1. **Принцыпы ООП (Инкапсуляция Наследование Полиморфизм)**

**Инкапсуляция** – ограничение доступа к полям и методам класса.

**Наследование** – наслдование класссов (их полей, методов) для использования, переопределения и расширения.

**Полиморфизм** – возможность работать с несколькими типами так, как будто это один и тот же тип. При этом поведение каждого типа будет уникальным, в зависимости от реализации. Дает возможность использовать наследников, как родителей. *Объект подкласса может использоваться всюду, где используется объект суперкласса.*

1. JVM / байт код Java
2. Загрузка классов (ClassLoader) <https://habrahabr.ru/post/103830/>

<http://java-online.ru/java-classloader.xhtml>

1. **Статическая и динамическая загрузка. Раннее и позднее связывание.**

**Статическая загрузка** класса происходит при использовании оператора "new".

**Динамическая загрузка** происходит "на лету" в ходе выполнения программы с помощью статического метода класса Class.forName(имя класса). Для чего нужна динамическая загрузка? Например мы не знаем какой класс нам понадобится и принимаем решение в ходе выполнения программы передавая имя класса в статический метод forName().

**Раннее связывание** - это когда метод, который будет вызван, известен во время компиляции, например, вызов статического метода. Раннее связывание происходит на этапе компиляции. Оно применяется при вызове обычных методов (не виртуальных).

**Позднее связывание** - это когда вызов метода может быть осуществлен только во время выполнения и у компилятора нет информации, чтобы проверить корректность такого вызова. В java это можно сделать при помощи рефлексии. Позднее связывание происходит во время выполнения. Выполняется оно при вызове виртуальных функций класса-потомка для определения того, какой именно метод следует вызывать.

Исходя из того, что раннее связывание выполняется на этапе компиляции, а позднее - в рантайме, первый вариант обладает лучшим быстродействием, однако второй необходим для реализации полиморфизма.

1. **Память приложения. Heap (куча) и Стек.**

Java **Heap** (куча) используется Java Runtime для выделения памяти под объекты и JRE классы. Создание нового объекта также происходит в куче. Здесь работает сборщик мусора: освобождает память путем удаления объектов, на которые нет каких-либо ссылок. Любой объект, созданный в куче, имеет глобальный доступ и на него могут ссылаться с любой части приложения.

**Стековая память** в Java работает по схеме LIFO (Последний-зашел-Первый-вышел). Всякий раз, когда вызывается метод, в памяти стека создается новый блок, который содержит примитивы и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок также перестает использоваться, тем самым предоставляя доступ для следующего метода. Размер стековой памяти намного меньше объема памяти в куче.

1. **Різниця і++ ++і (постфикс и префикс)**

Когда мы используем операции после переменной (i- -), мы вначале используем значение переменной для наших целей и только после этого изменяем ее.

Когда мы используем операции до переменной (++i), мы вначале изменяем значение переменной и только после этого используем значение переменной.

1. **Вложенные и анонимные классы.**

Вложенный (nested) класс - это класс, который объявлен внутри объявления другого класса.

class OuterClass { // Внешний класс (outer class)

static class StaticNestedClass { // вложенный статический класс

}

}

**1) Статические вложенные классы** не имеют доступа к нестатическим полям и методам обрамляющего класса, что в некотором роде аналогично статическим методам, объявленным внутри класса. Доступ к нестатическим полям и методам может осуществляться только через ссылку на экземпляр обрамляющего класса. Кроме этого, static nested классы имеют доступ к любым статическим методам внешнего класса, в том числе и к приватным. Статические члены классов (static nested classes) - как и любой другой статический метод, имеет доступ к любым статическим методам своего внешнего класса, в том числе и к приватным. К нестатическим полям и методам обрамляющего класса он не может обращатся напрямую. Он может использовать их только через ссылку на экземпляр класса родителя.

**2) Вложенный НЕстатический класс = внутренний.**

Внутренние классы в Java делятся на такие три вида:

* внутренние классы-члены (member inner classes);

Внутренние классы-члены ассоциируются не с самим внешним классом, а с его экземпляром. При этом они имеют доступ ко всем его полям и методам.

* локальные классы (local classes);

Локальные классы (local classes) определяются в блоке Java кода. На практике чаще всего объявление происходит в методе некоторого другого класса. Доступ к классу – только внутри метода.

* анонимные классы (anonymous classes).

Анонимный класс (anonymous class) - это локальный класс без имени.

new Thread(new Runnable() {

public void run() {

...

}

}).start();

На основании анонимного класса создается поток и запускается с помощью метода start класса Thread. Синтаксис создания анонимного класса базируется на использовании оператора new с именем класса (интерфейса) и телом новосозданного анонимного класса.

Основное ограничение при использовании анонимных классов - это невозможность описания конструктора, так как класс не имеет имени. Аргументы, указанные в скобках, автоматически используются для вызова конструктора базового класса с теми же параметрами. Вот пример:

class Clazz {

Clazz(int param) { }

public static void main(String[] args) {

new Clazz(1) { }; // правильное создание анонимного класса

new Clazz() { }; // неправильное создание анонимного класса

} }

Использование анонимных классов оправдано во многих случаях, когда:

* тело класса является очень коротким;
* нужен только один экземпляр класса;
* класс используется в месте его создания или сразу после него;

1. **Конструктори. Конструктор по замовчуванню**

Конструктор — это метод класса, который инициализирует новый объект после его создания. Имя конструктора всегда совпадает с именем класса, в котором он расположен.

Если конструктор не был указан, то при создании объектов будет использован конструктор по умолчанию (=пустой конструктор). Class A{ … A(){} }

Если вы используете какие — то конструкторы, то конструктор по умолчанию не создается. Таким образом, если у нас есть только конструктор от трех параметров, то мы никак не можем создать объект данного класса не вызвав его.

class Line {

protected int lenght;

Line(int l) { // Конструктор

length = l;

} }

Имея дело с перегруженными конструкторами, удобно один конструктор вызывать из другого через ключевое слово this. При выполнении конструктора this() сначала выполняется перегруженный конструктор, который соответствует списку параметров. Затем выполняются операторы, находящиеся внутри исходного конструктора, если таковые существуют. Вызов конструктора this() должен быть первым оператором в конструкторе.

class Cat{

int age;

int birthday;

// Инициализируем переменные явно

Cat(int i, int j) {

age = i;

birthday = j;

}

// Инициализируем переменные одним и тем значением

Cat(int i) {

this(i, i); // вызывается Cat(i, i);

}

}

1. **Як викликаються конструктори при наслідуванні**

Від батьківского класу до наслідників.

В каком порядке вызываются конструкторы классов, образующих иерархию, при ее создании? В иерархии классов конструкторы вызываются в порядке наследования, начиная с суперкласса, и заканчивая подклассом. Более того, поскольку super () должен быть первым оператором, выполняемым в конструкторе подкласса, этот порядок остается неизменным, независимо от того, используется ли форма super (). Если конструктор super () не применяется, программа использует конструктор каждого суперкласса, заданный по умолчанию или не содержащий параметров. Если немного подумать, становится ясно, что выполнение конструкторов в порядке наследования имеет смысл. Поскольку суперкласс ничего не знает о своих подклассах, любая инициализация, которую он должен выполнить, полностью независима и, возможно, обязательна для выполнения любой инициализации, выполняемой подклассом. Поэтому она должна выполняться первой.

1. **Як в конструкторі нащадка викликати конструктор батька**

Конструкторы в Java не наследуются. Вам нужно явным образом определить конструктор в классе-наследнике. Если вы хотите, чтобы он делал то же, что и конструктор предка, то нужно явно вызвать конструктор предка при помощи ключевого слова super. Обратите внимание, что вызов конструктора класса-предка должен быть первой строкой конструктора класса-наследника.

class B extends A {

public B(int x, int y){

super (x,y)

}

}

Если у предка есть конструктор по умолчанию (без параметров), то в наследнике его переопределять не нужно. Если у предка нет конструктора без параметров, то класс-потомок должен определить свой конструктор.

Ключевое слово super – вызываем конструктор или метод родительского класса, который примет наши параметры.

1. **Что такое null**

null это ключевое слово в Java, значение по умолчанию любых ссылочных типов, проще говоря, для всех объектов.

1. **Abstract classes VS Interfaces.**

|  |  |
| --- | --- |
| Abstract classes | Interfaces |
| Описывает *сам класс. «Класс-заготовка»* | Описывает *что класс умеет* |
| Может иметь работающие методы | Только объявление методов |
| Наследники обязательно должны реализовать только абстрактные методы | Имплементаторы обязательно реализуют все методы |
| В Java множественного наследования нет | Можно имплементить много интерф-сов |
| Может содержать разные поля | Все поля только *static final* |

Вообще объект абстрактного класса создать нельзя. Однако, если в АК нет абстрактных методов, можно создать объект анонимного класса, который его наследует и ничего в нем не меняет.

MyAbstractClass ac = new MyAbstractClass() {};

1. **Що таке перегрузка і перезагрузка**

Перегрузка – методы с одинаковым именем, но с разными наборами параметров. При наследовании также можно перегружать методы предка.

Перезагрузка – переопределение уже существующего (у предка) метода.

public class Example{ // класс Object имеет метод toString() = className@hashcode

@Override // аннотация

public String toString(){ … }

}

1. **Що таке static/final. Статические классы и методы.**

|  |  |  |
| --- | --- | --- |
|  | static | final |
| поля | Доступны без создания экземпляра класса,  Статические переменные едины для всех объектов класса (одинаковая ссылка). | Не могут быть изменены после первой инициализации. (Или объявлены и сразу иниц-ны, или объявлены и иниц-я в конструкторе) |
| методы | Доступны без создания экземпляра класса,  Одинаковы для всех объектов класса.  static методы соответственно могут взаимодействовать только со static полями | Коли треба заборонити перевизначення (override) вашого методу у класах-нащадках (перевантаження (overload) дозволене) |
| классы | Статические вложенные классы | Заборона наслідувати клас |

* К статическим методам и переменным можно обращаться через имя класса
* Статические блоки выполняются во время загрузки класса
* Локальные переменные не могут быть объявлены как static
* Абстрактные методы не могут быть static
* Static поля не сериализуются (только при реализации интерфейса Serializable)
* Только static переменные класса могут быть переданы в конструктор с параметрами, вызывающийся через слово super(//параметр//) или this(//параметр//)

!!! Ключевое слово final для объектных типов гарантирует неизменяемость лишь ссылки, но не самого объекта. Например, если у вас есть final-ссылка на ArrayList<T>, вы тем не менее можете добавлять в него новые элементы или изменять существующие.

public class Sphere {

public static final double PI = 3.141592653589793; // this might as well be a constant

public final double radius;

Sphere(double r) {

radius = r;

} … }

\* Статические методы могут работать только с статическими полями класса. Доступ к нестатическим полям и методам может осуществляться только через ссылку на экземпляр класса.

КАКИМ ОБРАЗОМ ИЗ ВЛОЖЕННОГО КЛАССА ПОЛУЧИТЬ ДОСТУП К ПОЛЮ ВНЕШНЕГО КЛАССА.

* Если класс внутренний то: *OuterClass.this.поле\_внешнего*\_класса.
* Если класс статический внутренний(вложенный),то в методе нужно создать объект внешнего класса, и получить доступ к его полю. Или второй вариант объявить это поле внешнего класса как static

1. **Порядок инициализации. Статические блоки**

Статические блоки вызываются в момент инициализации класса (когда ClassLoader будет грузить его в MetaSpace) **ТОЕСТЬ когда создается первый экземпляр класса или в момент первого обращения к static полю или методу класса**, и могут использоваться для иницализации статических переменных. Зачем вызывать каждый раз инициализацию статических переменных в конструкторе, если можно сделать это единожды, таким образом не тратить ресурсы на повторные инициализации?

Поля с модификатором static могут быть использованы даже если не было создано не одного экземпляра класса (ни одного объекта) - и они являются общими для всех экземпляров класса.

static {

...

}

Порядок инициализации таков:

1. Статические элементы родителя
2. Статические элементы наследника
3. Глобальные переменные родителя
4. Конструктор родителя
5. Глобальные переменные наследника
6. Конструктор наследника

class Insect {

private int i = 9; [9]

protected int j; [10]

Insect() { [8]

print("i = " + i + ", j = " + j); [11]

j = 39; [12]

}

private static int x1 = printInit("static Insect.x1 initialized"); [2]

static int printInit(String s) { [3]

print(s);

return 47;

}

}

public class Beetle extends Insect {

private int k = printInit("Beetle.k initialized"); [13]

public Beetle() { [7]

print("k = " + k); [14]

print("j = " + j); [15]

}

private static int x2 = printInit("static Beetle.x2 initialized"); [4]

public static void main(String[] args) { [1]

print("Beetle constructor"); [5]

Beetle b = new Beetle(); [6]

}

}

1. **break / continue / return**

Используя оператор *break*, можно вызвать немедленное завершение цикла, пропуская условное выражение и другой код в теле цикла. Когда приложение встречает оператор break внутри цикла, оно прекращает выполнение цикла и передаёт управление оператору, следующему за циклом. При использовании вложенных циклов оператор break осуществляет выход только из самого внутреннего цикла, не оказывая влияния на внешний цикл.

В случае циклов while и do-while оператор *continue* приводит к пропуску операторов тела цикла, не выполненных на данной итерации, и к передаче управления на код, проверяющий условие завершения цикла. Оператор continue, стоящий в цикле for, приводит к тому, что управление немедленно передается третьей части заголовка цикла, т.е. коду, который, выполняется на каждой итерации после операторов тела цикла

Оператор *return* используют для выполнения явного выхода из метода. Оператор можно использовать в любом месте метода для возврата управления тому объекту, который вызвал данный метод. Таким образом, return прекращает выполнение метода, в котором он находится.

1. **Type casting (Приведение типов)**

**Downcasting** - нисходящее приведение, то есть приведение от предка к потомку предку подтипу возможно если только исходная переменная является подтипом приводимого типа. Ошибка может возникать runtime или на этапе компиляции.

**Upcasting** - восходящее приведение типов или приведение типов от потомка к предку, разрешено всегда.

Запрещено приведение типов не "лежащих" в одной иерархии (как в случае Double и Stringили LinkedList и ArrayList). [Class Hierarchy](http://docs.oracle.com/javase/6/docs/api/java/util/package-tree.html)

Object object;

String s=(String )object;

//разрешено, поскольку тип переменной object может быть String'ом, но это выяснится только в runtime

Double object=new Double(1.0);

String s=(String )object;

//ошибка будет на этапе компиляции поскольку тип object известен компилятору и он не приводим к типу String

Object object=new Double(1.0);

Number n=(Number )object;

//разрешено, поскольку Double является подтипом Number

Приведение  типов это установка типа переменной или объекта отличного от текущего. В ждава есть два вида приведения:

* автоматическое
* не автоматическое

Автоматическое  происходит например: byte-> short->int->long->float->double

то есть если мы расширяем тип, то явное преобразование не требуется, приведение происходит автоматически. Если же мы сужаем, то необходимо явно указывать приведение типа.

В случае же с объектами, то мы можем  сделать автоматическое приведение от наследника к родителю, но никак  не наоборот, тогда вылетит ClassCastException.

1. **Immutable objects**

Immutable объект - это объект, состояние которого после создания невозможно изменить. В случае Java это значит что все поля экземпляра у класс отмечены как final и являются примитивами или тоже immutable типами.

Пример:

public **final class** ImmutableTest {

**private final** String name; **private final** Date dateOfBirth; **private final** String[] skills;

public **ImmutableTest(String name, Date dateOfBirth, String... skills)** {

this.name = name;

**//при конструировании объекта, полям присваиваются новые объекты, вместо ссылок**

this.dateOfBirth = new Date(dateOfBirth.getTime());

this.skills = skills.clone();

}

public String getName() {

return name;

}

**// Возвращаем новый объект Date, вместо передачи ссылки на текущий**

public Date getDateOfBirth() {

return new Date(dateOfBirth.getTime());

}

**// Возвращаем клонированный массив, вместо ссылки на объект массива**

public String[] getSkills() {

return skills.clone();

}

}

После создания экземпляра его модификация невозможна.

Простейший пример **immutable** класса из JDK это String. Любые методы, которые вы вызовите на строке (например description.toLowerCase()) вернут новую строку, а не модифицируют исходную.

Пример **mutable** класс из JDK - Date. Например myDate.setHours(x) модифицирует сам экземпляр myDate!

В случае многопоточного программирования преимущества immutable классов очевидны: после создания объекты можно передавать другим потокам и они всегда будут в актуальном состоянии. Т.е. вам не надо проверять не устарело ли состояние вашего экземпляра и не модифицировал ли его другой поток пока вы с ним работаете. Например, у вас есть метод bill(Date endDate), в нём вы наивно проверяете соответствие endDate каким-то предварительным условиям и начинаете с ней работать. В этот момент другой поток может изменить endDate, например установит её глубоко в прошлое. Последствия могут быть самыми удивительными.

1. Методы обратного вызова (callback)
2. **Serializable / serialization**

Сериализация — процесс перевода какой-либо структуры данных в последовательность битов. Обратной к операции сериализации является операция десериализации (структуризации) — восстановление начального состояния структуры данных из битовой последовательности. Сериализация используется для передачи объектов по сети и для сохранения их в файлы.

В Листинге 1 только одна вещь отличается от создания нормального класса, это реализация интерфейса java.io.Serializable. Интерфейс Serializable это интерфейс-маркер; в нём не задекларировано ни одного метода. Но говорит сериализующему механизму, что класс может быть сериализован. Теперь у нас есть всё необходимое для сериализации объекта, следующим шагом будет фактическая сериализация объекта (сохранение состояния экземпляра TestSerial в файл с именем temp.out). Она делается вызовом метода writeObject() класса java.io.ObjectOutputStream, как показано в листинге 2.

import java.io.Serializable; // Листинг 1

class TestSerial implements Serializable {

public byte version = 100;

public byte count = 0;

}

public static void main(String args[]) throws IOException { // Листинг 2

FileOutputStream fos = new FileOutputStream("temp.out");

ObjectOutputStream oos = new ObjectOutputStream(fos);

TestSerial ts = new TestSerial();

oos.writeObject(ts);

oos.flush();

oos.close();

}

Для воссоздания объекта из файла, необходимо применить код из листинга 3.

public static void main(String args[]) throws IOException { // Листинг 3

FileInputStream fis = new FileInputStream("temp.out");

ObjectInputStream oin = new ObjectInputStream(fis);

TestSerial ts = (TestSerial) oin.readObject();

System.out.println("version="+ts.version);

}

Восстановление объекта происходит с помощью вызова метода oin.readObject(). В методе происходит чтение набора байт из файла и создаие точной копии графа оригинального объекта. oin.readObject() может прочитать любой сериализованный объект, поэтому необходимо полученный объект приводить к конкретному типу. Выполненный код выведет version=100 в стандартный вывод.

1. **Generics (Обобщения)**

Обобщения - это параметризованные типы. С их помощью можно объявлять классы, интерфейсы и методы, где тип данных указан в виде параметра. Обобщения добавили в язык безопасность типов.

Generics позволяют абстрагировать множество типов. Наиболее распространенными примерами являются Коллекции.

Вот типичное использование такого рода (без Generics):

1. List myIntList = new LinkedList();

2. myIntList.add(new Integer(0));

3. Integer x = **(Integer)** myIntList.iterator().next();

Как правило, программист знает, какие данные должны быть в List'e. Тем не менее, стоит обратить особое внимание на Приведение типа (**"Cast"**) в строчке 3. Компилятор может лишь гарантировать, что метод next() вернёт Object, но чтобы обеспечить присвоение переменной типа Integer правильным и безопасным, требуется Cast. Cast не только создает беспорядки, но дает возможность появление ошибки "Runtime Error" из-за невнимательности программиста.

И появляется такой вопрос: "Как с этим бороться? " В частности: "Как же зарезервировать List для определенного типа данных?"

Как раз такую проблему решают Generics.

1. List**<Integer>** myIntList = new LinkedList**<Integer>**();

2. myIntList.add(new Integer(0));

3. Integer x = myIntList.iterator().next();

Рассмотрим пример с обобщением.

class Gen<T> {

T ob; // объявление объекта типа T

Gen(T o) { // Передать конструктору ссылку на объект типа T

ob = o;

}

T getob() { // Вернуть ob

return ob;

}

void showType() { // Показать тип T

System.out.println("Тип T: " + ob.getClass().getName());

}

}

/// Работаем с обобщённым классом

Gen<Integer> iOb; // Создаём Gen-ссылку для Integer

iOb = new Gen<Integer>(77); // Создаём объект Gen<Integer>

iOb.showType(); // Показать тип данных, используемый iOb

System.out.println("Значение: " + iOb.getob()); // Получить значение iOb

Можно указать два и более параметров типа через запятую.

С Java 7 существует укороченная запись, когда справа не указывается тип. Компилятор сам догадается по левой части выражения (Списочный массив, который использует обобщещние):

ArrayList<String> catNames = new ArrayList<>(); // вместо new ArrayList<String>();

1. **Hashcode & equals** [**https://habrahabr.ru/post/168195/**](https://habrahabr.ru/post/168195/)

1. Хеширование - преобразование массива входных данных произвольной длины в битовую строку фиксированной длины, выполняемое определённым алгоритмом. Хеш-код вычисляется на основании содержимого объекта (значения полей)

* У одинаковых объектов (или одного и того же) хеш-коды одинаковые.
* Если хеш-коды разные, то и входные объекты гарантированно разные.
* Если хеш-коды равны, то входные объекты не всегда равны.
* *JVM использует случайные числа, так что при каждом запуске программы хеш-код того же объекта скорее всего будет разным.*

int hCode;

hCode = object.hashCode();

Заглянем в исходный код метода hashCode() в классе Object:

public native int hashCode();

Вот собственно и вся реализация. Ключевое слово native означает, что реализация данного метода выполнена на другом языке (C++). При вычислении хэш-кода для объектов класса Object по умолчанию используется Park-Miller RNG алгоритм. В основу работы данного алгоритма положен генератор случайных чисел. Это означает, что при каждом запуске программы у объекта будет разный хэш-код. Получается, что используя реализацию метода hashCode() от класса Object, мы при каждом создании объекта класса, будем получать разные хеш-коды. Мало того, перезапуская программу, мы будем получать абсолютно разные значения, поскольку это просто случайное число.

2. Метод equals() сравнивает содержимое объектов.

public class DemoBlackBox {

public static void main(String[] args) {

BlackBox object1 = new BlackBox(5, 10);

BlackBox object2 = new BlackBox(5, 10);

}}

Содержимое этих объектов одинаково, то есть эквивалентно. Для проверки эквивалентности в классе Object существует метод equals(), который сравнивает содержимое объектов и выводит значение типа boolean true, если содержимое эквивалентно, и false — если нет.

object1.equals(object2);// должно быть true, поскольку содержимое объектов эквивалентно

Эквивалентность и хеш-код тесно связанны между собой, поскольку хеш-код вычисляется на основании содержимого объекта (значения полей) и если у двух объектов одного и того же класса содержимое одинаковое, то и хеш-коды должны быть одинаковые.

Рассмотрим исходный код метода equals() в классе Object.

public boolean equals(Object obj) {

return (this == obj);

}

При сравнение объектов, операция “==” вернет true лишь в одном случае — когда ссылки указывают на один и тот-же объект. В данном случае не учитывается содержимое полей.

Теперь понято, почему *Object.equals() работает не так как нужно, ведь он сравнивает ссылки, а не содержимое объектов*.

Поэтому, при создании пользовательского класса, *принято переопределять методы hashCode() и equals() таким образом, что бы учитывались поля объекта*.

Это можно сделать вручную либо воспользовавшись средствами генерации исходного кода в IDE. Например, в Eclipse это Source → Generate hashCode() and equals()...

1. Индексация
2. **Iterator / foreach**

An Iterable is a simple representation of a series of elements that can be iterated over. It does not have any iteration state such as a "current element". Instead, it has one method that produces an Iterator.

An Iterator is the object with iteration state. It lets you check if it has more elements using hasNext() and move to the next element (if any) using next().

An implementation of Iterable is one that provides an Iterator of itself:

public interface Iterable<T>{

Iterator<T> iterator();

}

An iterator is a simple way of allowing some to loop through a collection of data without assignment privileges (though with ability to remove).

public interface Iterator<E>{

boolean hasNext();

E next();

void remove();

}

iterator VS foreach.docx

1. **Comparable / Comparator**

Comparator Comparable Examples.htm

Comparable это интерфейс класса, чьи объекты можно сравнивать, а коллекции соответственно сортировать методом Collections.sort(). Компаратор нужен в тех случаях когда порядок сортировки надо применить к структурам данных типа sorted set и картам отображения (maps) или к коллекциям объектов которые не имеют возможность задания порядка типа "природная сортировка". Природной называется сортировка в порядке от меньшего к большему (1 < 2 < 3 или A < B < C).

Comparator это интерфейс, который реализуется вне класса, объекты которого будут сравниваться. Метод Collections.sort() принимает реализующие его объекты как дополнительный, необязательный параметр, для того, чтобы использовать его алгоритм сортировки. Смысл использовать Comparator имеет тогда, когда сравниваешь объекты чужого класса, где сравнение не реализовано, или когда хочешь иметь возможность сравнивать свой класс несколькими способами. Насаждает сортировку объектов для каждого класса которые имплементирует этот интерфейс.

Используйте Comparable в пределах собственных классов, а Comparator применительно к структурам типа Map, Set или List.

1. **Reflection**

Рефлексия позволяет вам получить информацию о внутреннем строении класса — поля, методы и т.д. — и позволяет обратиться к полям, методам и другим артифактам через эту информацию. Причем, что самое важное — это все можно делать уже при исполнении программы — как часто говорят, в рантайме (runtime).

Если брать технические возможности, то в первую очередь можно выделить две группы действий:

1. Можно узнать всю информацию о классе — модификаторы, методы, поля, конструкторы, константы, суперклассы, реализуемые классом интерфейсы.
2. Можно работать с классом (объектом), а именно — создать объект класса, выполнить методы, получить или установить значения полей.
3. Фабрики
4. **Singletones**

Шаблон Singleton накладывает ограничения на создание экземпляра класса и гарантирует, что в JVM (виртуальной джава машине) существует только один экземпляр данного класса. Класс Singleton-а должен иметь глобальную точку доступа для получения экземпляра класса. Чаще всего Singleton-классы создаются для работы с файловыми системами, базами данных и т.п..

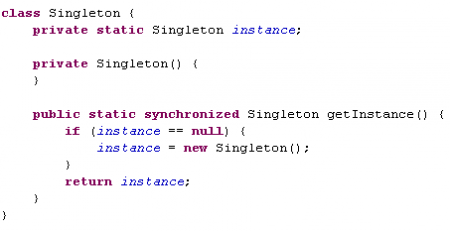
Область применения

1) В системе должно существовать не более одного экземпляра заданного класса. Даже при использовании мультипоточности.

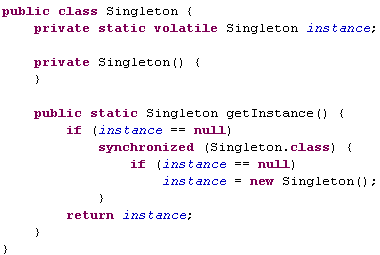
2) Экземпляр должен быть легко доступен для всех клиентов данного класса.

3) Создание объекта on demand, то есть, когда он понадобится первый раз, а не во время инициализации системы.

В Java можно реализовать через статический конструктор и синхронизированный метод для создания/вызова.



synchronized в методе getInstance снижает производительость при каждом вызове метода, так что можно перенести synchronized в место ПЕРВОГО создания инстанса. (подход Double-Checked Locking) и указание инстанса как volatile.



+ Ленивая инициализация

+ Высокая производительность

1. **Сравнение стрингов**

* str1.toLowerCase().contains(str2.toLowerCase()) = boolean
* equals - просто сравнивает строки
* compareTo - определяет, какая строка больше. Для этого метод сначала сравнивает первые буквы строк, потом вторы и так до тех пор, пока не дойдет до конца строки или не встретит различие. Большей считается строка, чей символ больший по алфавиту.
* Оператор == сравнивает ссылки

Однако в некоторых случаях строки гарантированно представлены одним и тем же объектом благодаря пулу строк. Подобная оптимизация существует в языках со сборкой мусора в таком виде: строка является неизменным объектом, и присваивание a=b не создаёт новой строки. Недостаток этого решения в том, что программист должен знать и использовать внутренний механизм построения строк наподобие StringBuilder (Java).

Используйте метод concat, если слияние нужно провести только один раз, для остальных случаев рекомендовано использовать или оператор "+" или StringBuffer / StringBuilder. Также стоит отметить, что получить NPE (NullPointerException), если один с операндов равен null, невозможно с помощью оператора "+" или "+=", чего не скажешь о методе concat.

1. **Access modifiers (private default protected public), описати дефолтний. Как получить доступ к прайват полю другого класса.**

|  |  |  |
| --- | --- | --- |
|  | поле / метод | класс |
| public | Будь-де | Будь-де |
| protected | В межах пакету + усі нащадки | - |
| default | В межах пакету | В межах пакету |
| private | В межах класу | - |

Доступ к прайват полю другого класса – через геттер.

1. **Multithreading. Многопоточность, процессы. Що таке потік і що таке процес.**

**extends Thread / implements Runnable**

Один **поток** – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

**Процесс** — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса, но бывают и исключения (например, браузер Chrome создает отдельный процесс для каждой вкладки, что дает ему некоторые преимущества, вроде независимости вкладок друг от друга). Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств).

В классе **Thread** имеется несколько методов, которые можно переопределить в порожденном классе. Из них обязательному переопределению подлежит только метод **run()**. Этот же метод, безусловно, должен быть определен и при реализации интерфейса **Runnable**. Некоторые программирующие на Java считают, что создавать подкласс, порожденный от класса **Thread**, следует только в том случае, если нужно дополнить его новыми функциями. Так, если переопределять любые другие методы из класса **Thread** не нужно, то можно ограничиться только реализацией интерфейса **Runnable**. Кроме того, реализация интерфейса **Runnable** позволяет создаваемому потоку наследовать класс, отличающийся от **Thread**.

Multithreading.docx

1. **Synchronized / volitale**

To make a method synchronized, simply add the synchronized keyword to its declaration:

public class SynchronizedCounter {

private int c = 0;

public synchronized void increment() { c++; }

public synchronized void decrement() { c--; }

public synchronized int value() { return c; }

}

If count is an instance of SynchronizedCounter, then making these methods synchronized has two effects:

* First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
* Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Слово synchronized говорит о том, что прежде чем поток сможет вызвать этот метод у нашего объекта, он должен «захватить» наш объект и потом выполнить нужный метод. Если у класса 2, 3 и более методов synchronized, то при выполнении хотя бы одного, блокируется весь объект. Это значит, что все методы, обозначенные как synchronized недоступны для других потоков. И еще раз — сначала «захватили», потом выполнили метод, потом «отпустили». Теперь объект свободен и кто первый успел из потоков его захватить — тот и прав.

В случае если метод объявлен как static, то объектом-монитором становится класс целиком и доступ к нему блокируется на уровне всех объектов этого класса.

!!!!! если есть static synchronized и просто synchronized то его тоже нужно сделать static, так как это получаются разные мониторы (весь класс у static synchronized и экземаляр у просто synchronized).

private volatile int a;

Модификатор volatile накладывает некоторые дополнительные условия на чтение/запись переменной. Важно понять две вещи о volatile переменных:

* Операции чтения/записи volatile переменной являются атомарными.
* Результат операции записи значения в volatile переменную одним потоком, становится виден всем другим потокам, которые используют эту переменную для чтения из нее значения.

1. **Try/catch/finally**

try{

…

} catch (MyException me) { // Собственная ошибка

…

} catch (Exception e) { // Словим что-угодно

…

} finally { // этот блок выполнится в любом случае

…

}

1. **Throw Exception / Custom Exceptions**

Exceptions.docx

1. **Отношения (Ассоциация, Агрегация, Композиция).**

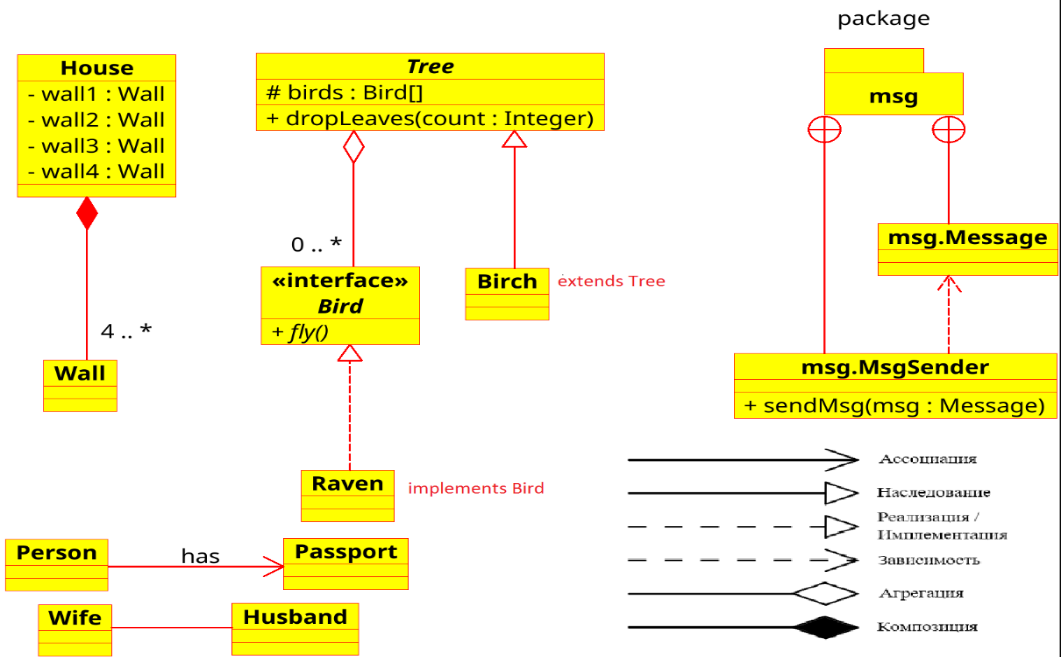
* Наследование = обобщение / расширение.
* Ассоциация = объекты определенных классов как-то связаны в общем виде (могут ссылаться один на другой). СТУДЕНТ – ПРЕПОДОВАТЕЛЬ
* Агрегация = СТУДЕНТ – КРУЖОК // студент может посещать много кружков
* Композиция = ДВИГАТЕЛЬ – МАШИНА // двигатель одновременно может быть только в одной машине.

На практике, важное различие - время жизни.

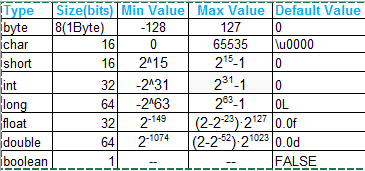
Если время жизни у объектов одинаковое, то это композиция и отношение можно рассматривать как "владение".

Если время жизни разное, то это просто агрегация. Мы осведомлены об этом объекте, однако он живет сам по себе.

1. **Диаграмма классов (UML)**

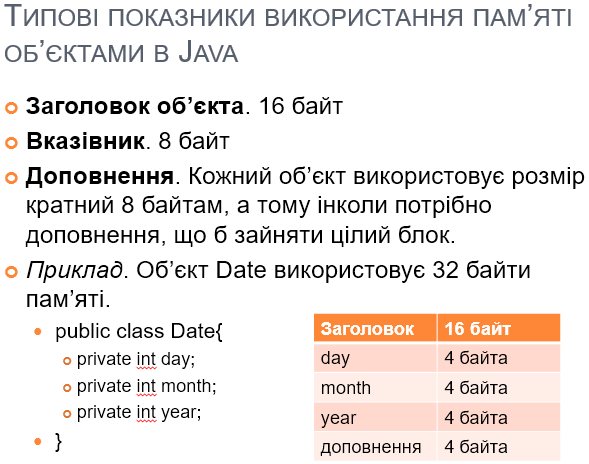
****

1. Main patterns
2. Garbage collector
3. Динамическое связывание
4. **Примитивные типы данных Java. Хранение в памяти.**



\* the size of a boolean variable in Java is implementation dependent.

Массивы: xN + 24 байта, например, int[] = 4N+24.

****

1. Структуры данных <https://ru.wikipedia.org/wiki/Структура_данных> {
2. **Стек і черга, різниця.**

Стек – принцып Last In First Out.

Очередь – First In First Out.

1. Деревья
2. BST
3. 2-3 trees
4. Червоно-чорні дерева
5. Проход дерева (в глубину, вширь)

}

1. **Коллекции** <http://docs.oracle.com/javase/tutorial/collections/index.html>

<https://dzone.com/articles/an-introduction-to-the-java-collections-framework> {

1. **Коллекции, какие классы наследуют (Collection, Set, List, Queue)**

Коллекции предназначены для хранения ссылок на объекты

* именно ссылок (сами объекты хранятся не в коллекциях, а в куче)
* именно на объекты (данные примитивных типов храниться в коллекциях не могут)
* **Collection** – базовый интерфейс для всех коллекций (кроме ассоциативных массивов). Применяется когда нужно передавать коллекции максимально обобщенно, не привязываясь к их типу (Set / List / Queue ...)

Операции:

* Изменение

**add, remove, clear**

* Просмотр

**isEmpty, size, contains, iterator**

* Групповые операции

**addAll** (объединение)

**removeAll** (вычитание)

**retainAll** (пересечение)

**containsAll** (определяет является ли коллекция подмножеством другой коллекции)

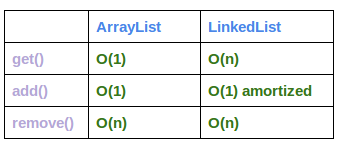
**toArray** (Преобразование в массив)

* **Set** – Повторы не допускаются, желательно использовать только с immutable объектами. (Чтобы не было двух Сереж с разными волосами).Все методы такие же как в Collection. Получив ссылку типа Collection с помощью instanceof Set можно заранее убедиться в отсутствии дубликатов в коллекции.
* **List –** Допускаются повторы и null. Порядок элементов при просмотре гарантируется Представляет собой «массив переменной длины».

Является наследником Collection. Кроме этого имеет свои методы:

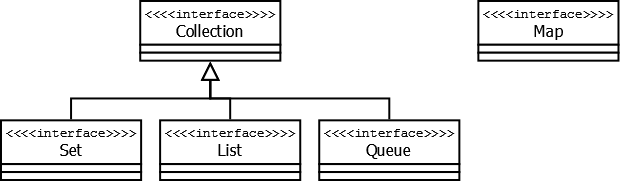
* **get, set**
* **add, remove (по индексу)**
* **indexOf, lastIndexOf**
* **listIterator**
* **subList**

Производительность вставки/удаления и произвольного доступа **существенно** зависит от реализации.

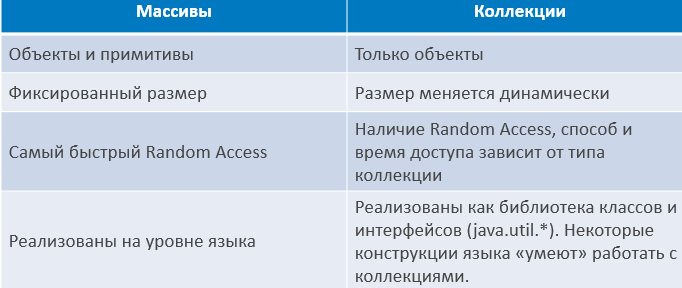


* **Queue -** интерфейс описывает коллекции с предопределённым способом вставки и извлечения элементов, а именно — очереди FIFO (first-in-first-out). Помимо методов, определённых в интерфейсе Collection, определяет дополнительные методы для извлечения и добавления элементов в очередь.

1. **Базові інтерфейси колекцій**



1. **Arrays vs Collections**



1. **HashSet** Уникальные данные (сравнение по хеш-коду). В классах объектов, заносимых в HashSet, hashCode() должен быть переопределен (override). Пример – страны. null допускается.
2. **LinkedHashSet** … хранит данные в порядке добавления. null допускается.
3. **TreeSet** (Black-Red Tree) … хранит отсортированые данные. Нужен реализованый Компаратор для хранения собственных типов, null допускается. При natural ordering null не допускается.
4. **ArrayList** <https://habrahabr.ru/post/128269/> Допускаются одинаковые элементы.

Плохо для очень больших или часто пополняющихся коллекций (основан на базе массива, при приближении к определенному уровню заполнения автоматически расширяется (создается бОльший массив и в него копируется старый)). При всередину массив также пересоздается.

Вы можете использовать конструктор ArrayList(capacity) и указать свою начальную емкость списка. При удалении элементов размер не меняется. Если заранее известно, сколько элементов следует хранить, то перед заполнением массива вызовите метод ensureCapacity(): mCatNames.ensureCapacity(100);

Первоначальную ёмкость можно задать и в конструкторе в качестве параметра. ArrayList<String> mCatNames = new ArrayList<>(100);

Если вы уверены, что списочный массив будет иметь постоянный размер, то можете использовать метод trimToSize(). Это может способствовать рациональному использованию памяти.

Реализует маркерный интерфейс RandomAccess (Его наличие означает быструю работу методов произвольного доступа к элементам (по индексу) таких как get, set). null допускается.

1. **LinkedList** <https://habrahabr.ru/post/127864/> Допускаются одинаковые элементы, хранятся в порядке добавления. RandomAccess отсутствует (медленный доступ к элементам) но быстрое добавление. null допускается.
2. **PriorityQueue** - единственная прямая реализация интерфейса Queue (не считая LinkedList, который больше является списком, чем очередью). Особенностью данной очереди является возможность управления порядком элементов. По-умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи объекта Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.
3. **HashMap** <https://habrahabr.ru/post/128017/> HashMap представляет из себя массив записей (Map.Entry<K,V>). Ключи и значения могут быть любых типов, в том числе и null. На основании хэшкода ключа вычисляется ячейка массива, куда нужно положить запись Map.Entry<K,V>. При совпадении хэшкодов в соответствующей ячейке создается LinkedList и значения вставляются в него. При поиске, сначала по хэшкоду определяется номер ячейки массива, и если в ячейке LinkedList, в котором хранятся объекты из Ключа, Значения и ссылки на следующий элемент листа, то, чтобы взять правильное значение с листа, используется equals для ключей.

Элементы хранятся по принципу natural ordering.

Map<String, String> map = new HashMap<String, String>();  
map.put("1", "a");  
map.put("1", "2");

**Hashtable** – как HashMap, но потокобезопасный (что уменьшает производительность). Не разрешает использование null в качестве ключей и значений. Элементы хранятся по принципу reversed natural ordering.

1. **TreeMap** – расширяет класс AbstractMap и реализует интерфейс NavigatebleMap. Класс создает коллекцию, которая применяет красно-черное дерево для хранения элементов. Сортирует элементы по ключу в естественном порядке или по заданому компаратору. Гарантирует достаточно буструю (log(n)) скорость доступа для операций containsKey, get, put и remove.

// Map – это объект, хранящий попарно значения(values) и ключи(keys).

}



1. Потоки ввода / вывода.
2. Алгоритмы поиска и сортировки
3. Table <http://docs.oracle.com/javase/tutorial/uiswing/components/table.html>
4. View <https://docs.oracle.com/javase/7/docs/api/javax/swing/text/View.html>
5. JSF
6. Java Mail
7. **JDBC Template / Prepared statements**

JDBC.docx

1. ORM / JPA / Hibernate
2. **JSP**

JSP (JavaServer Pages) — технология, позволяющая веб-разработчикам создавать содержимое, которое имеет как статические, так и динамические компоненты.

1. **MVC + чому не можна об'єднати два рівні в MVC**

MVC— схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо.

* Модель (Model) предоставляет данные представлению и реагирует на команды контроллера, изменяя свое состояние.
* Представление (View) отвечает за отображение данных модели пользователю, реагируя на изменения модели.
* Контроллер (Controller) интерпретирует действия пользователя, оповещая модель о необходимости изменений.

Начинающие программисты очень часто трактуют архитектурную модель MVC как *пассивную модель MVC*: модель выступает исключительно совокупностью функций для доступа к данным, а контроллер содержит бизнес-логику. В результате — код моделей по факту является средством получения данных из СУБД, а контроллер — типичным модулем, наполненным бизнес-логикой (см. «скрипт» в терминологии веб-программирования). В результате такого понимания — MVC-разработчики стали писать код, который Pádraic Brady охарактеризовал как Fat Stupid Ugly Controllers:

Среднестатистический ТТУК получал данные из БД (используя уровень абстракции базы данных, делая вид, что это модель) или манипулировал, проверял, записывал, а также передавал данные в Представление. Такой подход стал очень популярен потому, что использование таких контроллеров похоже на классическую практику использования отдельного php-файла для каждой страницы приложения.

В объектно-ориентированном программировании используется *активная модель MVC*, где модель — это не только совокупность кода доступа к данным и СУБД, но и вся бизнес-логика; также, модели могут инкапсулировать в себе другие модели. Контроллеры же, — как элементы информационной системы, — ответственны лишь за:

* Приём запроса от пользователя;
* Анализ запроса;
* Выбор следующего действия системы, соответственно результатам анализа (например, передача запроса другим элементам системы).

1. **JavaBean**

JavaBeans — классы в языке Java, написанные по определённым правилам. Они используются для объединения нескольких объектов в один для удобной передачи данных.

Чтобы [класс](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) мог работать как bean, он должен соответствовать определённым соглашениям об именах методов, конструкторе и поведении. Эти соглашения дают возможность создания инструментов, которые могут использовать, замещать и соединять JavaBeans.

Правила описания гласят:

* Класс должен иметь [конструктор](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) без параметров, с модификатором доступа public. Такой конструктор позволяет инструментам создать объект без дополнительных сложностей с параметрами.
* Свойства класса должны быть доступны через get, set и другие методы (так называемые [методы доступа](https://ru.wikipedia.org/wiki/Accessor)), которые должны подчиняться стандартному соглашению об именах. Это легко позволяет инструментам автоматически определять и обновлять содержание bean’ов. Многие инструменты даже имеют специализированные редакторы для различных типов свойств.
* Класс должен быть [сериализуем](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F#Java). Это даёт возможность надёжно сохранять, хранить и восстанавливать состояние bean независимым от платформы и виртуальной машины способом.
* Класс должен иметь переопределенные методы equals(), hashCode() и toString().

Так как требования в основном изложены в виде соглашения, а не [интерфейса](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81), некоторые разработчики рассматривают JavaBeans, как [Plain Old Java Objects](https://ru.wikipedia.org/wiki/POJO), которые следуют определённым правилам именования. (POJO — простой Java-объект, не унаследованный от какого-то специфического объекта и не реализующий никаких служебных интерфейсов сверх тех, которые нужны для бизнес-модели.Термин придуманн в пику EJB (Enterprise JavaBeans)). Пример:

public class PersonBean implements java.io.Serializable {

private String name;

private boolean deceased;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public boolean isDeceased() {

return deceased;

}

public void setDeceased(boolean deceased) {

this.deceased = deceased;

}

@Override public boolean **equals**(Object o) {…}

@Override public int **hashCode**() {…}

@Override public String **toString**() { … }

}

1. **Spring** {

Spring MVC — основные понятия, архитектура.

В основе Spring лежит паттерн Inversion of control. Применительно к легковесным контейнерам, основная идея этого паттерна заключается в устранении зависимости компонентов или классов приложения от конкретных реализаций вспомогательных интерфейсов и делегировании полномочий по управлению созданием нужных реализаций IoC контейнеру.

Основные преимущества IoC контейнеров:

* управление зависимостями
* упрощение повторного использования классов или компонентов
* упрощение unit-тестирования
* более "чистый" код (Классы больше не занимаются инициализацией вспомогательных объектов. Не стоит, конечно "перегибать палку", управляя созданием абсолютно всех объектов через IoC. В IoC контейнер лучше всего выносить те интерфейсы, реализация которых может быть изменена в текущем проекте или в будущих проектах.)

**Inversion of Control** (инверсия управления) — это некий абстрактный принцип, набор рекомендаций для написания слабо связанного кода. Суть которого в том, что каждый компонент системы должен быть как можно более изолированным от других, не полагаясь в своей работе на детали конкретной реализации других компонентов.

**Dependency Injection** (внедрение зависимостей) — это одна из реализаций этого принципа (помимо этого есть еще Factory Method, Service Locator).

**Spring MVC**

Фреймворк Spring Web model-view-controller (MVC) или по нашему модель-представление-контроллер построен вокруг DispatcherServlet, который распределяет запросы по обработчикам. В нём настраивается мэппинг запросов, локали, временные зоны и многое другое. Обработчик по умолчанию строится на аннотациях @Controller и @RequestMapping, которые предоставляют широкий набор гибких методов для обработки запросов. После версии Spring 3.0. механизм @Controller так же позволяет создавать RESTful веб сайты и приложения, используя аннотацию @PathVariable и другие возможности.

**DispatcherServlet**

Spring MVC построен вокруг центрального сервлета, который распределяет запросы по контроллерам (Front Controller), а также предоставляет другие широкие возможности при разработке веб приложений. На самом деле DispatcherServlet — полностью интегрированный сервлет в Spring IoC контейнер и таким образом получает доступ ко всем возможностям Spring.

Обработка запросов в DispatcherServlet показана на рисунке ниже.

[](http://javastudy.ru/wp-content/uploads/2015/04/mvc.png)

DispatcherServlet — это обычный сервлет (наследуется от базового класса HttpServlet), и его также необходимо описывать в web.xml вашего веб приложения. Вам необходимо указать мэппинг запросов, которые будут обрабатываться в DispatcherServlet, путем указания URL в web.xml. Ниже показана стандартная конфигурация Java EE необходимая для настройки DispatcherServlet:

|  |  |
| --- | --- |
|  | <web-app>      <servlet>          <servlet-name>example</servlet-name>          <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>          <load-on-startup>1</load-on-startup>      </servlet>      <servlet-mapping>          <servlet-name>example</servlet-name>          <url-pattern>/example/\*</url-pattern>      </servlet-mapping>  </web-app> |

Все запросы, начинающиеся с /example будут обработаны в диспетчере с именем example. Также вы можете настроить его с помощью аннотаций:

|  |  |
| --- | --- |
|  | public class MyWebApplicationInitializer implements WebApplicationInitializer {      @Override      public void onStartup(ServletContext container) {          ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new DispatcherServlet());          registration.setLoadOnStartup(1);          registration.addMapping("/example/\*");      }  } |

Где WebApplicationInitializer — интерфейс, предоставляемый Spring MVC, который гарантирует инициализацию при старте контейнера.

**Жизненный цикл запроса**

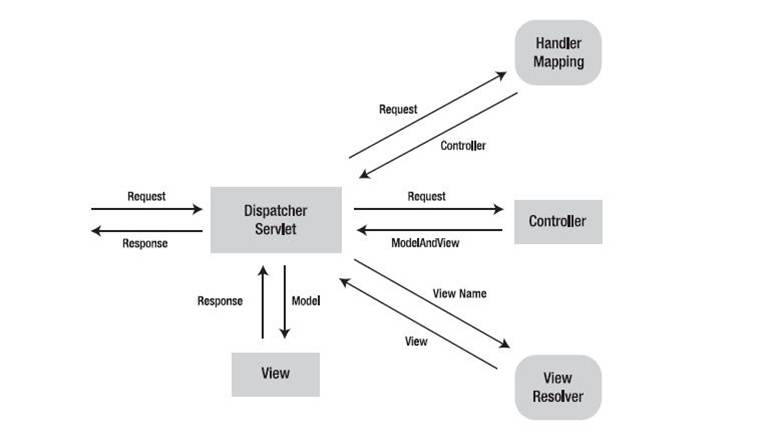
Приведу картинку из интернета, которая показывает жизненный цикл запроса. Возможно по ней кому-то будет легче понять как происходит обработка запросов от начала и до конца. В принципе здесь всё тоже самое, но добавлены некоторые пункты, названия которых говорят сами за себя.

[](http://javastudy.ru/wp-content/uploads/2015/04/RequestLifecycle.png)

1. **DispatcherServlet**

В самом начале входящий запрос попадает в Front controller (для нас он не виден). Далее он перенаправляет его в Controller. Именно за эту часть и отвечает разработчик. После обработки запроса посылается ответ к Front controller, и после этого результат используется для отображения представления. Вот такое легкое описание шаблона проектирования модель-представление-контроллер:). Всё это работает в контейнере сервлетов (на картинке подпись — Tomcat).

Немного более подробно показано на следующей схеме:

[](http://javastudy.ru/wp-content/uploads/2015/04/SpringMVC-detail.jpg)

Что здесь показано?

* Вначале DispatcherServlet (диспетчер сервлетов) получает запрос, далее он смотрит свои настройки, чтобы понять какой контроллер использовать (на рисунке Handler Mapping).
* После получения имени контроллера запрос передается в него (на рисунке Controller). В контроллере происходит обработка запроса и обратно посылается ModelAndView (модель — сами данные; view (представление) — как эти данные отображать).
* DispatcherServlet на основании полученного ModelAndView ищет какое представление ему использовать (View Resolver) и получает в ответе имя представления View
* В представление передаются данные (model) и обратно, если необходимо, посылается ответ от представления.

1. **@Autowired**

Аннотация @Autowired отмечает конструктор, поле или метод как требующий автозаполнения инъекцией зависимости Spring. Чтобы аннотация @Autowire присвоила переменной значения соответствующего bean'а, необходимо чтобы этот bean либо был объявлен в xml конфигурации приложения, либо существовал класс с соответствующей иньекцией управления.

1. **@Service / @Component**

Аннотации @Service и @Component взаимозаменяемы. Обе аннотации говорят спрингу, что класс, над которыми они проставлены является бином, то есть кандидатом на автоматическое обнаружение если в конфигурационном файле проставлен тег context:annotation-config и тег автоматического сканирования context:component-scan или только тег context:component-scan. Отличие между ними только в идеи использования аннотированного класса. Аннотацию @Service лучше применять к бину, который предоставляет службу другим бинам. Аннотация @Component лучше всего подходит для бинов-свойств, то есть тех бинов которые предоставляют значения свойств другим бинам, а не выполняют код, результаты которого используют другие бины.

1. **Dependency injection**

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Класс X зависит от класса Y, если выполняется одно из следующих условий:

\* X has-a Y и вызывает его

\* X is-a Y

\* X зависит от некоторого класса Z, который зависит от Y (принцип транзитивности)

X зависит от Y не значит, что Y зависит от X. Если же существуют обе зависимости, то это называется циклической зависимостью: X не может быть использован без Y, и наоборот. Существование большого числа циклических зависимостей в объектно-ориентированной программе может быть показателем не оптимального программного построения.

Если объект x (класса X) вызывает методы объекта y (класса Y), то X зависит от Y. Зависимость может быть обращена введением третьего класса, а именно интерфейсного класса I, который должен содержать все методы, которые x может вызвать у y. Кроме того, Y должен реализовать интерфейс I. X и Y сейчас оба зависят от I, и класс X более не зависит от класса Y; предполагается, что X не реализует Y.

Это исключение зависимости класса X от Y введением интерфейса I называется Inversion of Control (или Dependency Injection).

Следует отметить, что Y может зависеть от других классов. До внесения изменений X зависел от Y, таким образом X косвенно зависел от всех классов, от которых зависит Y. Применением Inversion of Сontrol все эти косвенные зависимости также были разорваны — не только зависимость X от Y. Новый интерфейс I ни от чего не зависит.

1. **Бины и их скоупы**

В Spring-е бином (bean) называют любой класс, который управляется контейнером Spring. То есть такими вещами, как создание экземпляра бина, его инициализация, внедрение зависимостей и параметров, деинициализация, генерация всевозможных оберток над бином, занимается не ваш код, а IoC-контейнер Spring-а.

|  |  |
| --- | --- |
| Scope | Description |
| singleton | This scopes the bean definition to a single instance per Spring IoC container (default). |
| prototype | This scopes a single bean definition to have any number of object instances. |
| request | This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |
| global-session | This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext. |

**Singleton**

По умолчанию все создаваемые управляемые компоненты создаются в единственном экземпляре, то есть являются синглтонами и все зависимые компоненты от него компоненты используют этот экземпляр, как это показано на рисунке 2. В этом примере создается один единственный экземпляр компонента с идентификатором service и использующие его компоненты Service Client 1, Service Client 2 и Service Client 3 используют этот экземпляр

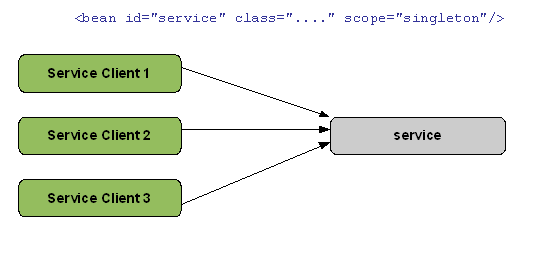


Рисунок 2: Режим Singleton

Этот режим можно также задать в конфигурационном файле в явном виде, для этого существует два способа: либо определяющий бин xml элемент содержит атрибут singleton значение которого равно true, либо атрибут scope, значение которого задано как singleton. То есть определение управляемого компонента

<bean scope="singleton" class="com.company.scopedemo.BeanClass" id="singletonBean">

эквивалентно определению

<bean id="singletonBean" class="com.company.scopedemo.BeanClass" singleton ="true"/>

Хотя как уже сказано, атрибуты scope (singleton) в этом случае являются избыточными, поскольку этот режим инициализации используется по умолчанию.

Начиная с версии Spring 3 не существует возможность объявлять режим инициализации с помощью xml атрибута singleton.

**Prototype**

Если у управляемого компонента (bean) указан режим prototype, то для всех зависимых от него компонент создается новый экземпляр этого управляемого компонента, как это продемонстрировано на рисунке 3:

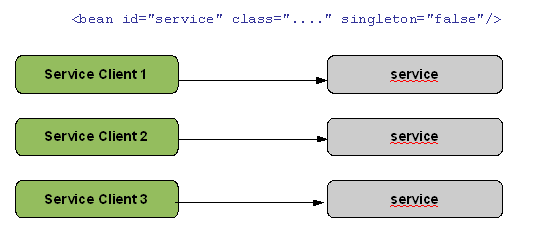


Рисунок 3: Режим Prototype

Этот режим задается в конфигурационном файле либо определением атрибута «singleton» в определении компонента со значением false, либо определением атрибута «scope» со значением «prototype»:

<bean id="prototypeBean" class="com.company.scopedemo.BeanClass" scope="prototype"/>

или

<bean id="prototypeBean" class="com.company.scopedemo.BeanClass" singleton="false"/>

Начиная с версии Spring 3 не существует возможность объявлять режим инициализации с помощью xml атрибута singleton.

**Режимы session, request и global-session**

Эти режимы используются в Spring Framework в web приложениях и определяют, что управляемый компонент создается в единственном экземпляре во время либо запроса (режим request), либо пользовательской HTTP сессии (режим session), либо глобальной HTTP сессии (режим global session), которая определена для портлетов (portlet). В отличие от режимов prototype и singleton, для конфигурации которых можно использовать два различных аттрибута, эти режимы определяются в конфигурационном файле только атрибутом scope, например:

<bean id="prototypeBean" class="com.company.scopedemo.BeanClass" scope="globalSession"/>

или

<bean id="prototypeBean" class="com.company.scopedemo.BeanClass" scope="request"/>

Spring Framework создает управляемые компоненты с этими режимами только внутри web контекста, например XmlWebApplicationContext. Попытка создать бин с одним из этих режимов внутри обычного контейнера (например XmlBeanFactory) приведет к генерации исключения IllegalStateException.

1. **Rowmapper**

We can use RowMapper interface to fetch the records from the database using query() method of JdbcTemplate class. In the execute of we need to pass the instance of RowMapper now.

*Syntax of query method using RowMapper*

public T query(String sql,RowMapper<T> rm)

*RowMapper Interface*

RowMapper interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

*Advantage of RowMapper over ResultSetExtractor*

RowMapper saves a lot of code becuase it internally adds the data of ResultSet into the collection.

*Method of RowMapper interface*

It defines only one method mapRow that accepts ResultSet instance and int as the parameter list. Syntax of the method is given below:

public T mapRow(ResultSet rs, int rowNumber)throws SQLException

Example

public List<Employee> getAllEmployeesRowMapper(){

return template.query("select \* from employee",new RowMapper<Employee>(){

@Override

public Employee mapRow(ResultSet rs, int rownumber) throws SQLException {

Employee e=new Employee();

e.setId(rs.getInt(1));

e.setName(rs.getString(2));

e.setSalary(rs.getInt(3));

return e;

}

});

}

1. **Spring security**

Spring Security это Java/JavaEE framework, предоставляющий механизмы построения систем аутентификации и авторизации, а также другие возможности обеспечения безопасности для корпоративных приложений, созданных с помощью Spring Framework.

Spring Security пример.htm

}

1. **Сервлет**

Сервлет является интерфейсом Java, реализация которого расширяет функциональные возможности сервера. Сервлет взаимодействует с клиентами посредством принципа запрос-ответ.

Жизненный цикл сервлета состоит из следующих шагов:

1. В случае отсутствия сервлета в контейнере.
   1. Класс сервлета загружается контейнером.
   2. Контейнер создает экземпляр класса сервлета.
   3. Контейнер вызывает метод init(). Этот метод инициализирует сервлет и вызывается в первую очередь, до того, как сервлет сможет обслуживать запросы. За весь жизненный цикл метод init() вызывается только один раз.
2. Обслуживание клиентского запроса. Каждый запрос обрабатывается в своем отдельном потоке. Контейнер вызывает метод service() для каждого запроса. Этот метод определяет тип пришедшего запроса и распределяет его в соответствующий этому типу метод для обработки запроса. Разработчик сервлета должен предоставить реализацию для этих методов. Если поступил запрос, метод для которого не реализован, вызывается метод родительского класса и обычно завершается возвращением ошибки инициатору запроса.
3. В случае если контейнеру необходимо удалить сервлет, он вызывает метод destroy(), который снимает сервлет из эксплуатации. Подобно методу init(), этот метод тоже вызывается единожды за весь цикл сервлета.
4. **REST**

RepresentationalStateTransfer – это стиль архитектуры программного обеспечения для распределенных систем, таких как World Wide Web, который, как правило, используется для построения веб-служб.

Реализация REST позволяет получать и манипулировать данными по средствам HTTP запросов.

(API = Application Programming Interface).

1. **GET POST PUT UPDATE запросы, когда используются.**

HTTP методы:

* GET для получения и поиска данных
* POST для добавления данных
* PUT для обновления данных
* DELETE для удаления данных

1. **JSON**

Формат JSON используется для представления объектов в виде строки.

Данные в формате JSON представляют собой:

* JavaScript-объекты { ... } или
* Массивы [ ... ] или
* Значения одного из типов:
  + строки в двойных кавычках,
  + число,
  + логическое значение true/false,
  + null.

Почти все языки программирования имеют библиотеки для преобразования объектов в формат JSON.

Основные методы для работы с JSON в JavaScript – это:

* JSON.parse – читает объекты из строки в формате JSON.

Вызов JSON.parse(str) превратит строку с данными в формате JSON в JavaScript-объект/массив/значение.

Например:

var numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1

Или так:

var user = '{ "name": "Вася", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

user = JSON.parse(user);

alert( user.friends[1] ); // 1

Данные могут быть сколь угодно сложными, объекты и массивы могут включать в себя другие объекты и массивы. Главное чтобы они соответствовали формату.

* JSON.stringify – превращает объекты в строку в формате JSON, используется, когда нужно из JavaScript передать данные по сети.

Метод JSON.stringify(value, replacer, space) преобразует («сериализует») значение в JSON-строку.

Пример использования:

var event = {

title: "Конференция",

date: "сегодня"

};

var str = JSON.stringify(event);

alert( str ); // {"title":"Конференция","date":"сегодня"}

// Обратное преобразование.

event = JSON.parse(str);

При сериализации объекта вызывается его метод toJSON.

Если такого метода нет – перечисляются его свойства, кроме функций.

var room = {

number: 23,

toJSON: function() {

return this.number;

}};

1. **Inner, Outers (Left, Right) Join. Cross Join. (NOT) IN.**

* INNER JOIN

SELECT a.id AS a\_id, b.id AS b\_id

FROM a INNER JOIN b ON a.id = b.id

Запрос вернет объединенные данные, которые *пересекаются* по условию, указанному в ON. Выбираются только совпадающие данные из объединяемых таблиц.

* LEFT JOIN

Чтобы получить данные, которые подходят по условию частично, необходимо использовать внешнее объединение - OUTER JOIN. Такое объединение вернет данные из обеих таблиц (совпадающие по условию объединения) ПЛЮС дополнит выборку оставшимися данными из внешней таблицы, которые по условию не подходят, заполнив недостающие данные значением NULL.

LEFT - указывает что "внешней" таблицей будет находящаяся слева.

* RIGHT JOIN

RIGHT - указывает что "внешней" таблицей будет находящаяся спарва.

* CROSS / FULL JOIN

FULL JOIN возвращает `объединение` объединений LEFT и RIGHT таблиц, комбинируя результат двух запросов.

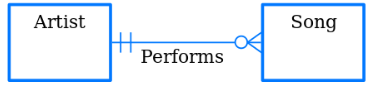
CROSS JOIN возвращает перекрестное (декартово) объединение двух таблиц. Результатом будет выборка всех записей первой таблицы объединенная с каждой строкой второй таблицы. Важным моментом является то, что для кросса не нужно указывать условие объединения.

* Результат вложенного запроса, в котором присутствует ключевое слово IN (или NOT IN) — это список из нуля или более значений. После того как вложенный запрос вернул результат, он используется внешним запросом.

+ Объяснение SQL объединений JOIN- LEFT-RIGHT-INNER-OUTER.htm

1. **ER-model**

ER -диаграммы используют графическое изображение сущностей предметной области, их свойств (атрибутов), и взаимосвязей между сущностями.



1. **Операции ON DELETE и ON UPDATE**

Выражения ON DELETE и ON UPDATE внешних ключей используются для указания действий, которые будут выполняться при удалении строк родительской таблицы (ON DELETE) или изменении родительского ключа (ON UPDATE). Один и тот же внешний ключ может иметь различные действия, указанные для ON DELETE и ON UPDATE. Действия внешних ключей во многом похожи на триггеры.

В базе данных SQLite действия ON DELETE и ON UPDATE, ассоциированные с внешним ключом, могут быть следующими: NO ACTION, RESTRICT, SET NULL, SET DEFAULT или CASCADE. Если действие не указывается специально, оно по умолчанию является NO ACTION.

* NO ACTION: опция «NO ACTION» означает, что когда родительский ключ изменяется или удаляется из базы данных, никаких специальных действий не производится.
* RESTRICT: действие «RESTRICT» заключается в том, что приложению запрещается удалять (для ON DELETE RESTRICT) или изменять (для ON UPDATE RESTRICT) родительский ключ, когда существует один или несколько ссылающихся на него дочерних ключей. Различие между эффектом от действия RESTRICT и обычным поведением внешнего ключа заключается в том, что действие RESTRICT запускается тогда, когда изменяется поле записи, а не тогда, когда завершается текущий запрос, как в случае немедленного ограничения, и не тогда, когда завершается текущая транзакция, как в случае отложенного ограничения. Даже если внешний ключ был заявлен как отложенный, если его действием был объявлен RESTRICT, то SQLite будет выдавать сообщение об ошибке немедленно, как только удаляется или изменяется родительский ключ, от которого зависят дочерние ключи.
* SET NULL: если действие сконфигурировано как «SET NULL», то при удалении родительского ключа (для ON DELETE SET NULL) или его изменении (для ON UPDATE SET NULL) столбцы дочернего ключа будут устанавливаться в значение NULL во всех строках дочерней таблицы, которые ссылаются на удаляемую/изменяемую строку родительской таблицы.
* SET DEFAULT: действие «SET DEFAULT» похоже на SET NULL за тем исключением, что значение каждого столбца дочернего ключа устанавливается не в NULL а в значение по умолчанию для данного столбца. О том, как столбцам назначаются значения по умолчанию, можно узнать в подробной документации по CREATE TABLE.
* CASCADE: действие «CASCADE» распространяет операции удаления и изменения родительского ключа на зависящие от него дочерние ключи. Для действия ON DELETE CASCADE это выражается в том, что каждая строка дочерней таблицы, которая ассоциирована с удаляемой родительской строкой, также будет удалена. Для действия ON UPDATE CASCADE это выражается в том, что значения, сохранённые в зависящем дочернем ключе, будут заменены на новые значения родительского ключа.

1. **DAO / DTO**

Data Access Object (DAO) — это объект, который предоставляет абстрактный интерфейс к какому-либо типу базы данных или механизму хранения. Определённые возможности предоставляются независимо от того, какой механизм хранения используется и без необходимости специальным образом соответствовать этому механизму хранения. Этот шаблон проектирования применим ко множеству языков программирования, большинству программного обеспечения, нуждающемуся в хранении информации и к большей части баз данных, но традиционно этот шаблон связывают с приложениями на платформе Java Enterprise Edition, взаимодействующими с реляционными базами данных через интерфейс JDBC, потому что он появился в рекомендациях от фирмы Sun Microsystems.

Data Transfer Object (DTO) — один из шаблонов проектирования, используется для передачи данных между подсистемами приложения. Data Transfer Object, в отличие от business object или data access object не должен содержать какого-либо поведения. В Enterprise JavaBeans DTO используется для сериализации.

Entity beans представляют объекты, находящиеся в постоянном хранилище, например, в базе данных. С одной стороны, это очень удобно, так как программа-клиент не должна заботиться о подсоединении к базе данных напрямую. С другой стороны, каждое изменение в entity bean может вызывать методы удалённого доступа, что увеличивает нагрузку на сеть и снижает скорость работы программы. Sun Java Center порекомендовал для решения этой проблемы изолировать все данные в отдельный объект и передавать этот объект в entity bean одним методом. В версии EJB 3.0 модель записи данных была изменена, эта проблема была разрешена и нужда в DTO отпала.

Не следует путать DTO с Value Object - это разные паттерны. Путаница вызвана тем, что в ранней литературе по J2EE для описания DTO использовался термин "Value Object", однако позже его заменили на "Transfer Object".

1. **Bcrypt / salt**

bcrypt — адаптивная криптографическая функция формирования ключа, используемая для защищенного хранения паролей.

Соль (также модификатор) — строка данных, которая передаётся хеш-функции вместе с паролем. Используется для удлинения строки пароля, чтобы увеличить сложность взлома. Например, вы хешируете и храните свои пароли в [MD5](https://ru.wikipedia.org/wiki/MD5). Если ваша база будет украдена — злоумышленник довольно просто восстановит большинство исходных паролей, используя заранее подготовленные [радужные таблицы](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B4%D1%83%D0%B6%D0%BD%D0%B0%D1%8F_%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0). Если же мы «посолим» пароль, то есть соединим строчку из 10—20 случайных символов с паролем и уже от этой строчки найдем MD5, — стандартные таблицы не будут работать, так как они не рассчитаны на поиск такой длинной строки.

1. End2end шифрование
2. **Maven**, Gradle, Ant.

Apache Maven — фреймворк для автоматизации сборки проектов на основе описания их структуры. В большинстве случаев апгрейд библиотеки сводится к изменению номера версии в pom.xml. Вся структура проекта описывается в файле pom.xml (POM – Project Object Model), который должен находиться в корневой папке проекта.

Aртефакт — это, по сути, любая библиотека, хранящаяся в репозитории. Это может быть какая-то зависимость или плагин.

Зависимости — это те библиотеки, которые непосредственно используются в вашем проекте для компиляции кода или его тестирования.

Плагины же используются самим Maven'ом при сборке проекта или для каких-то других целей (деплоймент, создание файлов проекта для Eclipse и др.).

Архетип — это некая стандартная компоновка файлов и каталогов в проектах различного рода (веб, swing-проекты и прочие). Другими словами, Maven знает, как обычно строятся проекты и в соответствии с архетипом создает структуру каталогов.

1. Jenkinson
2. **Bigserial** [**http://postgresql.ru.net/manual/datatype-numeric.html**](http://postgresql.ru.net/manual/datatype-numeric.html)

Числовой тип с автоувеличением в БД. Должен использоваться если вы ожидаете. совместное использование более чем 231 индентификаторов в одной таблице. (8 байт, 1 – 2^63).

1. Java Code Conventions
2. **Модель OSI**

OSI.docx

1. TCP/IP, порты.
2. Java 8. (Методы в интерфейсах, Лямда-выражения).
3. **Компиляторы и интерпретаторы.**

И компиляторы и интерпретаторы преобразуют исходный код в машинный код, только разными путями. **Интерпретатор читает исходный код программы и выполняет его.** Преобразование исходного кода в бинарный и выполенение выполняется построчно.  
 исходный код программы -> интерпретатор -> ОС -> результат  
 **Компиляторы же, полностью переобразовывают исходный код программы в бинарный (а не построчно, как в случае с интрепретаторами), который ОС может выполнять самостоятельно.** То есть, для запуска программы иметь компилятор нет необходимости.  
 исходный код программы -> компилятор -> *объектный код* -> ОС -> результат

1. **Синтаксический сахар.**

это синтаксические возможности, применение которых не влияет на поведение программы, но делает использование языка более удобным для человека.

В противоположность понятие «синтаксическая соль» обозначает дополнительные технически бесполезные конструкции в языке программирования, которые правила языка требуют употреблять при выполнении потенциально небезопасных действий. Чтобы, используя их, программист подтверждал, что сомнительное действие предпринято им сознательно.