public** **static** **void** printArray(Integer[] arr) {

System.out.println(Arrays.toString(arr));

}

}

Попытка передать массив примитивов в метод, который принимает на вход массив объектов, вызовет ошибку компиляции. Напоследок, еще раз кратко сравним примитивы и обертки **Примитивы:**

* имеют преимущество в производительности

**Обертки:**

* Позволяют не нарушать принцип “все является объектом”, благодаря чему числа, символы и булевы значения true/false не выпадают из этой концепции
* Расширяют возможности работы с этими значениями, предоставляя удобные методы и поля
* Необходимы, когда какой-то метод может работать исключительно с объектами

# Класс String в Java

Класс **String** в Java предназначен для работы со строками в Java. Все строковые литералы, определенные в Java программе (например, "abc") — это экземпляры класса String. Давай посмотрим на его ключевые характеристики:

1. Класс реализует интерфейсы Serializable и CharSequence. Поскольку он входит в пакет java.lang, его не нужно импортировать.
2. Класс **String** в Java — это **final** класс, который не может иметь потомков.
3. Класс String — immutable класс, то есть его объекты не могут быть изменены после создания. Любые операции над объектом String, результатом которых должен быть объект класса String, приведут к созданию нового объекта.
4. Благодаря своей неизменности, объекты класса String являются потокобезопасными и могут быть использованы в многопоточной среде.
5. Каждый объект в Java может быть преобразован в строку через метод toString, унаследованный всеми Java-классами от класса Object.

## Работа с Java String

Это один из самых часто используемых классов в Java. В нем есть методы для анализа определенных символов строки, для сравнения и поиска строк, извлечения подстрок, создания копии строки с переводом всех символов в нижний и верхний регистр и прочие. Список всех методов класса String можно изучить в [официальной документации](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html). Также в Java реализован несложный механизм конкатенации (соединения строк), преобразования примитивов в строку и наоборот. Давай рассмотрим некоторые примеры работы с классом String в Java.

### Создание строк

Проще всего создать экземпляр класса String, присвоив ему значение строкового литерала:

String s = "I love movies";

Однако у класса String есть много [конструкторов](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#String--), которые позволяют:

* создать объект, содержащий пустую строку
* создать копию строковой переменной
* создать строку на основе массива символов
* создать строку на основе массива байтов (с учетом кодировок)
* и т.д.

### Сложение строк

Сложить две строки в Java довольно просто, воспользовавшись оператором +. Java позволяет складывать друг с другом и переменные, и строковые литералы:

**public** **static** **void** main(String[] args) {

String day = "День";

String and = "и";

String night = "Ночь";

String dayAndNight = day + " " + and + " " + night;

}

Складывая объекты класса String с объектами других классов, мы приводим последние к строковому виду. Преобразование объектов других классов к строковому представлению выполняется через неявный вызов метода toString у объекта. Продемонстрируем это на следующем примере:

**public** **class** StringExamples {

**public** **static** **void** main(String[] args) {

Human max = **new** Human("Макс");

String out = "Java объект: " + max;

System.out.println(out);

// Вывод: Java объект: Человек с именем Макс

}

**static** **class** Human {

**private** String name;

**public** Human(String name) {

**this**.name = name;

}

@Override

**public** String toString() {

**return** "Человек с именем " + name;

}

}

}

### Сравнение строк

Для сравнения строк можно воспользоваться методом equals():

**public** **static** **void** main(String[] args) {

String x = "Test String";

System.out.println("Test String".equals(x)); // true

}

Когда при сравнении строк нам не важен регистр, нужно использовать метод equalsIgnoreCase():

**public** **static** **void** main(String[] args) {

String x = "Test String";

System.out.println("test string".equalsIgnoreCase(x)); // true

}

### Перевод объекта/примитива в строку

Для перевода экземпляра любого Java-класса или любого примитивного типа данных к строковому представлению, можно использовать метод String.valueOf():

**public** **class** StringExamples {

**public** **static** **void** main(String[] args) {

String a = String.valueOf(1);

String b = String.valueOf(12.0D);

String c = String.valueOf(123.4F);

String d = String.valueOf(123456L);

String s = String.valueOf(true);

String human = String.valueOf(**new** Human("Alex"));

System.out.println(a);

System.out.println(b);

System.out.println(c);

System.out.println(d);

System.out.println(s);

System.out.println(human);

/\*

Вывод:

1

12.0

123.4

123456

true

Человек с именем Alex

\*/

}

**static** **class** Human {

**private** String name;

**public** Human(String name) {

**this**.name = name;

}

@Override

**public** String toString() {

**return** "Человек с именем " + name;

}

}

}

### Перевод строки в число

Часто бывает нужно перевести строку в число. У классов оберток примитивных типов есть методы, которые служат как раз для этой цели. Все эти методы начинаются со слова parse. Рассмотрим ниже перевод строки в целочисленное (Integer) и дробное (Double) числа:

**public** **static** **void** main(String[] args) {

Integer i = Integer.parseInt("12");

Double d = Double.parseDouble("12.65D");

System.out.println(i); // 12

System.out.println(d); // 12.65

}

### Перевод коллекции строк к строковому представлению

Если нужно преобразовать все элементы некоторой коллекции строк к строковому представлению через произвольный разделитель, можно использовать такие методы класса String Java:

* join(CharSequence delimiter, CharSequence... elements)
* join(CharSequence delimiter, Iterable<? extends CharSequence> elements)

Где delimiter — разделитель элементов, а elements — массив строк / экземпляр коллекции строк. Рассмотрим пример, в котором мы преобразуем список строк в строку, разделяя каждую точкой с запятой:

**public** **static** **void** main(String[] args) {

List<String> people = Arrays.asList(

"Philip J. Fry",

"Turanga Leela",

"Bender Bending Rodriguez",

"Hubert Farnsworth",

"Hermes Conrad",

"John D. Zoidberg",

"Amy Wong"

);

String peopleString = String.join("; ", people);

System.out.println(peopleString);

/\*

Вывод:

Philip J. Fry; Turanga Leela; Bender Bending Rodriguez; Hubert Farnsworth; Hermes Conrad; John D. Zoidberg; Amy Wong

\*/

}

### Разбиение строки на массив строк

Эту операцию выполняет метод split(String regex) В качестве разделителя выступает строковое регулярное выражение regex. В примере ниже произведем операцию, обратную той, что мы выполняли в предыдущем примере:

**public** **static** **void** main(String[] args) {

String people = "Philip J. Fry; Turanga Leela; Bender Bending Rodriguez; Hubert Farnsworth; Hermes Conrad; John D. Zoidberg; Amy Wong";

String[] peopleArray = people.split("; ");

**for** (String human : peopleArray) {

System.out.println(human);

}

/\*

Вывод:

Philip J. Fry

Turanga Leela

Bender Bending Rodriguez

Hubert Farnsworth

Hermes Conrad

John D. Zoidberg

Amy Wong

\*/

}

### Определение позиции элемента в строке

В языке Java String предоставляет набор методов для определения позиции символа/подстроки в строке:

1. indexOf(int ch)
2. indexOf(int ch, int fromIndex)
3. indexOf(String str)
4. indexOf(String str, int fromIndex)
5. lastIndexOf(int ch)
6. lastIndexOf(int ch, int fromIndex)
7. lastIndexOf(String str)
8. lastIndexOf(String str, int fromIndex)

Где:

1. ch — искомый символ (char)
2. str — искомая строка
3. fromIndex — позиция с которой нужно искать элемент
4. методы indexOf — возвращают позицию первого найденного элемента
5. методы lastIndexOf — возвращают позицию последнего найденного элемента

Если искомый элемент не найден, методы вернут в строке -1. Попробуем найти порядковый номер букв A, K, Z, Я в английском алфавите, но будем иметь ввиду, что индексация символов в строке в Java начинается с нуля:

**public** **static** **void** main(String[] args) {

String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

System.out.println(alphabet.indexOf('A')); // 0

System.out.println(alphabet.indexOf('K')); // 10

System.out.println(alphabet.indexOf('Z')); // 25

System.out.println(alphabet.indexOf('Я')); // -1

}

### Извлечение подстроки из строки

Для извлечения подстроки из строки класс String в Java предоставляет методы:

* substring(int beginIndex)
* substring(int beginIndex, int endIndex)

Рассмотрим, как с помощью методов определения позиции элемента и извлечения подстроки мы можем получить имя файла из его пути:

**public** **static** **void** main(String[] args) {

String filePath = "D:\\Movies\\Futurama.mp4";

**int** lastFileSeparatorIndex = filePath.lastIndexOf('\\');

String fileName = filePath.substring(lastFileSeparatorIndex + 1);

System.out.println(fileName); //9

}

### Перевод строки в верхний/нижний регистр:

Класс String предоставляет методы для перевода строки в верхний и нижний регистры:

* toLowerCase()
* toUpperCase()

Рассмотрим работу данных методов на примере:

**public** **static** **void** main(String[] args) {

String fry = "Philip J. Fry";

String lowerCaseFry = fry.toLowerCase();

String upperCaseFry = fry.toUpperCase();

System.out.println(lowerCaseFry); // philip j. fry

System.out.println(upperCaseFry); // PHILIP J. FRY

}

# Массивы в Java

* [Объявление массива](https://javarush.com/groups/posts/massivy-java#%D0%9E%D0%B1%D1%8A%D1%8F%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2%D0%B0)
* [Создание массива](https://javarush.com/groups/posts/massivy-java#%D0%A1%D0%BE%D0%B7%D0%B4%D0%B0%D0%BD%D0%B8%D0%B5-%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2%D0%B0)
* [Длина массива](https://javarush.com/groups/posts/massivy-java#%D0%94%D0%BB%D0%B8%D0%BD%D0%B0-%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2%D0%B0)
* [Инициализация массива](https://javarush.com/groups/posts/massivy-java#%D0%98%D0%BD%D0%B8%D1%86%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F)
* [Вывод массива](https://javarush.com/groups/posts/massivy-java#%D0%9A%D0%B0%D0%BA-%D0%B2%D1%8B%D0%B2%D0%B5%D1%81%D1%82%D0%B8-%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2-%D0%B2-Java-%D0%BD%D0%B0-%D1%8D%D0%BA%D1%80%D0%B0%D0%BD)

Представьте себе ячейки в камере хранения. Каждая имеет свой номер, и в каждой хранится какой-то объект “Багаж”. Или винная карта, в которой все виды вина пронумерованы и когда вы делаете заказ, вам достаточно назвать номер напитка. Или список студентов группы, в котором в первой ячейке будет записан студент “Андреев”, а в последней — “Яковлев”. Или список пассажиров самолёта, за каждым из которых закреплено место с определённым номером. В Java чтобы работать с подобными структурами, то есть множеством однородных данных, часто используют массивы в Java.

## Что такое массив?

Массив — это структура данных, в которой хранятся элементы одного типа. Его можно представить, как набор пронумерованных ячеек, в каждую из которых можно поместить какие-то данные (один элемент данных в одну ячейку). Доступ к конкретной ячейке осуществляется через её номер. Номер элемента в массиве также называют **индексом**. В случае с Java массив однороден, то есть во всех его ячейках будут храниться элементы одного типа. Так, массив целых чисел содержит только целые числа (например, типа int), массив строк — только строки, массив из элементов созданного нами класса Dog будет содержать только объекты Dog. То есть в Java мы не можем поместить в первую ячейку массива целое число, во вторую String, а в третью — “собаку”.

## Объявление массива

### Как объявить массив?

Как и любую переменную, массив в Java нужно объявить. Сделать это можно одним из двух способов. Они равноправны, но первый из них лучше соответствует стилю Java. Второй же — наследие языка Си (многие Си-программисты переходили на Java, и для их удобства был оставлен и альтернативный способ). В таблице приведены оба способа объявления массива в Java:

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Объявление массива, Java-синтаксис** | **Примеры** | **Комментарий** |
| 1. | dataType[] arrayName; | **int**[] myArray;  Object[]  arrayOfObjects; | Желательно объявлять массив именно таким способом, это Java-стиль |
| 2. | dataType arrayName[]; | **int** myArray[];  Object  arrayOfObjects[]; | Унаследованный от С/С++ способ объявления массивов, который работает и в Java |

В обоих случаях **dataType** — тип переменных в массиве. В примерах мы объявили два массива. В одном будут храниться целые числа типа int, в другом — объекты типа Object. Таким образом при объявлении массива у него появляется имя и тип (тип переменных массива). **arrayName** — это имя массива.

## Создание массива

### Как создать массив?

Как и любой другой объект, создать массив Java, то есть зарезервировать под него место в памяти, можно с помощью оператора **new**. Делается это так:

**new** typeOfArray [length];

Где **typeOfArray** — это тип массива, а **length** — его длина (то есть, количество ячеек), выраженная в целых числах (int). Однако здесь мы только выделили память под массив, но не связали созданный массив ни с какой объявленной ранее переменной. Обычно массив сначала объявляют, а потом создают, например:

**int**[] myArray; // объявление массива

myArray = **new** **int**[10]; // создание, то есть, выделение памяти для массива на 10 элементов типа int

Здесь мы объявили массив целых чисел по имени myArray, а затем сообщили, что он состоит из 10 ячеек (в каждой из которых будет храниться какое-то целое число). Однако гораздо чаще массив создают сразу после объявления с помощью такого сокращённого синтаксиса:

**int**[] myArray = **new** **int**[10]; // объявление и выделение памяти “в одном флаконе”

Обратите внимание: После создания массива с помощью **new**, в его ячейках записаны значения по умолчанию. Для численных типов (как в нашем примере) это будет 0, для boolean — false, для ссылочных типов — null. Таким образом после операции

**int**[] myArray = **new** **int**[10];

мы получаем массив из десяти целых чисел, и, пока это не изменится в ходе программы, в каждой ячейке записан 0.

Длина массива в Java

Как мы уже говорили выше, длина массива — это количество элементов, под которое рассчитан массив. **Длину массива нельзя изменить после его создания.** Обратите внимание: в Java элементы массива нумеруются с нуля. То есть, если у нас есть массив на 10 элементов, то первый элемент массива будет иметь индекс 0, а последний — 9. Получить доступ к длине массива можно с помощью переменной **length**. Пример:

**int**[] myArray = **new** **int**[10]; // создали массив целых чисел на 10 элементов и присвоили ему имя myArray

System.out.println(myArray.length); // вывели в консоль длину массива, то есть количество элементов, которые мы можем поместить в массив

Вывод программы:

10

## Инициализация массива и доступ к его элементам

Как создать массив в Java уже понятно. После этой процедуры мы получаем не пустой массив, а массив, заполненный значениями по умолчанию. Например, в случае int это будут 0, а если у нас массив с данными ссылочного типа, то по умолчанию в каждой ячейке записаны null. Получаем доступ к элементу массива (то есть записываем в него значение или выводим его на экран или проделываем с ним какую-либо операцию) мы по его индексу. Инициализация массива — это заполнение его конкретными данными (не по умолчанию). **Пример:** давайте создадим массив из 4 сезонов и заполним его строковыми значениями — названиями этих сезонов.

String[] seasons = **new** String[4]; /\* объявили и создали массив. Java выделила память под массив из 4 строк, и сейчас в каждой ячейке записано значение null (поскольку строка — ссылочный тип)\*/

seasons[0] = "Winter"; /\* в первую ячейку, то есть, в ячейку с нулевым номером мы записали строку Winter. Тут мы получаем доступ к нулевому элементу массива и записываем туда конкретное значение \*/

seasons[1] = "Spring"; // проделываем ту же процедуру с ячейкой номер 1 (второй)

seasons[2] = "Summer"; // ...номер 2

seasons[3] = "Autumn"; // и с последней, номер 3

Теперь во всех четырёх ячейках нашего массива записаны названия сезонов. Инициализацию также можно провести по-другому, совместив с инициализацией и объявлением:

String[] seasons = **new** String[] {"Winter", "Spring", "Summer", "Autumn"};

Более того, оператор new можно опустить:

String[] seasons = {"Winter", "Spring", "Summer", "Autumn"};

## Как вывести массив в Java на экран?

Вывести элементы массива на экран (то есть, в консоль) можно, например, с помощью цикла for. Ещё один, более короткий способ вывода массива на экран будет рассмотрен в пункте “Полезные методы для работы с массивами", ниже. А пока рассмотрим пример с циклическим выводом массива:

String[] seasons = **new** String[] {"Winter", "Spring", "Summer", "Autumn"};

**for** (**int** i = 0; i < 4; i++) {

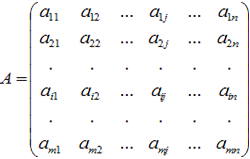
System.out.println(seasons[i]);

}

В результате программа выведет следующий результат:

Winter Spring Summer Autumn

## Одномерные и многомерные Java массивы

А что, если мы захотим создать не массив чисел, массив строк или массив каких-то объектов, а массив массивов? Java позволяет это сделать. Уже привычный нам массив **int[] myArray = new int[8]** — так называемый одномерный массив. А массив массивов называется двумерным. Он похож на таблицу, у которой есть номер строки и номер столбца. Или, если вы учили начала линейной алгебры, — на матрицу. Для чего нужны такие массивы? В частности, для программирования тех же матриц и таблиц, а также объектов, напоминающих их по структуре. Например, игровое поле для шахмат можно задать массивом 8х8. Многомерный массив объявляется и создается следующим образом:

Int[][] myTwoDimentionalArray = **new** **int** [8][8];

В этом массиве ровно 64 элемента: myTwoDimentionalArray[0][0], myTwoDimentionalArray[0][1], myTwoDimentionalArray[1][0], myTwoDimentionalArray[1][1] и так далее вплоть до myTwoDimentionalArray[7][7]. Так что если мы с его помощью представим шахматную доску, то клетку А1 будет представлять myTwoDimentionalArray[0][0], а E2 — myTwoDimentionalArray[4][1]. Где два, там и три. В Java можно задать массив массивов… массив массивов массивов и так далее. Правда, трёхмерные и более массивы используются очень редко. Тем не менее, с помощью трёхмерного массива можно запрограммировать, например, кубик Рубика.

## Полезные методы для работы с массивами

Для работы с массивами в Java есть класс **java.util.Arrays** (arrays на английском и означает “массивы”). В целом с массивами чаще всего проделывают следующие операции: заполнение элементами (инициализация), извлечение элемента (по номеру), сортировка и поиск. Поиск и сортировка массивов — тема отдельная. С одной стороны очень полезно потренироваться и написать несколько алгоритмов поиска и сортировки самостоятельно. С другой стороны, все лучшие способы уже написаны и включены в библиотеки Java, и ими можно законно пользоваться.

Вот три полезных метода этого класса

### Сортировка массива

Метод void sort(int[] myArray, int fromIndex, int toIndex) сортирует массив целых чисел или его подмассив по возрастанию.

### Поиск в массиве нужного элемента

int binarySearch(int[] myArray, int fromIndex, int toIndex, int key). Этот метод ищет элемент key в **уже отсортированном массиве** myArray или подмассиве, начиная с fromIndex и до toIndex. Если элемент найден, метод возвращает его индекс, если нет - (-fromIndex)-1.

### Преобразование массива к строке

Метод String toString(int[] myArray) преобразовывает массив к строке. Дело в том, что в Java массивы не переопределяют toString(). Это значит, что если вы попытаетесь вывести целый массив (а не по элементам, как в пункте “[Вывод массива на экран](https://javarush.com/groups/posts/massivy-java#%D0%9A%D0%B0%D0%BA-%D0%B2%D1%8B%D0%B2%D0%B5%D1%81%D1%82%D0%B8-%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2-%D0%B2-Java-%D0%BD%D0%B0-%D1%8D%D0%BA%D1%80%D0%B0%D0%BD)”) на экран непосредственно (System.out.println(myArray)), вы получите имя класса и шестнадцатеричный хэш-код массива (это определено Object.toString()). Если вы — новичок, вам, возможно, непонятно пояснение к методу toString. На первом этапе это и не нужно, зато с помощью этого метода упрощается вывод массива. Java позволяет легко выводить массив на экран без использования цикла. Об этом — в примере ниже.

### Пример на sort, binarySearch и toString

Давайте создадим массив целых чисел, выведем его на экран с помощью toString, отсортируем с помощью метода sort и найдём в нём какое-то число.

**class** Main {

**public** **static** **void** main(String[] args) {

**int**[] array = {1, 5, 4, 3, 7}; //объявляем и инициализируем массив

System.out.println(array);//пытаемся вывести наш массив на экран без метода toString - получаем 16-ричное число

System.out.println(Arrays.toString(array));//печатаем массив "правильно"

Arrays.sort(array, 0, 4); //сортируем весь массив от нулевого до четвёртого члена

System.out.println(Arrays.toString(array));//выводим отсортированный массив на экран

**int** key = Arrays.binarySearch(array, 5); // ищем key - число 5 в отсортированном массиве.

//метод binarySearch выдаст индекс элемента остортированного массива, в котором "спрятано" искомое число

System.out.println(key);//распечатываем индекс искомого числа

System.out.println(Arrays.binarySearch(array, 0));//а теперь попробуем найти число, которого в массиве нет,

// и сразу же выведем результат на экран

}

}

Вывод программы:

[I@1540e19d [1, 5, 4, 3, 7] [1, 3, 4, 5, 7] 3 -1

В первой строке — попытка вывода на экран массива без toString, во второй — вывод массива посредством toString, в третьей выведен отсортированный массив, в четвёртой — индекс искомого числа 5 в отсортированном массиве (помните, что считаем с нуля, поэтому четвёртый элемент массива имеет индекс 3). В пятой строке видим значение **-1**. Такого индекса у массива не бывает. Вывод сигнализирует о том, что искомого элемента (в данном случае, 0) в массиве нет.

## Главное о массивах

* Главные характеристики массива: тип помещённых в него данных, имя и длина.  
  Последнее решается при инициализации (выделении памяти под массив), первые два параметра определяются при объявлении массива.
* Размер массива (количество ячеек) нужно определять в int
* Изменить длину массива после его создания нельзя.
* Доступ к элементу массива можно получить по его индексу.
* В массивах, как и везде в Java, элементы нумеруются с нуля.
* После процедуры создания массива он наполнен значениями по умолчанию.
* Массивы в языке Java устроены не так, как в C++. Они почти совпадают с указателями на динамические массивы.

# Дата и время

В принципе, записать текущую дату и время в Java вполне возможно в обычную строку.

**public** **class** Main {

**public** **static** **void** main(String[] args) {

String date = "11 июня 2018 года";

System.out.println(date);

}

}

Но у такого подхода много недостатков. Класс String создан для работы с текстом, и методы у него соответствующие. Если нам нужно будет как-то управлять датой (прибавить к ней 2 часа, например) — String тут не справится. Или, например, — вывести в консоль текущую дату и время на момент компиляции программы. Тут String тоже не поможет: пока ты напишешь код и запустишь его - время изменится и в консоль будет выведено неактуальное. Поэтому в Java его создателями были предусмотрены несколько классов для работы с датами и временем. Первый из них - это класс java.util.Date

## Класс Date Java

Мы указали для него полное название, поскольку в другом пакете в Java есть еще класс java.sql.Date. Не перепутай! Первое что нужно о нем знать — **он хранит дату в миллисекундах**, которые прошли с 1 января 1970 года. Для этой даты есть даже отдельное название — “Unix-время” Довольно интересный способ, согласен? :) Второе, что стоит запомнить: если создать объекта Date с пустым конструктором — результатом будет **текущая дата и время на момент создания объекта**. Помнишь, мы писали, что для даты в формате String такая задача будет проблематичной? Класс Date ее легко решает.

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Date date = **new** Date();

System.out.println(date);

}

}

Запусти этот код несколько раз, и увидишь, как время каждый раз будет меняться:) Это возможно именно благодаря хранению в миллисекундах: они являются самой маленькой единицей времени, поэтому результаты настолько точные. Существует и другой конструктор для Date: можно указать точное количество миллисекунд, которое прошло с 00:00 1 января 1970 года до требуемой даты, и она будет создана:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Date date = **new** Date(1212121212121L);

System.out.println(date);

}

}

Вывод в консоль:

Fri May 30 08:20:12 MSD 2008

У нас получилось 30 мая 2008 года. “Fri” означает день недели — “Friday” (пятница), а MSD — “Moscow Daylight Saving” (московское летнее время). Миллисекунды передаются в формате long, поскольку их количество чаще всего не влезает в int. Итак, какие операции с датами нам могут понадобиться в работе? Ну, самое очевидное, конечно — **сравнение**. Определить была ли одна дата позже или раньше другой. Это можно сделать по-разному. Например, можно вызвать метод **Date.getTime()**.Он вернет количество миллисекунд, прошедших с полуночи 1 января 1970 года. Просто вызовем его у двух объектов Date и сравним между собой:

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Date date1 = **new** Date();

Date date2 = **new** Date();

System.out.println((date1.getTime() > date2.getTime())?

"date1 позже date2" : "date1 раньше date2");

}

}

Вывод:

date1 раньше date2

Но есть и более удобный способ, а именно — использовать специальные методы класса Date: before(), after() и equals(). Все они возвращают результат в формате boolean. Метод before() проверяет, была ли наша дата раньше той, которую мы передаем в качестве аргумента:

**public** **class** Main {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Date date1 = **new** Date();

Thread.sleep(2000);//приостановим работу программы на 2 секунды

Date date2 = **new** Date();

System.out.println(date1.before(date2));

}

}

Вывод в консоль:

true

Похожим образом работает и метод after(), он проверяет была ли ли наша дата позже той, которую мы передаем в качестве аргумента:

**public** **class** Main {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

Date date1 = **new** Date();

Thread.sleep(2000);//приостановим работу программы на 2 секунды

Date date2 = **new** Date();

System.out.println(date1.after(date2));

}

}

Вывод в консоль:

false

В наших примерах мы “усыпляем” программу на 2 секунды, чтобы две даты гарантированно отличались. На быстрых компьютерах время между созданием date1 и date2 может быть меньше одной миллисекунды, и в таком случае оба метода — и before(), и after() — будут возвращать false. А вот метод equals() в такой ситуации вернет true! Ведь он сравнивает именно количество миллисекунд, прошедших с 00:00 1 января 1970 для каждой даты. Объекты будут считаться равными только в том случае, если совпадают вплоть до миллисекунды:

**public** **static** **void** main(String[] args) {

Date date1 = **new** Date();

Date date2 = **new** Date();

System.out.println(date1.getTime());

System.out.println(date2.getTime());

System.out.println(date1.equals(date2));

}

Вот еще на что нужно обратить внимание. Если ты откроешь документацию класса Date на сайте Oracle, то увидишь, что многие его методы и конструкторы были обозначены словом Deprecated (“нерекомендуемый‘). Вот, посмотри: [Class Date](https://docs.oracle.com/javase/7/docs/api/java/util/Date.html#before(java.util.Date)) Вот что сами создатели Java говорят про те части классов, которые стали deprecated: ***“Программный элемент, аннотированный @Deprecated, является тем, что программистам не рекомендуется использовать, как правило, потому, что это опасно, или потому, что существует лучшая альтернатива.”*** Это не означает, что этими методами вообще нельзя пользоваться. Более того, если ты сам попробуешь запустить код с их использованием в IDEA — он, скорее всего будет работать Возьмем для примера deprecated метод Date.getHours(), который возвращает количество часов из объекта Date.

**public** **static** **void** main(String[] args) {

Date date1 = **new** Date();

System.out.println(date1.~~getHours()~~);

}

Если на момент запуска кода у вас, например, время 14:21, он выведет число 14. Как видите, deprecated-метод зачеркнут, но он вполне себе работает. Это методы не стали убирать совсем, чтобы не сломать кучу уже написанного с их использованием кода. То есть эти методы не “сломаны” и не “удалены”, просто их не рекомендуют использовать по причине наличия более удобной альтернативы. О ней, кстати, написано прямо в документации: Большинство методов класса Date было перенесено в его улучшенную, расширенную версию — класс Calendar. С ним мы и познакомимся дальше:)

## Java Calendar

В версии Java 1.1 появился новый класс — Calendar. Он сделал работу с датам в Java несколько проще, чем она выглядела раньше. Единственной реализацией класса Calendar, с которой мы и будем работать, является класс GregorianCalendar (он реализует Григорианский календарь, по которому живет большинство стран мира). Его основное удобство заключается в том, что он умеет работать с датами в более удобном формате. Например, он может:

* Прибавить к текущей дате месяц или день
* Проверить, является ли год високосным;
* Получить отдельные компоненты даты (например, получить из целой даты номер месяца)
* А также — внутри него разработана очень удобная система констант (многие из них мы увидим ниже).

Еще одним важным отличием класса Calendar является то, что в нем реализована константа Calendar.Era: ты можешь установить для даты эру **BC** (“Before Christ” - до рождества Христова, т.е. “до нашей эры”) или **AC** (“After Christ” - “наша эра”). Давай рассмотрим все это на примерах. Создадим календарь с датой 25 января 2017 года:

**public** **static** **void** main(String[] args) {

Calendar calendar = **new** GregorianCalendar(2017, 0 , 25);

}

Месяцы в классе Calendar (как и в Date, кстати) начинаются с нуля, поэтому мы передали число 0 в качестве второго аргумента. Главное при работе с классом Calendar — понимать, что это именно **календарь**, а не отдельная дата. **Дата** — это просто несколько чисел, обозначающих конкретный промежуток времени. А календарь - это целое устройство, с помощью которого можно много чего с датами делать:) Это достаточно хорошо видно, если попробовать вывести объект Calendar в консоль: Вывод:

java.util.GregorianCalendar[time=?,areFieldsSet=false,areAllFieldsSet=false,lenient=true,zone=sun.util.calendar.ZoneInfo[id="Europe/Moscow",offset=10800000,dstSavings=0,useDaylight=false,transitions=79,lastRule=null],firstDayOfWeek=2,minimalDaysInFirstWeek=1,ERA=?,YEAR=2017,MONTH=0,WEEK\_OF\_YEAR=?,WEEK\_OF\_MONTH=?,DAY\_OF\_MONTH=25,DAY\_OF\_YEAR=?,DAY\_OF\_WEEK=?,DAY\_OF\_WEEK\_IN\_MONTH=?,AM\_PM=0,HOUR=0,HOUR\_OF\_DAY=0,MINUTE=0,SECOND=0,MILLISECOND=?,ZONE\_OFFSET=?,DST\_OFFSET=?]

Видишь сколько информации! У календаря есть куча свойств, которыми не обладает обычная дата, и все они выводятся в консоль (так работает метод toString() в классе Calendar). Если при работе тебе нужно просто получить из календаря простую дату, т.е. объект Date — это делается при помощи метода Calendar.getTime() (название не самое логичное, но тут уж ничего не поделаешь):

**public** **static** **void** main(String[] args) {

Calendar calendar = **new** GregorianCalendar(2017, 0 , 25);

Date date = calendar.getTime();

System.out.println(date);

}

Вывод:

Wed Jan 25 00:00:00 MSK 2017

Вот теперь мы “упростили” календарь до обычной даты. Поехали дальше. Помимо цифровых обозначений месяцев, в классе Calendar можно использовать константы. Константы — это статические поля класса Calendar с уже установленным значением, которое нельзя изменить. Этот вариант на самом деле даже лучше, поскольку такое написание улучшает читаемость кода.

**public** **static** **void** main(String[] args) {

GregorianCalendar calendar = **new** GregorianCalendar(2017, Calendar.JANUARY , 25);

}

Calendar.JANUARY — одна из констант для обозначения месяца. При таком варианте именования никто не забудет, например, что цифра “3” обозначает апрель, а не привычный нам третий по счету месяц - март. Просто пишешь Calendar.APRIL - и все:) Все поля календаря (число, месяц, минуты, секунды и т.д.) можно устанавливать по отдельности с помощью метода set(). Он очень удобен, поскольку в классе Calendar для каждого поля выделена своя константа, и итоговый код будет выглядеть максимально просто. Например, в прошлом примере мы создали дату, но не установили для нее текущее время. Давай установим время 19:42:12

**public** **static** **void** main(String[] args) {

Calendar calendar = **new** GregorianCalendar();

calendar.set(Calendar.YEAR, 2017);

calendar.set(Calendar.MONTH, 0);

calendar.set(Calendar.DAY\_OF\_MONTH, 25);

calendar.set(Calendar.HOUR\_OF\_DAY, 19);

calendar.set(Calendar.MINUTE, 42);

calendar.set(Calendar.SECOND, 12);

System.out.println(calendar.getTime());

}

Вывод:

Wed Jan 25 19:42:12 MSK 2017

Мы вызываем метод set(), передаем в него константу (в зависимости от того поля, которое хотим изменить) и новое значение для этого поля. Получается, что метод set() — эдакий “супер-сеттер”, который умеет устанавливать значение не для одного поля, а для множества полей:) Прибавление и вычитание значений в классе Calendar осуществляется с помощью метода add(). В него необходимо передать то поле, которое ты хочешь изменить, и число - сколько именно ты хочешь прибавить/убавить от текущего значения. Например, вернем дату, которую мы создали, на 2 месяца назад:

**public** **static** **void** main(String[] args) {

Calendar calendar = **new** GregorianCalendar(2017, Calendar.JANUARY , 25);

calendar.set(Calendar.HOUR, 19);

calendar.set(Calendar.MINUTE, 42);

calendar.set(Calendar.SECOND, 12);

calendar.add(Calendar.MONTH, -2);//чтобы отнять значение - в метод нужно передать отрицательное число

System.out.println(calendar.getTime());

}

Вывод:

Fri Nov 25 19:42:12 MSK 2016

Отлично! Мы вернули дату на 2 месяца назад. В результате изменился не только месяц, но и год, с 2017 на 2016. Подсчет текущего года при переносе дат, конечно, выполняется автоматически и его не надо контролировать вручную. Но если для каких-то целей тебе нужно отключить это поведение — можно и так. Специальный метод roll() может прибавлять и убавлять значения, не затрагивая при этом остальные значения. К примеру, вот так:

**public** **static** **void** main(String[] args) {

Calendar calendar = **new** GregorianCalendar(2017, Calendar.JANUARY , 25);

calendar.set(Calendar.HOUR, 10);

calendar.set(Calendar.MINUTE, 42);

calendar.set(Calendar.SECOND, 12);

calendar.roll(Calendar.MONTH, -2);

System.out.println(calendar.getTime());

}

Мы сделали ровно то же самое, что и в предыдущем примере — отняли 2 месяца от текущей даты. Но теперь код сработал по-другому: месяц поменялся с января на ноябрь, но год как был 2017-ым, так и остался! Вывод:

Sat Nov 25 10:42:12 MSK 2017

Далее. Как мы и говорили выше, все поля объекта Calendar можно получить по отдельности. За это отвечает метод get():

**public** **static** **void** main(String[] args) {

GregorianCalendar calendar = **new** GregorianCalendar(2017, Calendar.JANUARY , 25);

calendar.set(Calendar.HOUR, 10);

calendar.set(Calendar.MINUTE, 42);

calendar.set(Calendar.SECOND, 12);

System.out.println("Год: " + calendar.get(Calendar.YEAR));

System.out.println("Месяц: " + calendar.get(Calendar.MONTH));

System.out.println("Порядковый номер недели в месяце: " + calendar.get(Calendar.WEEK\_OF\_MONTH));//порядковый номер недели в месяце

System.out.println("Число: " + calendar.get(Calendar.DAY\_OF\_MONTH));

System.out.println("Часы: " + calendar.get(Calendar.HOUR));

System.out.println("Минуты: " + calendar.get(Calendar.MINUTE));

System.out.println("Секунды: " + calendar.get(Calendar.SECOND));

System.out.println("Миллисекунды: " + calendar.get(Calendar.MILLISECOND));

}

Вывод:

Год: 2017

Месяц: 0

Порядковый номер недели в месяце: 4

Число: 25

Часы: 10

Минуты: 42

Секунды: 12

Миллисекунды: 0

То есть помимо “супер-сеттера” в классе Calendar есть еще и “супер-геттер” :) Еще один интересный момент — это, конечно, работа с эрами. Для создания даты “до нашей эры” нужно использовать поле Calendar.Era Например, создадим дату, обозначающую битву при Каннах, в которой Ганнибал победил войско Рима. Это произошло 2 августа 216 г. до н. э.:

**public** **static** **void** main(String[] args) {

GregorianCalendar cannes = **new** GregorianCalendar(216, Calendar.AUGUST, 2);

cannes.set(Calendar.ERA, GregorianCalendar.BC);

DateFormat df = **new** SimpleDateFormat("dd MMM yyy GG");

System.out.println(df.format(cannes.getTime()));

}

Здесь мы использовали класс SimpleDateFormat, чтобы вывести дату в более понятном нам формате (буквы “GG” отвечают как раз за вывод эры). Вывод:

02 авг 216 до н.э.

В классе Calendar есть еще много методов и констант, почитай про них в документации:

* [Class Calendar](https://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html)

## Перевод строки в Date

Для перевода String в Date можно воспользоваться вспомогательным классом Java — [SimpleDateFormat](https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html). Это класс, который нужен для приведения даты в определяемый вами формат. В свою очередь, он очень похож на [DateFormat](https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html). Единственное заметное различие между ними заключается в том, что SimpleDateFormat можно использовать для форматирования (преобразования даты в строку) и для парсинга строки в дату с поддержкой языкового стандарта, тогда как DateFormat не поддерживает языкового стандарта. Кроме того, DateFormat — это абстрактный класс, который обеспечивает базовую поддержку для форматирования и анализа дат, а SimpleDateFormat — это конкретный класс, расширяющий класс DateFormat. Вот так выглядит пример создания объекта SimpleDateFormat и форматирования Date:

SimpleDateFormat formater = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Date date = **new** Date(1212121212121L);

System.out.println(formatter.format(date));

В приведенном выше примере мы использовали шаблон "yyyy-MM-dd HH:mm:ss", который означает:

* 4 цифры для года (yyyy);
* 2 цифры для месяца (ММ);
* 2 цифры для дня (dd);
* 2 цифры для часов в 24-часовом формате (HH);
* 2 цифры для минут (mm);
* 2 цифры для секунд (ss).

Знаки разделения и порядок расстановки символов шаблона сохраняется. Вывод в консоль:

2008-05-30 08:20:12

Шаблонных букв для класса **SimpleDateFormat** довольно много. Чтобы ты не запутался, мы собрали их в таблицу:

|  |  |  |
| --- | --- | --- |
| **Символ** | **Описание** | **Пример** |
| G | эра (в английской локализации — AD и BC) | н.э. |
| y | год (4-х значное число) | 2020 |
| yy | год (последние 2 цифры) | 20 |
| yyyy | год (4-х значное число) | 2020 |
| M | номер месяца (без лидирующих нулей) | 8 |
| MM | номер месяца (с лидирующими нулями, если порядковый номер месяца < 10) | 04 |
| MMM | трехбуквенное сокращение месяца (в соответствии с локализацией) | янв |
| MMMM | полное название месяца | Июнь |
| w | неделя в году (без лидирующих нулей) | 4 |
| ww | неделя в году (с лидирующими нулями) | 04 |
| W | неделя в месяце (без лидирующих нулей) | 3 |
| WW | неделя в месяце (с лидирующим нулем) | 03 |
| D | день в году | 67 |
| d | день месяца (без лидирующих нулей) | 9 |
| dd | день месяца (с лидирующими нулями) | 09 |
| F | день недели в месяце (без лидирующих нулей) | 9 |
| FF | день недели в месяце (с лидирующими нулями) | 09 |
| E | день недели (сокращение) | Вт |
| EEEE | день недели (полностью) | пятница |
| u | номер дня недели (без лидирующих нулей) | 5 |
| uu | номер дня недели (с лидирующими нулями) | 05 |
| a | маркер AM/PM | AM |
| H | часы в 24-часовом формате без лидирующих нулей | 6 |
| HH | часы в 24-часовом формате с лидирующим нулем | 06 |
| k | количество часов в 24-часовом формате | 18 |
| K | количество часов в 12-часовом формате | 6 |
| h | время в 12-часовом формате без лидирующих нулей | 6 |
| hh | время в 12-часовом формате с лидирующим нулем | 06 |
| m | минуты без лидирующих нулей | 32 |
| mm | минуты с лидирующим нулем | 32 |
| s | секунды без лидирующих нулей | 11 |
| ss | секунды с лидирующим нулем | 11 |
| S | миллисекунды | 297 |
| z | часовой пояс | EET |
| Z | часовой пояс в формате RFC 822 | 300 |

Примеры комбинаций символов шаблонов:

|  |  |
| --- | --- |
| **Шаблон** | **Пример** |
| dd-MM-yyyy | 01-11-2020 |
| yyyy-MM-dd | 2019-10-01 |
| HH:mm:ss.SSS | 23:59.59.999 |
| yyyy-MM-dd HH:mm:ss | 2018-11-30 03:09:02 |
| yyyy-MM-dd HH:mm:ss.SSS | 2016-03-01 01:20:47.999 |
| yyyy-MM-dd HH:mm:ss.SSS Z | 2013-13-13 23:59:59.999 +0100 |
|  |  |

Если же немного ошибиться с форматом, то можно стать обладателем java.text.ParseException, а это не особо приятное достижение. Ну что же, небольшой экскурс по **SimpleDateFormat** окончен — вернёмся к переводу **java string to date**. **SimpleDateFormat** дает нам такие возможности, и мы рассмотрим этот процесс поэтапно.

1. Создаем строку, с которой нужно задать дату:

String strDate = "Sat, April 4, 2020";

1. Создаем новый объект **SimpleDateFormat** с шаблоном, который совпадает с тем, что у нас в строке (иначе распарсить не получится):

SimpleDateFormat formatter = **new** SimpleDateFormat("EEE, MMMM d, yyyy", Locale.ENGLISH);

Как вы видите, у нас тут появился аргумент Locale. Если же мы его опустим, он будет использовать значение Locale по умолчанию, которое не всегда является английским.

Если языковой стандарт не совпадает с входной строкой, то строковые данные, привязанные к языку, как у нас Mon или April, не будут распознаны и вызовут падение — java.text.ParseException, даже в том случае когда шаблон подходит.

Тем не менее, можно не указывать формат, если у нас используется шаблон, который не привязан к языку. Как пример — yyyy-MM-dd HH:mm:ss

1. Создаём дату с помощью форматтера, который в свою очередь парсит её из входной строки:
2. **try** {
3. Date date = formatter.parse(strDate);
4. System.out.println(date);
5. }
6. **catch** (ParseException e) {
7. e.printStackTrace();

}

Вывод в консоль:

Sat Apr 04 00:00:00 EEST 2020

Хммм….Но формат-то уже не тот!

Чтобы сделать тот же формат, вновь используем форматтер:

System.out.println(formatter.format(date));

Вывод в консоль:

Sat, April 4, 2020

## SimpleDateFormat и Calendar

SimpleDateFormat позволит тебе форматировать все создаваемые объекты Date и Calendar для последующего использования. Рассмотрим такой интересный момент как работа с эрами. Для создания даты “до нашей эры” нужно использовать поле Calendar.Era Например, создадим дату, обозначающую битву при Каннах, в которой Ганнибал победил войско Рима. Это произошло 2 августа 216 г. до н. э.:

**public** **static** **void** main(String[] args) {

GregorianCalendar cannes = **new** GregorianCalendar(216, Calendar.AUGUST, 2);

cannes.set(Calendar.ERA, GregorianCalendar.BC);

DateFormat df = **new** SimpleDateFormat("dd MMM yyy GG");

System.out.println(df.format(cannes.getTime()));

}

Здесь мы использовали класс SimpleDateFormat, чтобы вывести дату в более понятном нам формате (как указано выше, буквы “GG” отвечают как раз за вывод эры). Вывод:

02 авг 216 до н.э.

## Java Date Format

А вот еще один случай. Предположим, что данный формат даты нас не устраивает:

Sat Nov 25 10:42:12 MSK 2017

Так вот. С помощью наших возможностей в java date format его можно поменять его на свой собственный, без особых сложностей:

**public** **static** **void** main(String[] args) {

SimpleDateFormat dateFormat = **new** SimpleDateFormat("EEEE, d MMMM yyyy");

Calendar calendar = **new** GregorianCalendar(2017, Calendar.JANUARY , 25);

calendar.set(Calendar.HOUR, 10);

calendar.set(Calendar.MINUTE, 42);

calendar.set(Calendar.SECOND, 12);

calendar.roll(Calendar.MONTH, -2);

System.out.println(dateFormat.format(calendar.getTime()));

}

Вывод:

суббота, 25 Ноябрь 2017

# Регулярные выражения

**Регулярные выражения** — тема, которую программисты, даже опытные, зачастую откладывают на потом. Однако большинству Java-разработчиков рано или поздно придётся столкнуться с обработкой текстовой информации. Чаще всего — с операциями поиска в тексте и редактированием. Без регулярных выражений продуктивный и компактный программный код, связанный с обработкой текстов, попросту немыслим. Так что хватит откладывать, разберёмся с «регулярками» прямо сейчас. Это не такая уж и сложная задача.

## Что такое регулярное выражение RegEx?

На самом деле регулярное выражение (RegEx) – это шаблон для поиска строки в тексте. В Java исходным представлением этого шаблона всегда является строка, то есть объект класса String. Однако не любая строка может быть скомпилирована в регулярное выражение, а только та, которая соответствует правилам написания регулярного выражения – синтаксису, определенному в спецификации языка. Для написания регулярного выражения используются буквенные и цифровые символы, а также метасимволы – символы, имеющие специальное значение в синтаксисе регулярных выражений. Например:

String regex=”java”; // шаблон строки ”java”;

String regex=”\\d{3}”; // шаблон строки из трех цифровых символов;

## Создание регулярных выражений в Java

Чтобы создать RegEx в Java, нужно сделать два простых шага:

1. написать его в виде строки с учётом синтаксиса регулярных выражений;
2. скомпилировать эту строку в регулярное выражение;

Работа с регулярными выражениями в любой Java-программе начинается с создания объекта класса Pattern. Для этого необходимо вызвать один из двух имеющихся в классе статических методов compile. Первый метод принимает один аргумент – строковый литерал регулярного выражения, а второй – плюс еще параметр, включающий режим сравнения шаблона с текстом:

**public** **static** Pattern compile (String literal)

**public** **static** Pattern compile (String literal, **int** flags)

[Список](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#CASE_INSENSITIVE) возможных значений параметра flags определен в классе Pattern и доступен нам как статические переменные класса. Например:

Pattern pattern = Pattern.compile("java", Pattern.CASE\_INSENSITIVE);//поиск совпадений с шаблоном будет производиться без учета регистра символов.

По сути, класс Pattern — это конструктор регулярных выражений. Под «капотом» метод compile вызывает закрытый конструктор класса Pattern для создания скомпилированного представления. Такой способ создания экземпляра шаблона реализован с целью создания его в виде неизменяемого объекта. При создании производится синтаксическая проверка регулярного выражения. При наличии ошибок в строке – генерируется исключение PatternSyntaxException.

## Синтаксис регулярных выражений

Синтаксис регулярных выражений основан на использовании символов <([{\^-=$!|]})?\*+.>, которые можно комбинировать с буквенными символами. В зависимости от роли их можно разделить на несколько групп:

1. Метасимволы для поиска совпадений границ строк или текста

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| ^ | начало строки |
| $ | конец строки |
| \b | граница слова |
| \B | не граница слова |
| \A | начало ввода |
| \G | конец предыдущего совпадения |
| \Z | конец ввода |
| \z | конец ввода |

2. Метасимволы для поиска символьных классов

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| \d | цифровой символ |
| \D | нецифровой символ |
| \s | символ пробела |
| \S | непробельный символ |
| \w | буквенно-цифровой символ или знак подчёркивания |
| \W | любой символ, кроме буквенного, цифрового или знака подчёркивания |
| . | любой символ |

3. Метасимволы для поиска символов редактирования текста

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| \t | символ табуляции |
| \n | символ новой строки |
| \r | символ возврата каретки |
| \f | переход на новую страницу |
| \u 0085 | символ следующей строки |
| \u 2028 | символ разделения строк |
| \u 2029 | символ разделения абзацев |

4. Метасимволы для группировки символов

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| [абв] | любой из перечисленных (а,б, или в) |
| [^абв] | любой, кроме перечисленных (не а,б, в) |
| [a-zA-Z] | слияние диапазонов (латинские символы от a до z без учета регистра ) |
| [a-d[m-p]] | объединение символов (от a до d и от m до p) |
| [a-z&&[def]] | пересечение символов (символы d,e,f) |
| [a-z&&[^bc]] | вычитание символов (символы a, d-z) |

5. Метасимволы для обозначения количества символов – квантификаторы. Квантификатор всегда следует после символа или группы символов.

|  |  |
| --- | --- |
| **Метасимвол** | **Назначение** |
| ? | один или отсутствует |
| \* | ноль или более раз |
| + | один или более раз |
| {n} | n раз |
| {n,} | n раз и более |
| {n,m} | не менее n раз и не более m раз |

## Жадный режим квантификатора

Особенностью квантификаторов является возможность использования их в разных режимах: жадном, сверхжадном и ленивом. **Сверхжадный режим** включается добавлением символа «+» после квантификатора, а ленивый – символа «?». Например:

"А.+а" // жадный режим

"А.++а" // сверхжадный режим

"А.+?а" // ленивый режим

Попробуем на примере этого шаблона разобраться в работе квантификаторов в различных режимах. По умолчанию квантификатор работает в жадном режиме. Это означает, что он ищет максимально длинное совпадение в строке. В результате выполнения этого кода:

**public** **static** **void** main(String[] args) {

String text = "Егор Алла Александр";

Pattern pattern = Pattern.compile("А.+а");

Matcher matcher = pattern.matcher(text);

**while** (matcher.find()) {

System.out.println(text.substring(matcher.start(), matcher.end()));

}

}

**мы получим такой вывод: Алла Алекса** Алгоритм поиска по заданному шаблону "А.+а", выполняется в следующей последовательности:

1. В заданном шаблоне первый символ – это русский символ буквы А. Matcher сопоставляет его с каждым символом текста, начиная с нулевой позиции. На нулевой позиции в нашем тексте находиться символ Е, поэтому Matcher перебирает последовательно символы в тексте, пока не встретит совпадение с шаблоном. В нашем примере это символ на позиции №5.



1. После того, как найдено совпадение с первым символом шаблона, Matcher сверяет соответствие со вторым символом шаблона. В нашем случае это символ «.», который обозначает любой символ.



На шестой позиции – символ буквы л. Разумеется, он соответствует шаблону «любой символ».

1. Matcher переходит к проверке следующего символа из шаблона. В нашем шаблоне он задан с помощью квантификатора «.+». Поскольку количество повторений «любого символа» в шаблоне – один и более раз, Matcher берет по очереди следующий символ из строки и проверяет его на соответствие шаблону, до тех пор, пока будет выполняться условие «любой символ», в нашем примере – до конца строки (с поз. №7 -№18 текста).



По сути, Matcher, захватывает все строку до конца – в этом как раз и проявляется его «жадность».

1. После того как Matcher дошел до конца текста и закончил проверку для части шаблона «А.+», Matcher начинает проверку для оставшейся части шаблона – символ буквы а. Так как текст в прямом направлении закончился, проверка происходит в обратном направлении, начиная с последнего символа:



1. Matcher «помнит» количество повторений в шаблоне «.+» при котором он дошел до конца текста, поэтому он уменьшает количество повторений на единицу и проверяет соответствие шаблона тексту, до тех пор пока не будет найдено совпадение:



## Сверхжадный режим квантификатора

В сверхжадном режиме работа матчера аналогична механизму жадного режима. Отличие состоит в том, что при захватывании текста до конца строки поиск в обратном направлении не происходит. То есть первые три этапа при сверхжадном режиме будут аналогичны жадному режиму. После захвата всей строки матчер добавляет остаток шаблона и сравнивает с захваченной строкой. В нашем примере при выполнении метода main с шаблоном "А.++а" совпадений не будет найдено.



## Ленивый режим квантификатора

1. В этом режиме на начальном этапе, как и в жадном режиме, ищется совпадение с первым символом шаблона:



1. Далее ищется совпадение со следующим символом шаблона – любым символом:



1. В отличие от жадного режима, в ленивом ищется самое короткое совпадение в тексте, поэтому после нахождения совпадения со вторым символом шаблона, который задан точкой и соответствует символу на позиции №6 текста, Matcher будет проверять соответствие текста остатку шаблона – символу «а»



1. Поскольку совпадение с шаблоном в тексте не найдено (на позиции №7 в тексте находится символ «л»), Matcher добавляет еще один «любой символ» в шаблоне, так как он задан как один и более раз, и опять сравнивает шаблон с текстом на позициях с №5 по №8:



1. В нашем случае найдено совпадение, но конец текста ещё не достигнут. Поэтому с позиции №9 проверка начинается с поиска первого символа шаблона по аналогичному алгоритму и далее повторяется вплоть до окончания текста.



В результате работы метода main при использовании шаблона "А.+?а" мы получим следующий результат: Алла Алекса Как видно из нашего примера, при использовании разных режимов квантификатора для одного и того же шаблона мы получили разные результаты. Поэтому необходимо учитывать эту особенность и выбирать нужный режим в зависимости от желаемого результата при поиске.

## Экранирование символов в регулярных выражениях

Поскольку регулярное выражение в Java, а точнее — его исходное представление задается с помощью строкового литерала, необходимо учитывать те правила спецификации Java, которые касаются строковых литералов. В частности, символ обратной косой черты «\» в строковых литералах в исходном коде Java интерпретируется как символ управляющей последовательности, который предупреждает компилятор, что следующий за ним символ — специальный и что его нужно особым образом интерпретировать. Например:

String s=“The root directory is \nWindows”;//перенос Windows на новую строку

String s=“The root directory is \u00A7Windows”;//вставка символа параграфа перед Windows

Поэтому в строковых литералах, которые описывают регулярное выражение, и используют символ «\» (например, для метасимволов) **его нужно удваивать**, чтобы компилятор байт-кода Java не интерпретировал его по-своему. Например:

String regex=”\\s”; // шаблон для поиска символов пробела

String regex=”\\”Windows\\””; // шаблон для поиска строки ”Windows”

Двойной символ обратной косой черты также следует использовать для экранирования символов, задействованных в качестве специальных, если мы планируем их использовать как «обычные» символы. Например:

String regex=”How\\?”; // шаблон для поиска строки “How?”

## Методы класса Pattern

В классе Pattern есть и другие методы для работы с регулярными выражениями: **String pattern()** – возвращает исходное строковое представление регулярного выражения, из которого был создан объект Pattern:

Pattern pattern = Pattern.compile("abc");

System.out.println(Pattern.pattern())//"abc"

**static boolean matches(String regex, CharSequence input)** – позволяет проверить регулярное выражение, переданное в параметре regex на соответствие тексту, переданному в параметре input. Возвращает: true – если текст соответствует шаблону; false – в противном случае; Пример:

System.out.println(Pattern.matches("А.+а","Алла"));//true

System.out.println(Pattern.matches("А.+а","Егор Алла Александр"));//false

**int flags()** – возвращает значения параметра flags шаблона, которые были установлены при его создании, или 0, если этот параметр не был установлен. Пример:

Pattern pattern = Pattern.compile("abc");

System.out.println(pattern.flags());// 0

Pattern pattern = Pattern.compile("abc",Pattern.CASE\_INSENSITIVE);

System.out.println(pattern.flags());// 2

**String[] split(CharSequence text, int limit)** – разбивает текст, переданный в качестве параметра на массив элементов String. Параметр limit определяет предельное количество совпадений, которое ищется в тексте:

* при limit>0 – выполняется поиск limit-1 совпадений;
* при limit<0 – выполняется поиск всех совпадений в тексте
* при limit=0 – выполняется поиск всех совпадений в тексте, при этом пустые строки в конце массива отбрасываются;

Пример:

**public** **static** **void** main(String[] args) {

String text = "Егор Алла Анна";

Pattern pattern = Pattern.compile("\\s");

String[] strings = pattern.split(text,2);

**for** (String s : strings) {

System.out.println(s);

}

System.out.println("---------");

String[] strings1 = pattern.split(text);

**for** (String s : strings1) {

System.out.println(s);

}

}

Вывод на консоль: Егор Алла Анна -------- Егор Алла Анна Еще один метод класса для создания объекта Matcher рассмотрим ниже.

## Методы класса Matcher

Matcher представляет собой класс, из которого создается объект для поиска совпадений по шаблону. Matcher – это «поисковик», «движок» регулярных выражений. Для поиска ему надо дать две вещи: шаблон поиска и «адрес», по которому искать. Для создания объекта Matcher предусмотрен следующий метод в классе Pattern: **рublic Matcher matcher(CharSequence input)** В качестве аргумента метод принимает последовательность символов, в котором будет производиться поиск. Это объекты классов, реализующих интерфейс CharSequence. в качестве аргумента можно передать не только String, но и StringBuffer, StringBuilder, Segment и CharBuffer. Шаблоном для поиска является объект класса Pattern, на котором вызывается метод matcher. Пример создания матчера:

Pattern p = Pattern.compile("a\*b");// скомпилировали регулярное выражение в представление

Matcher m = p.matcher("aaaaab");//создали поисковик в тексте “aaaaab” по шаблону "a\*b"

Теперь с помощью нашего «поисковика» мы можем искать совпадения, узнавать позицию совпадения в тексте, заменять текст с помощью методов класса. Метод **boolean find()** ищет очередное совпадение в тексте с шаблоном. С помощью этого метода и оператора цикла можно производить анализ всего текста по событийной модели (осуществлять необходимые операции при наступлении события – нахождении совпадения в тексте). Например, с помощью методов этого класса **int start()** и **int end()** можно определять позиции совпадения в тексте, а с помощью методов **String replaceFirst(String replacement)** и **String replaceAll(String replacement)** заменять в тексте совпадения на другой текст replacement. Пример:

**public** **static** **void** main(String[] args) {

String text = "Егор Алла Анна";

Pattern pattern = Pattern.compile("А.+?а");

Matcher matcher = pattern.matcher(text);

**while** (matcher.find()) {

**int** start=matcher.start();

**int** end=matcher.end();

System.out.println("Найдено совпадение " + text.substring(start,end) + " с "+ start + " по " + (end-1) + " позицию");

}

System.out.println(matcher.replaceFirst("Ира"));

System.out.println(matcher.replaceAll("Ольга"));

System.out.println(text);

}

Вывод программы: Найдено совпадение Алла с 5 по 8 позицию Найдено совпадение Анна с 10 по 13 позицию Егор Ира Анна Егор Ольга Ольга Егор Алла Анна Из примера видно, что методы replaceFirst и replaceAll создают новый объект String – строку, представляющую собой исходный текст, в котором совпадения с шаблоном заменены на текст, переданный методу в качестве аргумента. Причём метод replaceFirst заменяет только первое совпадение, а replaceAll – все совпадения в тесте. Исходный текст остается без изменений. Использование других методов класса Matcher, а также примеры регулярных выражений можно посмотреть в этом цикле [статей](https://javarush.com/groups/posts/186-reguljarnihe-vihrazhenija-v-java-chastjh-3). Наиболее частые операции с регулярными выражениями при работе с текстом из классов Pattern и Matcher встроены в класс String. Это такие методы как split, matches, replaceFirst, replaceAll. Но на самом деле «под капотом» они используют классы Pattern и Matcher. Поэтому, если вам нужно заменить текст или сравнить строки в программе без написания лишнего кода, используйте методы класса String. Если же вам нужны расширенные возможности – вспомните о классах Pattern и Matcher.

## Заключение

Регулярное выражение описывается в Java-программе с помощью строк, подходящих под определённый правилами шаблон. При выполнении кода Java перекомпилирует эту строку в объект класс Pattern и использует объект класса Matcher для поиска соответствий в тексте. Как я уже говорил в начале, регулярные выражения очень часто откладывают на потом, считая сложной темой. Однако если разобраться с основами синтаксиса, метасимволами, экранированием и изучить примеры регулярных выражений, они оказываются гораздо проще, чем кажутся на первый взгляд.

# Исключения в Java

## Что такое исключения / exception java?

В мире программирования возникновение ошибок и непредвиденных ситуаций при выполнении программы называют **исключением**. В программе исключения могут возникать в результате неправильных действий пользователя, отсутствии необходимого ресурса на диске, или потери соединения с сервером по сети. Причинами исключений при выполнении программы также могут быть ошибки программирования или неправильное использование API. В отличие от нашего мира, программа должна четко знать, как поступать в такой ситуации. Для этого в Java предусмотрен механизм исключений.

## Кратко о ключевых словах try, catch, finally, throws

**Обработка исключений в Java основана на использовании в программе следующих ключевых слов:**

* **try** – определяет блок кода, в котором может произойти исключение;
* **catch** – определяет блок кода, в котором происходит обработка исключения;
* **finally** – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.

Эти ключевые слова используются для создания в программном коде специальных обрабатывающих конструкций: try{}catch, try{}catch{}finally, try{}finally{}.

* **throw** – используется для возбуждения исключения;
* **throws** – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение.

**Пример использования ключевых слов в Java-программе:**

//метод считывает строку с клавиатуры

**public** String input() **throws** MyException {//предупреждаем с помощью throws,

// что метод может выбросить исключение MyException

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.in));

String s = **null**;

//в блок try заключаем код, в котором может произойти исключение, в данном

// случае компилятор нам подсказывает, что метод readLine() класса

// BufferedReader может выбросить исключение ввода/вывода

**try** {

s = reader.readLine();

// в блок catch заключаем код по обработке исключения IOException

} **catch** (IOException e) {

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

// в блоке finally закрываем поток чтения

} **finally** {

// при закрытии потока тоже возможно исключение, например, если он не был открыт, поэтому “оборачиваем” код в блок try

**try** {

reader.close();

// пишем обработку исключения при закрытии потока чтения

} **catch** (IOException e) {

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

}

}

**if** (s.equals("")) {

// мы решили, что пустая строка может нарушить в дальнейшем работу нашей программы, например, на результате этого метода нам надо вызывать метод substring(1,2), поэтому мы вынуждены прервать выполнение программы с генерацией своего типа исключения MyException с помощью throw

**throw** **new** MyException("String can not be empty!");

}

**return** s;

}

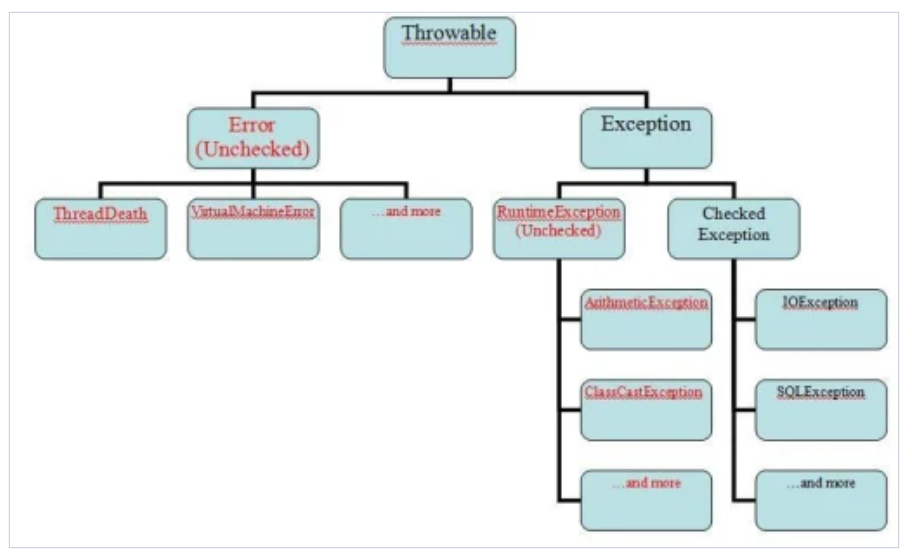
## Для чего нужен механизм исключений?

Посмотрим на пример из реального мира. Представьте, что на автомобильной дороге есть участок с аварийным мостом с ограниченной грузоподъемностью. Если по нему поедет автомобиль с массой, превышающей грузоподъемность моста, он может разрушиться, и ситуация для водителя может стать, мягко говоря, исключительной. Чтобы этого не произошло, дорожная служба заблаговременно устанавливает предупредительные знаки на дороге. Водитель автомобиля, глядя на предупреждающий знак, будет сравнивать массу своего автомобиля с разрешенной для проезда по мосту. Если она превышает ее – он поедет по объездному пути. Благодаря действиям дорожной службы водители грузового транспорта, во-первых, получили возможность заблаговременно изменять свой путь, во-вторых, предупреждены об опасности на основном пути, и, наконец, предупреждены о невозможности использования моста при определенных условиях.

Возможность предупреждения и разрешения исключительной ситуации в программе для ее продолжения – одна из причин использования исключений в Java. Механизм исключений также позволяет защитить написанный вами код (программный интерфейс) от неправильного использования пользователем за счет валидации (проверки) входящих данных. Давайте теперь на секунду побудем дорожной службой. Во-первых, вы должны знать места, где автомобилистов могут ждать неприятности. Во-вторых, вам нужно заготовить и установить предупредительные знаки. И, наконец, вам нужно предусмотреть объездные маршруты в случае опасности на основном пути. В Java механизм исключений работает похожим образом. На стадии разработки программы мы «ограждаем» опасные участки кода в отношении исключений с помощью блока try{}, предусматриваем «запасные» пути с помощью блока catch{}, в блоке finally{} мы пишем код, который выполняется в программе при любом исходе. В случаях, когда мы не можем предусмотреть «запасной путь» или намеренно хотим предоставить право его выбора пользователю, мы должны, по крайней мере, предупредить его об опасности. Почему? А вы только вообразите негодование водителя, который доедет до аварийного моста, по которому нельзя проехать, не встретив по дороге ни одного предупреждающего знака! В программировании при написании своих классов и методов мы не всегда можем предвидеть контекст их использования другими разработчиками в своих программах, поэтому не можем предвидеть на 100% правильный путь для разрешения исключительной ситуации. В то же время, правило хорошего тона — предупредить пользователей нашего кода о возможности исключительной ситуации. Механизм исключений Java позволяет нам сделать это с помощью throws – по сути, объявления общего поведения нашего метода, заключающееся в выбрасывании исключения, и предоставляя, таким образом, написание кода по обработке исключения в Java пользователю метода.

## Предупреждение о «неприятностях»

Когда вы не планируете обрабатывать исключение в своем методе, но хотите предупредить пользователей метода о возможных исключительных ситуациях — используйте ключевое слово throws. Это ключевое слово в сигнатуре метода означает, что при определенных условиях метод, может выбросить исключение. Такое предупреждение является частью интерфейса метода и предоставляет право пользователю на собственный вариант реализации обработчика исключения. После throws мы указываем тип выбрасываемого исключения. Обычно это наследники класса Exception Java. Поскольку Java является объектно-ориентированным языком, все исключения в Java представляют собой объекты.



## Иерархия исключений Java

При возникновении ошибки в процессе выполнения программы исполняющая среда JVM создает объект нужного типа из иерархии исключений Java – множества возможных исключительных ситуаций, унаследованных от общего «предка» – класса **Throwable**. **Исключительные ситуации, возникающие в программе, можно разделить на две группы:**

1. Ситуации, при которых восстановление дальнейшей нормальной работы программы невозможно
2. Восстановление возможно.

К первой группе относят ситуации, когда возникают исключения, унаследованные из класса Error. Это ошибки, возникающие при выполнении программы в результате сбоя работы JVM, переполнения памяти или сбоя системы. Обычно они свидетельствуют о серьезных проблемах, устранить которые программными средствами невозможно. Такой вид исключений в Java относится к неконтролируемым (unchecked) на стадии компиляции. К этой группе также относят RuntimeException – исключения, наследники класса Exception, генерируемые JVM во время выполнения программы. Часто причиной возникновения их являются ошибки программирования. Эти исключения также являются неконтролируемыми (unchecked) на стадии компиляции, поэтому написание кода по их обработке не является обязательным. Ко второй группе относят исключительные ситуации, предвидимые еще на стадии написания программы, и для которых должен быть написан код обработки. Такие исключения являются контролируемыми (checked). Основная часть работы разработчика на Java при работе с исключениями – обработка таких ситуаций.

## Создание исключения

При исполнении программы исключение генерируется JVM или вручную, с помощью оператора throw. При этом в памяти создается объект исключения и выполнение основного кода программы прерывается, а обработчик исключений **JVM** пытается найти способ обработать исключение.

## Обработка исключения

Создание блоков кода, для которых мы предусматриваем обработку исключений в Java, производится в программе с помощью конструкций try{}catch, try{}catch{}finally, try{}finally{}.



При возбуждении исключения в блоке try обработчик исключения ищется в следующем за ним блоке catch. Если в catch есть обработчик данного типа исключения – управление переходит к нему. Если нет, то JVM ищет обработчик этого типа исключения в цепочке вызовов методов до тех пор, пока не будет найден подходящий catch. После выполнения блока catch управление передается в необязательный блок finally. В случае, если подходящий блок catch не найден, JVM останавливает выполнение программы, и выводит стек вызовов методов – stack trace, выполнив перед этим код блока finally при его наличии. Пример обработки исключений:

**public** **class** Print {

**void** print(String s) {

**if** (s == **null**) {

**throw** **new** NullPointerException("Exception: s is null!");

}

System.out.println("Inside method print: " + s);

}

**public** **static** **void** main(String[] args) {

Print print = **new** Print();

List list= Arrays.asList("first step", **null**, "second step");

**for** (String s:list) {

**try** {

print.print(s);

}

**catch** (NullPointerException e) {

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

System.out.println("Exception was processed. Program continues");

}

**finally** {

System.out.println("Inside bloсk finally");

}

System.out.println("Go program....");

System.out.println("-----------------");

}

}

}

Результаты работы метода main:

Inside method print: first step

Inside bloсk **finally**

Go program....

-----------------

Exception: s is **null**!

Exception was processed. Program continues

Inside bloсk **finally**

Go program....

-----------------

Inside method print: second step

Inside bloсk **finally**

Go program....

-----------------

Блок finally обычно используется для того, чтобы закрыть открытые в блоке try потоки или освободить ресурсы. Однако при написании программы не всегда возможно уследить за закрытием всех ресурсов. Для облегчения нашей жизни разработчики Java предложили нам конструкцию try-with-resources, которая автоматически закрывает ресурсы, открытые в блоке try. Наш первый пример можно переписать так с помощью try-with-resources:

**public** String input() **throws** MyException {

String s = **null**;

**try**(BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.in))){

s = reader.readLine();

} **catch** (IOException e) {

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

}

**if** (s.equals("")){

**throw** **new** MyException ("String can not be empty!");

}

**return** s;

}

Благодаря возможностям Java, начиная с версии 7, мы также можем объединять перехват разнотипных исключений в одном блоке, делая код более компактным и читабельным. Например:

**public** String input() {

String s = **null**;

**try** (BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.in))) {

s = reader.readLine();

**if** (s.equals("")) {

**throw** **new** MyException("String can not be empty!");

}

} **catch** (IOException | MyException e) {

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

}

**return** s;

}

#### Итоги

Использование исключений в Java позволяет повысить отказоустойчивость программы за счет использования «запасных» путей, отделить логику основного кода от кода обработки исключительных ситуаций за счет использования блоков catch, а также дает нам возможность переложить обработку исключений на пользователя нашего кода с помощью throws.