

OOP

1. Что такое ООП?

Методология, парадигма программирования, основной концепцией является понятие объекта (объект отождествляется с предметной областью).

Программа представлена, как совокупность объектов, каждый из которых является **экземпляром** определенного **класса**, а классы образуют **иерархию наследования**.

КЛАСС – это описание ещё не созданного объекта, общий шаблон.

Шаблон состоит из:

- полей (имя, возраст для чел. и т.д.). Состояние/ряд меняющихся свойств.
- конструктора (первоначально инициализирует объект, заполняет нужные поля)
- методов (что умеет делать объект). Действия, функции конкретного объекта.

ОБЪЕКТ – это экземпляр класса, созданный по шаблону (выше) с собственным состоянием свойств.

Три составляющих каждого объекта:

- идентичность (identity) – то, что отличает один объект класса от другого (equals)
- состояние (state) - набор всех полей объекта и их значений
- поведение (behaviour) - набор всех методов класса объекта

Программа считается ООП, только если выполнены ВСЕ ТРИ условия:

1. основные логические элементы программы – это **объекты** (а не алгоритмы)
2. каждый объект – это **экземпляр класса**
3. классы образуют **иерархии**

2. Какие преимущества у ООП?

Делим программу на «модули»-классы, **ОБЪЕКТЫ** каждый из которых делает свою часть работы.

Код можно повторно использовать в любом месте программы, это экономит время (не нужно писать однотипные функции для разных сущностей).

«Более естественная» декомпозиция ПО существенно облегчает его разработку (код легко читается и быстро пишется).

Возможность создавать расширяемые системы (extensible systems), именно это отличает ООП от традиционных методов программирования.

3. Какие недостатки у ООП?

Снижение производительности и увеличение потребления памяти по сравнению с другими (процедурными) языками.

Справка: Большинство ранних императивных языков программирования - **процедурные**, в том числе **Фортран**, **Кобол**, **Алгол**, **Бейсик**, **Си**, **Паскаль**, **Форт**. Более поздние императивные языки, в частности, реализующие ООП парадигму C++, Java, как правило, **НЕ относят** к категории процедурных, поскольку принцип организации блоков выполнения подпрограммы в них реализуется на другом уровне **абстракции**.

Почему так? ООП требует распределения информации по множеству мелких инкапсулированных объектов, количество ссылок на эти объекты (для быстрого доступа к ним) растёт взрывными темпами, а значит падает производительность.

4. Назовите основные принципы ООП.

Объектно-ориентированное программирование обладает рядом принципов (**ПАНИ**)

- Полиморфизм
- Абстракция
- Наследование
- Инкапсуляция

5. Что такое инкапсуляция? (с примером)

Инкапсулируем, т.е. прячем данные и методы объекта в классе, скрываем детали реализации от пользователя.

Открываем только то, что нужно при последующем использовании (нужен ордер, а как реализован – скрыто).

Использование инкапсуляции гарантирует, что данные не будут искажены или изменены не надлежащим образом.

```
Order order = getOrder();
order.paymentReceived();
order.sendToDelivery();
order.cancel();
```

Очевидные примеры инкапсуляции:

- это модификаторы доступа (private, дефолт, protected, public)
- это класс, внутри которого описываем детали, используем далее везде
- на более высоком уровне, чем класс – это доступ на уровне пакета
- это также геттеры-сеттеры

Пример: если поле age у класса User не инкапсулировать, любой сможет написать: `User.age = -1000; (минус тысяча)` Механизм инкапсуляции позволяет защитить поле age при помощи метода-сеттера, в который можно поместить проверку (возраст не может быть отрицательным числом).

6. Что такое наследование? (с примером)

Наследование — это возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (суперкласса), добавляя при необходимости новые свойства и методы.

Наследование позволяет реализовывать сложные схемы с четкой иерархией «от общего к частному». Это облегчает понимание и масштабирование кода.

Зачем? Чтобы повторно использовать код.
Расширять функционал уже имеющихся классов за счет добавления нового функционала или изменения старого.

Пример: метод intValue() определён в классе BigDecimal, метод shortValue() объявлен в его родительском классе Number. Имея на руках экземпляр класса BigDecimal bd, мы можем вызывать на нём как intValue(), так и shortValue().

```
public final class BigDecimal extends Number {
    public int intValue() {
        // ...
    }
    // no shortValue() method,
    // it's inherited from Number
}
bd.intValue() bd.shortValue()
```

Работа с методами будет одинакова вне зависимости от уровня в иерархии наследования.

Наследование может быть одиночным и множественным (наследование одного класса сразу от нескольких других). **Множественное наследование запрещено в Java!**

В качестве альтернативы абстрактному методу (где все методы публичные и абстрактные), в Java **введена отдельная сущность – интерфейс** (по определению методы публичные и абстрактные, поэтому явно писать данные модификаторы не нужно).

Например, имеется следующий класс Person, описывающий отдельного человека:

```

1 class Person {
2     String name;
3     public String getName(){ return name; }
4     public Person(String name){
5         this.name=name;
6     }
7 }
8
9     public void display(){
10        System.out.println("Name: " + name);
11    }
12 }
```



Например, класс **Person**, человек.

И поскольку сотрудник – это также человек, то рационально сделать класс **Employee** производным (наследником, подклассом) от класса Person, который, в свою очередь, называется базовым классом, родителем или суперклассом.

Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника **ключевое слово extends**, после которого идет имя базового класса.

Если в базовом классе определены конструкторы, то в конструкторе производного класса нужно вызвать один из конструкторов базового класса с помощью ключевого слова **super**.

Справка: Указатель super - представляет экземпляр суперкласса.

В подклассе можем напрямую обращаться к свойствам и методам, унаследованным от родительского класса, но, если свойства или методы подкласса имеют одинаковое имя (затенение атрибутов, переопределение метода), их необходимо различать, прежде чем к ним можно будет получить доступ.

Между атрибутами с одним и тем же именем родительского и дочернего классов не существует переопределяющей связи, и одновременно существуют два пробела (дочерние классы охватывают свойства родительского класса), и для доступа необходимо использовать разные префиксы.

```

class A{
    int value = 10;
}
class B extends A{
    int value = 20;
    public void print(){
        int value = 30;
        System.out.println(value);      //30
        System.out.println(this.value); //20
        System.out.println(super.value); //10
    }
}
```

7. Что такое полиморфизм? (с примером)

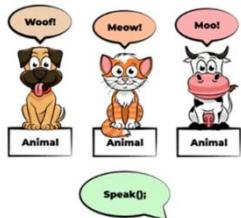
Полиморфизм — это возможность использовать **классы-потомки** в контексте, который был предназначен для класса-предка. Возможность работать с несколькими объектами так, будто это один и тот же объект. При этом поведение объектов будет разным в зависимости от типа объекта, к которому они принадлежат.

Однаковые методы разных объектов могут выполнять задачи разными способами.

Например, есть метод «получить цену доставки» для объекта «клиент». Для разных типов клиентов цена и условия доставки могут быть разными, хотя название метода при этом не меняется.

```

Client client = new VIPClient(); +
client.getDeliveryPrice(); +
// actually VIPClient.getDeliveryPrice()
// is executed
```



Таким образом, конкретный код, выполняемый при вызове метода объекта, **определяется КЛАССОМ объекта**, находящимся по ссылке в момент исполнения программы.

На рисунке слева разные животные (наследники Animal) говорят по-разному, хотя используют один метод **speak()**. Собака гавкает, кот мяукает, корова мычит.

Пример из задачи 1.1.4. → (используем разную реализацию в зависимости от класса в DAO слое).

catch — полиморфная конструкция, т.е. catch по типу Parent перехватывает лежащие экземпляры любого типа, который является Parent-ом (т.е. экземпляры непосредственно Parent-а или любого потомка Parent-а)

// Service на этот раз использует реализацию DAO через Hibernate (ТЭ)

// Переключаем переменную userDao

```
public class UserServiceImpl implements UserService {  
  
    // private final UserDao userDao = new UserDaoJDBCImpl(); // JDBC solution  
    private final UserDao userDao = new UserDaoHibernateImpl(); // Hibernate solution
```

Мы постоянно используем полиморфизм в базовых классах Java. Один очень простой пример — создание экземпляра класса `ArrayList` с объявлением типа как интерфейс `List`.

```
List<String> list = new ArrayList<>();
```

Полиморфизм бывает динамическим (переопределение) и **статическим** (перегрузка).

<https://www.youtube.com/watch?v=ynmxzCRiCU>

Переопределение метода (@Override) позволяет взять какой-то метод родительского класса и написать в каждом классе-наследнике свою **специфическую реализацию** этого метода. Новая реализация «заменит» родительскую в дочернем классе.

Пример: чтобы избежать создания кучи методов для подачи голоса (`voiceCat()` для мяуканья, `voiceSnake()` для шипения и т.д.), мы хотим, чтобы при вызове метода `voice()` змея шипела, кошка мяукала, а собака лаяла.

```
public class Cat extends Animal {  
    @Override  
    public void voice() {  
        System.out.println("Мяу!");  
    }  
}  
  
public class Dog extends Animal {  
    @Override  
    public void voice() {  
        System.out.println("Гав!");  
    }  
}  
  
public class Snake extends Animal {  
    @Override  
    public void voice() {  
        System.out.println("Ш-ш-ш!");  
    }  
}
```

То есть в классе-наследнике можно завести метод с таким же именем и параметрами, как и в базовом классе. При этом **тип возвращаемого значения** должен либо совпадать, либо быть подклассом (потомком) типа, возвращаемого базовым методом. А также **модификатор доступа** должен быть либо тем же, либо более открытым. Если эти условия выполнены, то метод класса-наследника заменит собой одноимённый метод из базового класса.

Статические методы нельзя переопределить, можно только перегрузить, т.к. статические поля и методы относятся к классу, а не экземпляру. К ним обращаемся через имя класса.

Перегрузка методов: Java разрешает определение внутри одного класса двух или более методов с одним именем, если только объявления их параметров различны. В этом случае методы называют перегруженными, а процесс — перегрузкой методов.

За счёт этого можно, например, эмулировать значение по умолчанию для параметров.

```
void foo(Number n) { System.out.print("N"); }
void foo(Integer i) { System.out.print("I"); }
void test() {
    Integer i = 42;
    Number n = i;
    foo(n); foo(i); // Напечатает "NI"
    System.out.println(null); // Ошибка компиляции
}
```

```
public final class String {
    → public int indexOf(int ch) {
        return indexOf(ch, 0);
    }

    → public int indexOf(int ch, int fromIndex) {
        // ...
    }
}
```

Справка: Конструкторы (так же, как и методы) могут быть перегружены. Бывают дефолтный (не принимающий аргументы) и параметризованный. Это определяется во время создания объекта. Есть также конструктор копирования см. дальше).

Полиморфными бывают:

- **Переменная** (может принимать значения разных типов)
- **Метод** (если хотя бы один аргумент является полиморфной переменной)

Выделяют два вида полиморфных методов/функций:

- **ad hoc, метод/функция ведет себя по-разному** для разных типов аргументов (например, метод draw() — рисует по-разному фигуры разных типов);
- **параметрическая функция ведет себя одинаково** для аргументов разных типов (например, метод add() — одинаково кладет в контейнер элементы разных типов).

7.1. Что такое абстракция (с примерами)?

Абстракция означает выделение главных, наиболее значимых характеристик предмета и наоборот — отbrasывание второстепенных, незначительных.

Например, телефон должен звонить (это минимум), год выпуска — объединяющая х-ка. Конструктор на год выпуска, методы звонить **call()** и принимать звонки **ring()**.

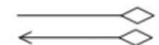
Поэтому для класса Employee мы зададим переменные String name, int age, int socialInsuranceNumber и int taxNumber. А от лишней для нас информации вроде цвета глаз откажемся, абстрагируемся.

А вот если мы создаем картотеку фотомоделей для агентства, ситуация резко меняется.

Для описания фотомодели нам как раз очень важны рост, цвет глаз и цвет волос, а номер ИНН не нужен.

Поэтому в классе Model мы создаем переменные String height, String hair, String eyes.

8. Что такое ассоциация, агрегация и композиция?

-  — агрегация (aggregation) — описывает связь «часть»—«целое», в котором «часть» может существовать отдельно от «целого». Ромб указывается со стороны «целого».
-  — композиция (composition) — подвид агрегации, в которой «части» не могут существовать отдельно от «целого».
-  — зависимость (dependency) — изменение в одной сущности (независимой) может влиять на состояние или поведение другой сущности (зависимой). Со стороны стрелки указывается независимая сущность.
-  — обобщение (generalization) — отношение наследования или реализации интерфейса. Со стороны стрелки находится суперкласс или интерфейс.

Теория ООП выделяет **отношения (связи) между классами:**

1. **Наследование «is a»** ЯВЛЯТЬСЯ (обобщение/расширение) – очень мощная связь
2. **Ассоциация «has a»** ИМЕТЬ (объекты ИМЕЮТ ссылки/ссылаются друг на друга)
 - агрегация (принадлежность как студент к кафедре/нескольким кафедрам)
 - композиция (жёсткое отношение, как один лист к одной книге)

Ассоциация означает, что объекты двух классов могут ссылаться один на другой, иметь связь друг с другом. Один класс включает в себя другой класс в качестве одного из полей.

Например, Оператор управляет Роботом. Соответственно возникает ассоциация между Роботом и Оператором. Ассоциация и есть описание связи между двумя объектами. Идея достаточно простая — два объекта могут быть связаны между собой и это надо как-то описать.

Реализация: В одном классе делается ссылка на другой и наоборот (не всегда). Дальше идет развитие данной идеи в зависимости от количества связей. Соединим Робота с Оператором, который им управляет. Между ними можно установить ассоциацию через ссылки в одном классе на другой класс. Класс Robot имеет ссылку на класс Operator и наоборот.

```
public class Robot {  
    private double x = 0;  
    private double y = 0;  
    protected double course = 0;  
    // Робот управляется оператором  
    private Operator operator;  
  
    public Robot(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    // Можно узнать какой оператор управляет роботом  
    public Operator getOperator() {  
        return operator;  
    }  
  
    // Можно установить оператора для робота  
    public void setOperator(Operator operator) {  
        this.operator = operator;  
    }  
  
    public class Operator {  
        private String firstName;  
        private String lastName;  
        // Оператор управляет конкретным роботом  
        private Robot robot;  
  
        public Operator(String firstName, String lastName) {  
            this.firstName = firstName;  
            this.lastName = lastName;  
        }  
  
        public String getFirstName() {  
            return firstName;  
        }  
  
        public String getLastName() {  
            return lastName;  
        }  
        // У оператора можно спросить каким роботом он управляет  
        public Robot getRobot() {  
            return robot;  
        }  
        // Оператору можно поручить управлять роботом  
        public void setRobot(Robot robot) {  
            this.robot = robot;  
        }  
    }
```

Агрегация и композиция являются частными случаями ассоциации. Это более конкретизированные отношения между объектами.

Композиция — это более жёсткое отношение, когда объект не только является частью другого объекта, но и вообще не может принадлежать кому-то.
Страница принадлежит только одной Книге.

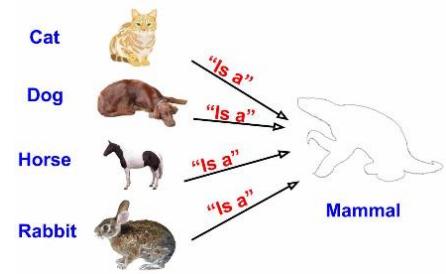
Агрегация — отношение, когда один объект является частью другого. Двигатель поставили в одну машину, потом в другую.

```
public class Main {  
    public static void main(String[] args) {  
        Car car = new Car(new Engine());  
  
        Engine engine = new Engine();  
        Car car1 = new Car(engine);  
        Car car2 = new Car(engine);  
    }  
}
```

Как запомнить? Тест:

- «Является» `is a` подразумевает наследование
- «Имеет» `has a` подразумевает ассоциацию (агрегацию или композицию).

Например, вложенный класс – это композиция.
А вложенный статический класс – это агрегация.



9. Расскажите про раннее и позднее связывание.

Связывание означает **наличие связи между ссылкой и кодом**.

Например, переменная, на которую вы ссылаетесь, привязана к коду, в котором она определена. Аналогично, вызываемый метод привязан к месту в коде, где он определен.

Существует два типа связывания методов в языке Java:

- раннее связывание, его ещё называют **статическим**
- позднее связывание, соответственно, **динамическое**

РАННЕЕ СТАТИЧЕСКОЕ СВЯЗЫВАНИЕ

Если метод известен компилятору, то происходит раннее связывание на этапе компиляции (ранняя стадия жизненного цикла программы – early binding).

Применяется для дженериков, перегруженных, private, final и static методов. В этом случае используются не конкретные объекты, а информация о типе, то есть ссылка. Статическое раннее связывание – используется тип ссылочной переменной.

ПОЗДНЕЕ ДИНАМИЧЕСКОЕ СВЯЗЫВАНИЕ

Динамическое, позднее связывание происходит во **время выполнения программы JVM**. То, какой метод вызвать, определяется конкретным объектом, так что в момент компиляции информация недоступна, т.к. объекты создаются во время выполнения (на поздней стадии жизненного цикла программы – late binding).

Динамическое позднее связывание используется для абстрактных и переопределённых методов. При динамическом связывании для нахождения нужного метода используется конкретный объект.

10. SOLID

SOLID – это принципы разработки программного обеспечения, следуя которым получаем хороший код, который в дальнейшем будет хорошо масштабироваться и поддерживаться в рабочем состоянии.

S Single Responsibility Principle – принцип единственной ответственности. Каждый класс должен иметь только одну зону ответственности.

O Open closed Principle – принцип открытости-закрытости. Классы должны быть открыты для расширения, но закрыты для изменения.

L Liskov substitution Principle – принцип подстановки Барбары Лисков. Должна быть возможность вместо базового (родительского) типа класса подставить любой его подтип (класс-наследник), при этом работа программы не должна измениться. Как запомнить? Поведение потомка не должно противоречить поведению родителя.

I Interface Segregation Principle – это принцип разделения интерфейсов.

Данный принцип обозначает, что не нужно заставлять клиента (класс) реализовывать интерфейс, который не имеет к нему отношения.

D Dependency Inversion Principle – это принцип инверсии зависимостей.

Абстракции НЕ должны зависеть от деталей. Детали должны зависеть от абстракций. Модули верхнего уровня НЕ должны зависеть от модулей нижнего уровня, НО должны зависеть от абстракции. Геометрическая фигура – абстракция. Круг, квадрат – детали.

Использование: Создание интерфейсов и их реализаций. Вычислить площадь геом. фигуры (метод).
Пример: терминал оплаты (абстракция) и разные карты оплаты (детали). Операции с деньгами.

11. Какие еще принципы можешь назвать?

Принцип	Суть принципа
DRY / Don't Repeat Yourself — Не повторяйся! ...	Повторение одного и того же кода в нескольких местах — очень плохая идея . В связи с этим есть рекомендация, если какой-либо код встречается в листинге более двух раз, то его нужно выносить в отдельный метод . Это общая рекомендация, на самом деле нужно задуматься о выделении метода даже если вы встречаете повторение второй раз. Все принципы разработки ПО важны, но этот — особенно!
OR / Occam's Razor — Бритва Оккама ...	Очень распространенная идея, которая пришла в программирование из философии. Принцип получил свое имя в честь английского монаха Уильяма из Оккама. Данный принцип гласит следующее: «Не следует множить сущее без необходимости» . В сфере программирования, это правило трактуется следующим образом — не нужно создавать лишние сущности без необходимости в них . То есть, всегда задумывайтесь над тем, получите ли вы выгоду выделив дополнительный класс или метод. Ведь если вы будете выделять в отдельный метод одну строку, которая при этом еще и нигде больше не повторяется, то только запутаете и усложните свой код.
KISS / Keep It Simple Stupid — Делай это проще ...	Код нужно писать простым, без сложных конструкций, так как в противном случае это может значительно усложнить поддержание и отладку. Кроме того, другому программисту будет намного сложнее разобраться в хитросплетениях и сложных ветвлений листинга, что тоже потребует дополнительных затрат сил и времени. Поэтому всегда старайтесь использовать простые и логичные конструкции без глубокой вложенности, так вы упростите жизнь и себе, и коллегам.
YAGNI / You Aren't Gonna Need IT — Вам это не понадобится ...	Проблема которой страдают многие программисты. Стремление разработать сразу весь потенциально необходимый (а иногда даже ненужный) функционал с самого начала процесса разработки. То есть, когда разработчик с самого начала добавляет все возможные методы в класс и реализует их, при этом в дальнейшем может даже ни разу ими и не воспользоваться. Поэтому согласно этой рекомендации, разрабатывайте только то, что вам необходимо в первую очередь , а в дальнейшем при необходимости наращивайте функциональные возможности. Так вы сэкономите многое количество сил, времени и нервов на отладку кода, который на самом деле не нужен.
BDUF / Big Design Up Front — Масштабное проектирование прежде всего	Это противоположный предыдущему принцип, который гласит, что перед тем, как приступить к разработке необходимо заранее продумать и спроектировать всю информационную систему вплоть до достаточно мелких деталей, и только после этого приступить к реализации по заранее подготовленному плану. Принцип имеет право на существование, но в последнее время встречается достаточно много его критики. В первую очередь это связано с устареванием плана за время проектирования и разработки. В связи с чем все равно приходится вносить последующие изменения. Но у него есть и неопровергнутые плюсы, <u>при грамотном проектировании можно значительно сократить затраты на дальнейшую отладку и исправление багов</u> . Кроме того, такие информационные системы обычно получаются более лаконичными и архитектурно правильными.

Open & Close / Открыто, закрыто — Открыто для внедрения новых областей, но закрыто для изменений	Код должен быть открытым для внедрения новых областей, но закрытым для изменений, не зависимо от того, пишете вы объекты на Java или модули на Python. Это относится ко всем видам проектов, но этот принцип особенно важен при выпуске библиотек или структур, предназначенных для использования другими пользователями.
SLAP / Single Level of Abstraction Principle (шлётнуть) — Принцип единого уровня абстракций	«Принцип единого уровня абстракций», SLAP, диктует нам, как мы должны организовывать свой код (в частности, функции), чтобы он оставался поддерживаемым. «Функции должны выполнять только одно действие, но выполнять его хорошо» (Роберт Мартин).
CQS / command-query separation — Разделение команд и запросов	Принцип разделения запросов (query) и команд на обработку (например сохранение данных или удаление), каждый из которых либо читает данные, либо обновляет их. Метод может быть запросом, возвращающим какое-то значение, или командой, но не одновременно тем и другим.

1. Java

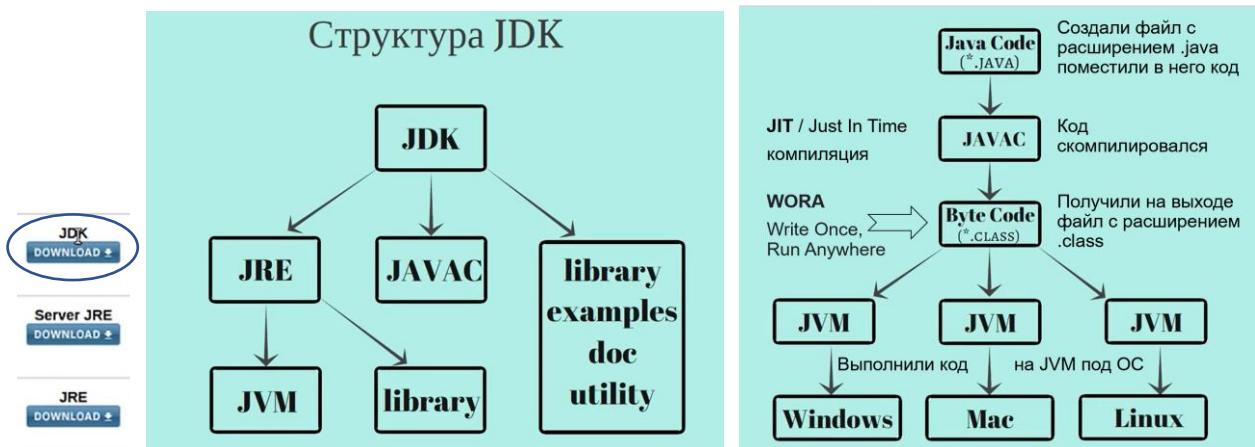
1. Какая основная идея языка?

Java – универсальный объектно-ориентированный язык со строгой типизацией.

В нём реализован принцип **WORA** (от английского: write once, run anywhere) или **«Написано однажды - работает везде»**. Это позволяет запускать приложения везде, где есть среда исполнения JRE (от английского: Java Runtime Environment).

Механизм работы программ следующий. Исходный материал транслируется в байт-код, который обрабатывается виртуальной машиной Java (JVM). При этом не имеет значения, какая операционная система установлена на устройстве.

Как это работает? https://youtu.be/FqcJt4_tKEw



Компилятор работает (узнать версию JDK)
javac -version

```
~/Vladynkin/hello_world/v1$ javac -version
javac 1.8.0_51
~/Vladynkin/hello_world/v1$
```

JUST IN TIME компиляция (JIT) – можно пересмотреть Владыкина 😊

Компиляция завершилась и в текущей директории появился файл с названием класса и расширением **.class**

```
~/Vladynkin/hello_world/v1$ javac -version
javac 1.8.0_51
~/Vladynkin/hello_world/v1$ echo $PATH
/home/stepic/Vladynkin/jdk1.8.0_51/bin:/usr/local/
sr/games:/usr/local/games
~/Vladynkin/hello_world/v1$ javac HelloWorld.java
~/Vladynkin/hello_world/v1$ ls
HelloWorld.class HelloWorld.java
~/Vladynkin/hello_world/v1$
```

Если классов несколько, то для каждого будет создан свой файл.

На белом фоне: так выглядит **.class** файл в текстовом редакторе в двоичном формате.

Чтобы прочитать байт-код, можно воспользоваться командой `javap -v HelloWorld.class`.
Получаем инструкцию байт-кода, выведенную в консоль.

Байт-код оперирует классами и методами также, как и исходный код. На уровне байт-кода **нет прямого доступа к памяти**.

JVM должны быть доступны все классы программы. Если она не найдёт какой-нибудь класс, то программа упадёт с ошибкой.

Классов в программе М.б. очень много, поэтому программа упаковывается в **архив JAR** (java архив).

Это обычный **ZIP архив**, но внутри содержится так называемый **манифест** с мета информацией о программе (имя, версия, имя главного класса).
Посмотреть содержимое архива, не распаковывая его:

```
~/Vladykin/hello_world/v1$ jar tf hw.jar  
META-INF/  
META-INF/MANIFEST.MF  
HelloWorld.class  
~/Vladykin/hello_world/v1$
```

Показываем JVM путь к сторонним классам и библиотекам (если они используются):
Для linux используется знак двоеточие, а для WIN точка с запятой.

```
~/Vladykin/hello_world/v1$ javac -classpath lib.jar HelloWorld.java  
~/Vladykin/hello_world/v1$ java -classpath lib.jar:hw.jar HelloWorld  
Hello world!  
~/Vladykin/hello_world/v1$
```

2. За счет чего обеспечивается кроссплатформенность?

Кроссплатформенность была достигнута за счёт создания виртуальной машина Java.

Java Virtual Machine или JVM – это программа, являющаяся прослойкой между операционной системой (ОС) и Java программой. В среде виртуальной машины выполняются коды Java программ. Сама JVM реализована для разных ОС.

Задача JVM – исполнять байт-код. Байт код для JVM может исполняться везде, где установлена JVM. Код не нужно перекомпилировать под каждую из платформ.

Например, реализация HotSpot (а есть и другое).

```
$ java -version  
java version "1.8.0_51"  
Java(TM) SE Runtime Environment (build 1.8.0_51-b16)  
Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode)  
$
```

Спецификация JVM описана компанией

The Java® Virtual Machine Specification

Oracle в документации:



Альтернативные языки программирования, которые могут работать на JVM.



Среды разработки: IntelliJ IDEA, NetBeans, Eclipse.

Инструменты для сборки программ: Ant (build.xml), Gradle (build.gradle), **Maven (pom.xml)**.

2. Какие преимущества у JAVA?

Объектно-ориентированное программирование

Структура данных становится объектом, которым можно управлять, создавать отношения между различными объектами.

Язык высокого уровня с простым синтаксисом и плавной кривой обучения

Синтаксис Java основан на C ++, поэтому Java похожа на С. Тем не менее, синтаксис Java проще, что позволяет новичкам быстрее учиться и эффективнее использовать код для достижения конкретных результатов.

Стандарт для корпоративных вычислительных систем

Корпоративные приложения — главное преимущество Java с 90-х годов, когда организации начали искать надежные инструменты программирования не на С.

Безопасность

Благодаря отсутствию указателей и Security Manager (политика безопасности, в которой можно указать правила доступа, позволяет запускать приложения Java в "песочнице").

Независимость от платформы

Можно создать Java-приложение на Windows, скомпилировать его в байт-код и запустить его на любой другой платформе, поддерживающей виртуальную машину Java (JVM). Таким образом, JVM служит уровнем абстракции между кодом и оборудованием.

Язык для распределенного программирования и комфортной удаленной совместной работы

Специфическая для Java методология распределенных вычислений называется Remote Method Invocation (RMI). RMI позволяет использовать все преимущества Java: безопасность, независимость от платформы и объектно-ориентированное программирование для распределенных вычислений.

Кроме того, Java также поддерживает программирование сокетов и методологию распределения CORBA для обмена объектами между программами, написанными на разных языках.

Автоматическое управление памятью (вопрос спорный: плюс или минус)

Разработчикам Java не нужно вручную писать код для управления памятью благодаря автоматическому управлению памятью (АММ).

Многопоточность

Поток — наименьшая единица обработки в программировании. Чтобы максимально эффективно использовать время процессора, Java позволяет запускать потоки одновременно, что называется многопоточностью.

Стабильность и сообщество

Сообщество разработчиков Java не имеет себе равных. Около 45% респондентов опроса StackOverflow 2018 используют Java.

4. Какие недостатки у Java?

Платное коммерческое использование (с 2019) – это утверждение спорно

Низкая производительность

Из-за компиляции и абстракции с помощью виртуальной машины, а также приложение очистки памяти.

Многословный код

Java — это более легкая версия неприступного C++, которая вынуждает программистов прописывать свои действия словами из английского языка. Это делает язык более понятным для неспециалистов, но менее компактным.

Не развитые инструменты по созданию GUI приложений на чистой Java.

5. Что такое JDK? Что в него входит?

JDK (Java Development Kit) – включает **JRE** (минимальная реализация JVM) и набор инструментов разработчика приложений на языке Java:

- компилятор JAVAC или **Just In Time компилятор**
- стандартные библиотеки классов Java (**library**)
- примеры
- документация
- различные утилиты (программы для выполнения специализированных типовых задач, связанных с работой оборудования и операционной системы (ОС)).



6. Что такое JRE? что в него входит?

JRE (Java Runtime Environment) – это минимально-необходимая реализация виртуальной машины для исполнения Java-приложений.

Состоит из **JVM**, **ClassLoader** и **Library** (стандартного набора библиотек и классов Java).

7. Что такое JVM?

JVM (Java Virtual Machine) – виртуальная машина Java исполняет байт-код Java, предварительно созданный из кода JIT компилятором, с помощью встроенного интерпретатора байткода. HotSpot представляет собой реализацию концепции JVM.

8. Что такое byte code?

Байт-код Java — набор инструкций, скомпилированный компилятором, исполняемый JVM.

9. Что такое загрузчик классов (ClassLoader)?

ClassLoader обеспечивает загрузку классов Java. Если говорить точнее, обеспечивают загрузку его наследники, конкретные загрузчики классов – сам **ClassLoader абстрактен**.

public abstract class ClassLoader используется для передачи в JVM скомпилированного байт-кода, хранится в файлах с расширением **.class**

Согласно спецификации Java SE для того, чтобы получить работающий в JVM код, необходимо выполнить три этапа, ТРИ основных действия в строгом порядке:

- 1) **Загрузка:** находит и импортирует двоичные данные для типа.

- связывание выполняет проверку, подготовку и (необязательно) разрешение
- проверка типа обеспечивает правильность импортируемого типа
- подготовка памяти выделяет память для переменных класса и инициализация памяти значениями по умолчанию

- 2) **Разрешение:** преобразует символические ссылки из типа в прямые ссылки.

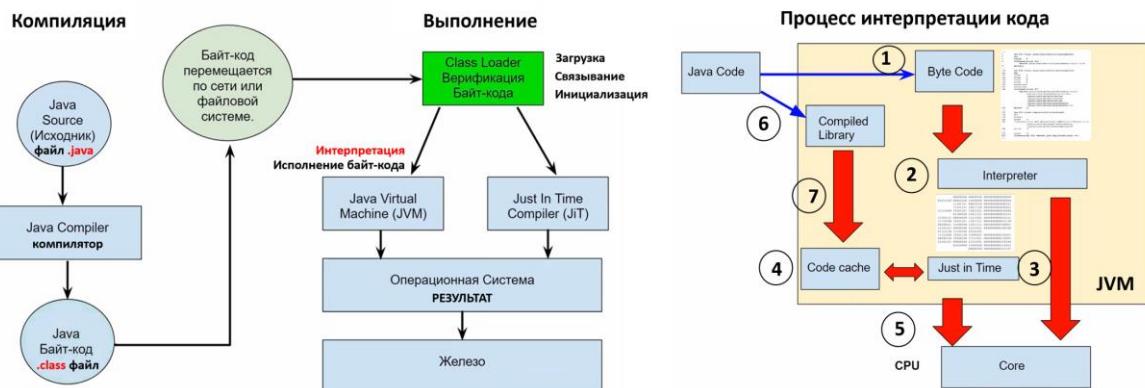
- 3) **Инициализация:** вызывает код Java, который инициализирует переменные класса их правильными начальными значениями.

Каждый загрузчик хранит указатель на родительский, чтобы суметь передать загрузку если сам будет не в состоянии этого сделать.

При запуске JVM, используются ТРИ типа загрузчиков (классов):

Смотри дальше подробное описание алгоритма динамической загрузки классов.

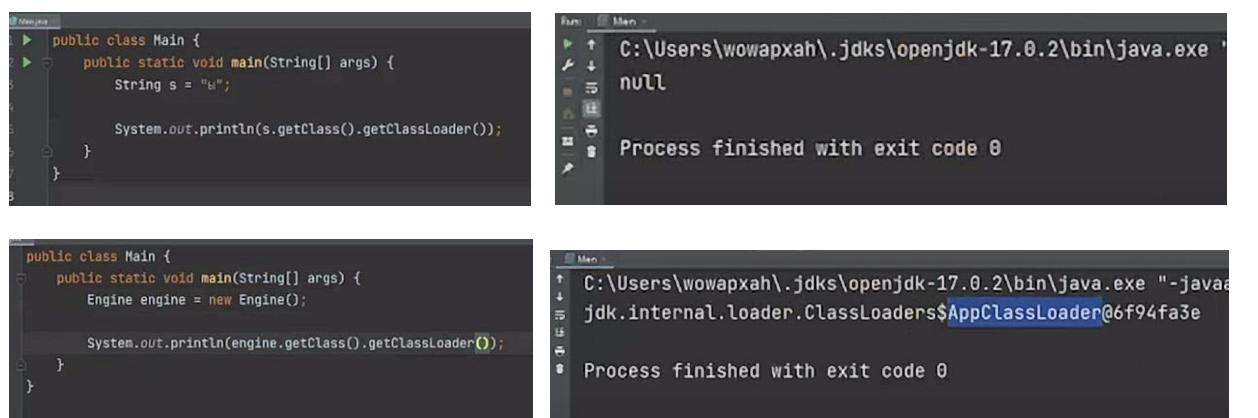
Bootstrap (базовый загрузчик)	Загружает <u>стандартные классы JDK</u> (Java Development Kit) из архива rt.jar
Extension ClassLoader (з-к расширений)	Загружает <u>классы расширений</u> , которые по умолчанию находятся в каталоге JAR файлы в директории lib/ext из JRE
System ClassLoader (системный з-к)	Загружает <u>классы приложения</u> , определённые в переменной среды окружения java.class.path



В Java используется иерархия загрузчиков классов, где корневым, разумеется, является **базовый**. Далее следует загрузчик **расширений**, а за ним уже **системный**.

Каждый загрузчик хранит указатель на родительский для того, чтобы смочь делегировать ему загрузку в том случае, если сам будет не в состоянии этого сделать.

Почему так? (авторский вопрос)



Нативный метод реализуется на языке, отличном от Java (например C, C++).

- **плюсы:** можно обратиться напрямую к операционной системе; возможность использования библиотек других языков.
- **минусы:** просадка производительности; теряется статическая типовая информация.

Стринг находится в ланг пакете, а все что в нем, грузится базовым Bootstrap загрузчиком.

10. Что такое JIT?

JIT (Just-in-time compilation) - компиляция на лету или **динамическая компиляция**.

Технология увеличения производительности программных систем, использующих байт-код, путем компиляции байт-кода в машинный код во время работы программы. В основном отвечает за оптимизацию производительности приложений во время выполнения.

11. Что такое сборщик мусора? (Garbage collector)

В Java используется автоматическое управление памятью. Программист выделяет память, а за освобождение отвечает JVM. Когда программа больше не ссылается на объект (прямые или косвенные ссылки), то объект удаляется, а память переиспользуется. **Сборщик мусора – это демон-поток**, который выполняет две задачи: **поиск и очистка мусора**.

Разбираемся на примере **HotSpot** (популярная реализация JVM). Все **объекты**, которые явно или неявно создаются Java-приложением, размещаются **в куче**.

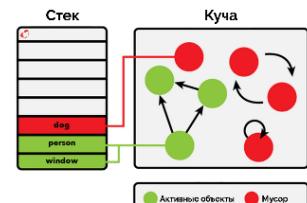
Что такое мусор, как определить, что объекты мёртвые?

Для обнаружения мусора есть ДВА основных подхода: **учёт ссылок и трассировка**.

Учет ссылок (Reference counting).

Если объект **не имеет ссылок, он считается мусором**.

Проблема: невозможно выявить циклические ссылки (когда два объекта не имеют внешних ссылок, но ссылаются друг на друга -> приводит к утечке памяти)



Трассировка (Tracing), используется в HotSpot (популярная разновидность JVM).

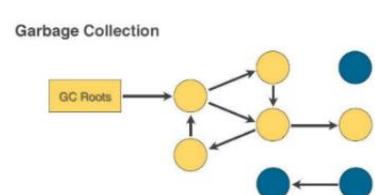
До объекта можно добраться из корневых точек (GC root).

Если до чего-то добраться нельзя, то это мусор.

Всё, что доступно из «живого» объекта, также является «живым».

Типы корневых точек **GC Roots** java приложения:

- объекты в статических полях классов
- объекты, доступные из стека потоков
- объекты из JNI (Java Native Interface) ссылок в native методах



Сборщики мусора бывают разные:

Java **HotSpot VM** предоставляет разработчикам на выбор СЕМЬ различных сборщика мусора:

Serial (последовательный) — самый простой вариант для приложений с небольшим объемом данных и не требовательных к задержкам. Редко когда используется, но на слабых компьютерах может быть выбран виртуальной машиной в качестве сборщика по умолчанию.

Parallel (параллельный) — наследует подходы к сборке от последовательного сборщика, но добавляет параллелизм в некоторые операции, а также возможности по автоматической подстройке под требуемые параметры производительности.

Concurrent Mark Sweep (CMS) — нацелен на снижение максимальных задержек путем выполнения части работ по сборке мусора параллельно с основными потоками приложения. Подходит для работы с относительно большими объемами данных в памяти. Использует инкрементальный алгоритм сборки (см. дальше).

Garbage-First (G1, см. ниже) — создан для постепенной замены CMS, особенно в серверных приложениях, работающих на многопроцессорных серверах и оперирующих большими объемами данных.

Epsilon GC — разработан для случаев, когда сборка мусора вообще не нужна.

Epsilon (2017, Алексей Шипилёв) выглядит как любой GC для OpenJDK, подключенный с помощью -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC.

Epsilon GC занимается только аллокацией памяти и ничего не делает для её освобождения. При достижении лимита памяти виртуальная машина Java остановится.

Зачем? тестирование производительности, анализ накладных расходов других сборщиков мусора, облегчение разработки виртуальной машины.

ZGC — пытается удерживать паузы на субмиллисекундном уровне, даже при работе с очень большими кучами.

Shenandoah GC — еще один сборщик, нацеленный на ультракороткие паузы независимо от размера кучи.

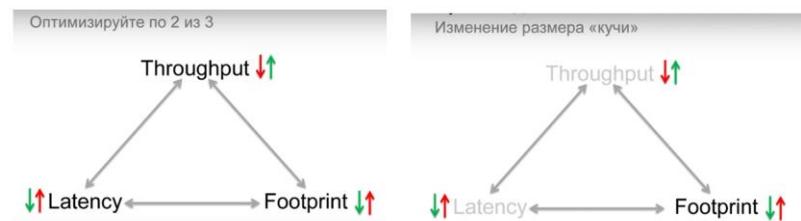
Как определяется производительность для сборщика мусора?

- **Throughput** (пропускная способность) – объём вычислительных ресурсов, затрачиваемых GC
- **Latency** (задержка) – на какое время прерывается работа приложения
- **Footprint** (след) – объём используемой памяти

Можно и нужно выбирать оптимальную стратегию, исходя из целей разработки.

Варианты:

- Оптимизируем 2 из 3 компонентов (от чего и во имя чего можем отказаться)
- Изменяем размер кучи: ячейки имеют размер от 1мб по дефолту до 32 мб (влияем)



Сборщик мусора выбирается в зависимости от цели и задач приложения.

Два подхода к сборке мусора, два алгоритма работы:

STW «stop-the-world» - остановка приложения на период уборки (остановка нужна, чтобы никакие новые изменения не произошли в программе, новые объекты не появились и т.д.)

- проще определять достижимость объектов «граф объектов заморожен»)
- проще перемещать объекты в куче (в процессе сборки куча может находиться в некорректном состоянии)

НО!

- приложение останавливается на время сборки мусора
- зависит от размера кучи (объёма памяти живых объектов)

Инкрементальная сборка

- Попытка уменьшить паузы, вызванные GC (за счёт большого количества коротких пауз и фоновой сборки)
- Требуется синхронизировать работу GC с приложением (барьеры на чтение/запись)

Такой алгоритм занимает больше времени.

STW

- Продолжительные паузы
- ... но никакой лишней нагрузки для потоков приложения
- Максимальный throughput



Инкрементальная сборка

- Короткие паузы
- Лишняя нагрузка в потоках приложения
- Минимальные паузы за счет снижения throughput



Универсального «UBER» GC не существует =)

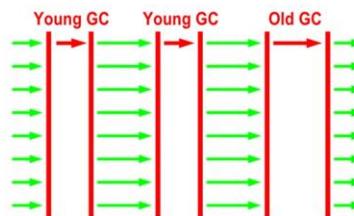
Красными границами обозначено время остановки программы STW «stop-the-world»

GC в Hotspot JVM

Взгляд извне

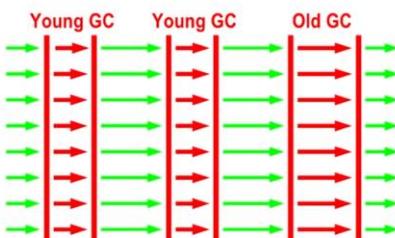
- **SerialGC**
 - последовательная сборка молодого и старого поколений
- **ParallelGC**
 - максимальный throughput
 - параллельная сборка молодого и старого поколений
- **CMS**
 - предсказуемость
 - по возможности, сборка мусора в фоновом режиме
- **G1**
 - предсказуемость
 - слабо подвержен фрагментации

SerialGC



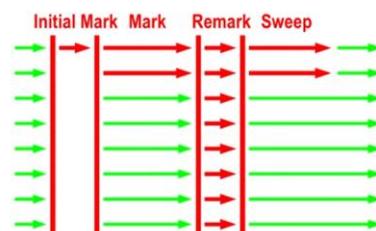
- Молодое поколение: последовательный копирующий GC
- Старшее поколение: последовательный Mark-Sweep-Compact
 - Аллокация: линейная
 - -XX:+UseSerialGC

ParallelGC



- Молодое поколение: параллельный копирующий
- Старшее поколение: параллельный Mark-Compact
 - Аллокация: линейная
- -XX:+UseParallelGC -XX:+UseParallelOldGC

CMS



- Молодое поколение: параллельный копирующий GC
- Старшее поколение: фоновый Mark-Sweep GC
 - Аллокация: free-листы
 - Компактификация только при FullGC
 - X:+UseConcMarkSweepGC

Сборка мусора с поколениями G1 или Garbage-First GC <https://youtu.be/iGRfyhE02IA>

- 1) Слабая гипотеза о поколениях («слабая» — это значит, что не обязательно так, но если в приложении это так устроено, то такая сборка будет наиболее эффективна)
 - большинство объектов временные
 - старые объекты редко ссылаются на молодые
- 2) Молодые и старые объекты содержатся отдельно
 - в «пространствах», называемых «поколения» (generations)
 - возможны разные алгоритмы для молодого и старого поколения
 - молодое поколение можно собирать отдельно от старого

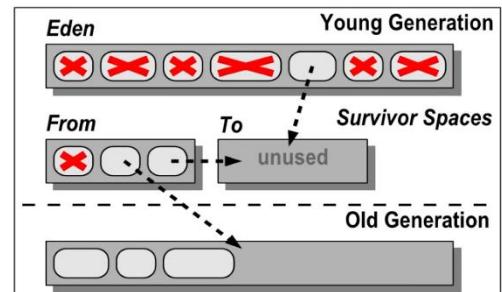
Процессы сборки мусора разделяются на несколько видов: Minor GC, Major GC, Full GC. Сначала происходит малая сборка мусора в области непостоянных объектов (Young Gen)

1) Minor GC (малая)

Частый и быстрый, работает только с областью памяти "young generation".

- приложение приостанавливается на начало сборки мусора (такие остановки называются stop-the-world)
- «живые» объекты из Eden (тут создаются new Object) перемещаются в область памяти «To»
- «живые» объекты из «From space» перемещаются в «To space» или в «Old generation», если они достаточно «старые»
- Eden и «From» очищаются от мусора
- «To space» и «From space» меняются местами
- «очищенное» приложение возобновляет работу

Молодое поколение: сборка мусора

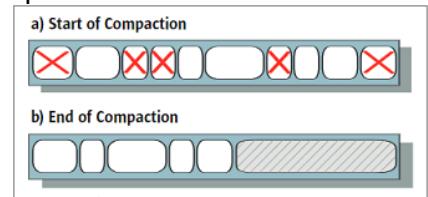


Когда место заканчивается в области непостоянных объектов, тогда запускается полная или старшая сборка мусора. И она работает сразу в обоих поколениях (Young & Old).

2) Major GC (старшая)

Редкий и более длительный, затрагивает объекты старшего поколения.

В принцип работы «major GC» добавляется процедура «уплотнения», позволяющая более эффективно использовать память. В процедуре живые объекты перемещаются в начало. Таким образом, мусор остается в конце памяти.



3) Full Garbage Collection (полная)

Полный сборщик мусора сначала запускает Minor, а затем Major.

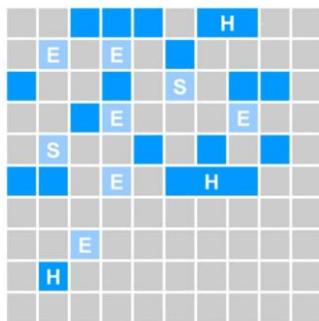
Хотя порядок может быть изменен, если старое поколение заполнено, и в этом случае он освобождается первым, чтобы позволить ему получать объекты от молодого поколения. Сборка отработала, потом происходит дефрагментация объектов памяти и цикл начинается сначала.

Garbage-First, G1 GC:

- фоновый и параллельный
- высокая предсказуемость работы (паузы малы!)
- сборщик мусора с поколениями, НО...

Куча состоит из регионов. Нет ФИЗИЧЕСКОГО разделения между молодым и старым поколением. Принадлежность к регионам определяется динамически. Для каждого региона известно, где находятся объекты, ссылающиеся на него.

Максимальное количество «регионов» в куче G1 GC – это 2^{11} (два в одиннадцатой). Каждый регион фиксированного размера от 1 мб (по дефолту) до 32 мб.

G1: Структура «кучи»Разбита на регионы **фиксированного размера от 1МБ до 32МБ****Молодое поколение**

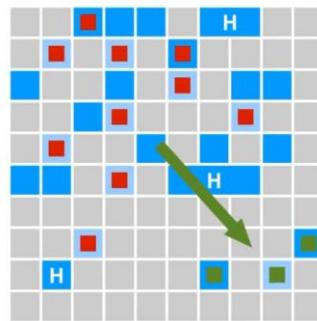
- Набор регионов
 - Eden
 - Survivor
- Выбирается динамически

Eden – аллокация объектов

Survivor (выжившие) – пережили хотя бы одну сборку мусора

Большие объекты

- Не помещается в регион
- Называется "humongous"
- Хранится в наборе смежных регионов

G1: Структура «кучи»**Collection Set**

- Регионы, в которых будет происходить GC
 - Все молодое поколение
 - Некоторые регионы из старшего поколения
- Фоновая маркировка определяет наиболее подходящие

Сборка

- Копирование объектов в регионы, помеченные как часть «To»-пространства
 - Survivor-регионы
 - Регионы из старшего поколения

Справка: Большие объекты еще называют **Акселераторами** (иногда их называют гигантскими, огромными или «humongous»).

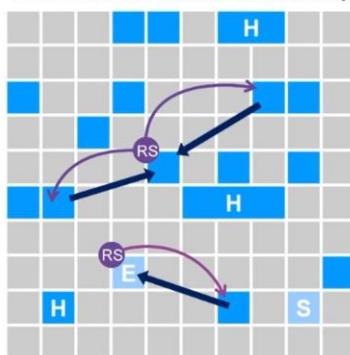
Объекты-акселераторы, размер которых настолько велик, что создавать их в Eden, а потом таскать за собой по Survivor'ам слишком накладно. В этом случае они сразу размещаются в **Tenured** (старшее поколение).

RSet хранит инфо о местонахождении ссылок на OLD объекты из региона. Почему old? Хранить ссылки из старшего поколения на младшее нет смысла, т.к. молодое УЖЕ собрано на первом этапе. В молодом поколении % мёртвых объектов максимальный.

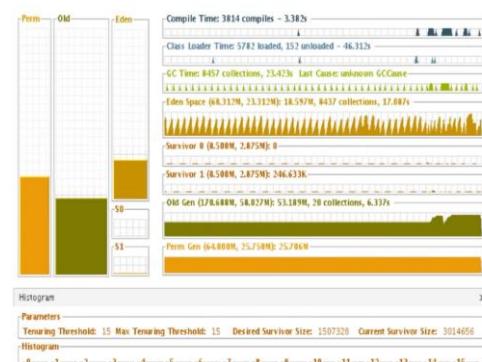
Справа полезный плагин, визуализирующий процесс сборки мусора.

G1: Структура «кучи»

Освобождение памяти : компактификация за счёт копирования

**RSet == Remembered Set**

- Информация о местонахождении ссылок на объекты из региона
- Позволяет собирать регионы независимо
- RSet поддерживается
 - Из старого в молодое поколение
 - Между регионами в старом поколении

VisualVM / VisualGC

12. Что такое Heap и Stack память в Java? Чем они отличаются?

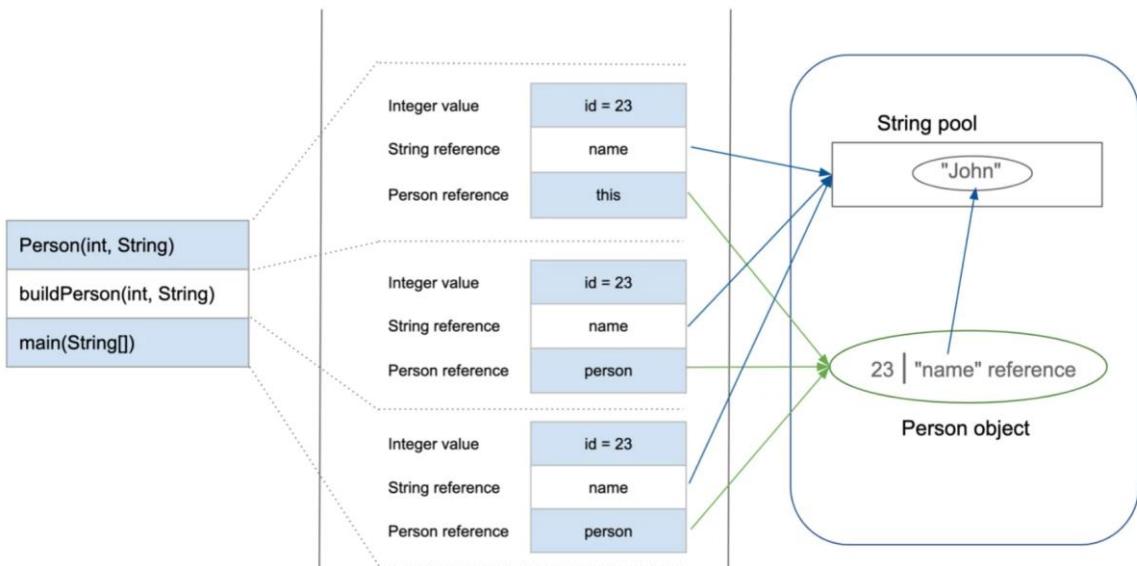
Для оптимальной работы приложения JVM делит память:

- на **область стека Stack** – тут хранятся примитивы и ссылки на объекты
- и **область кучи Heap** – тут создаются и хранятся объекты

Call Stack

Stack Memory

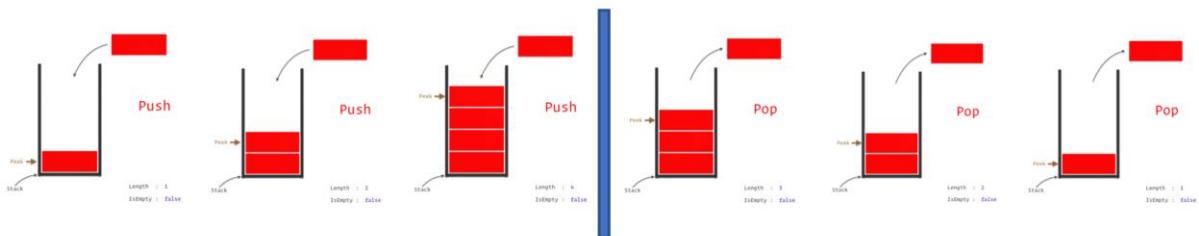
Heap Space



Stack содержит **stack frame**'ы, они делятся на три части:

- параметры метода
- указатель на предыдущий фрейм
- локальные переменные

Стек работает по схеме **LIFO** (последним вошел, первым вышел, похоже на стопку книг).



Как это работает:

Когда вызывается новый метод, содержащий примитивные значения или ссылки на объекты, то на вершине стека под них выделяется блок памяти. Когда метод завершает выполнение, блок памяти (frame), отведененный для его нужд, очищается и пространство становится доступным для следующего метода. При этом поток выполнения программы возвращается к месту вызова этого метода с последующим переходом к следующей строке кода.

Особенности стека:

- Он заполняется и освобождается по мере вызова и завершения новых методов
- Переменные в стеке существуют до тех пор, пока выполняется метод, в котором они были созданы
- Если память стека будет заполнена, Java бросит исключение **StackOverflowError**
- Доступ к этой области памяти осуществляется быстрее, чем к куче
- Стек потокобезопасен, т.к. для каждого потока создается свой отдельный стек

Структура **Heap** зависит от выбранного сборщика мусора (см. выше)

Куча или Heap — название структуры данных, с помощью которой реализована динамически распределяемая память приложения.

Размер кучи — размер памяти, выделенный операционной системой (ОС) для хранения кучи (под кучу).

Для Java 11 и выше Значение Xmx составляет 25% доступной памяти с максимальным объемом 25 ГБ. Однако при наличии 2 ГБ или менее физической памяти устанавливается значение 50 % доступной памяти с минимальным значением 16 МБ и максимальным значением 512 МБ.

Для Java 8 значение Xmx — это половина доступной памяти, минимум 16 МБ и максимум 512 МБ.

Куча разбита на несколько более мелких частей, называемых поколениями:

- **Young Generation** — область, где размещаются недавно созданные объекты. Когда она заполняется, происходит быстрая сборка мусора.
- **Old Generation** — здесь хранятся долгоживущие объекты. Когда объекты из Young Generation достигают определенного порога «возраста», они перемещаются в Old Generation
- **Permanent Generation** — эта область содержит метаинформацию о классах и методах приложения (начиная с Java 8 эта область памяти была упразднена).

Особенности кучи:

- Когда эта область памяти полностью заполняется, Java бросит **OutOfMemoryError**
- Доступ к ней медленнее, чем к стеку
- Эта память, в отличие от стека, автоматически не освобождается. Для сбора неиспользуемых объектов используется сборщик мусора
- В отличие от стека, куча не является потокобезопасной и ее необходимо контролировать, правильно синхронизируя код

MetaSpace — это специальное пространство кучи, отделенное от кучи основной памяти. JVM хранит здесь весь статический контент. Это включает в себя все статические методы, примитивные переменные и ссылки на статические объекты. Кроме того, он содержит данные о байт-коде, именах и JIT-информации.

С помощью опций Xms и Xmx можно настроить начальный и максимально допустимый размер кучи соответственно. Существуют также опции для настройки величины стека.

Суммируем **различия** между памятью стека и пространством кучи:

Свойства	Стек	Куча
Использование приложением	Для каждого потока используется свой стек	Пространство кучи является общим для всего приложения
Размер	Предел размера стека определен операционной системой	При запуске процесса ОС выделяет память для размещения кучи. В дальнейшем память для кучи (под кучу) может выделяться динамически.
Хранение	Хранит <u>примитивы и ссылки</u> на объекты	Все созданные <u>объекты</u> хранятся в куче
Порядок	Работает по схеме последним вошел, первым вышел (LIFO)	Доступ к этой памяти осуществляется с помощью сложных методов управления памятью, включая Young Generation, Old и Permanent Generation
Существование	Память стека существует пока выполняется текущий метод	Пространство кучи существует пока работает приложение
Скорость	Обращение к памяти стека происходит значительно быстрее, чем к памяти кучи	Медленнее, чем стек
Выделение и освобождение памяти	Эта память автоматически выделяется и освобождается, когда метод вызывается и завершается соответственно (StackOverflowError – нет места)	Память в куче выделяется, когда создается новый объект и освобождается сборщиком мусора, когда в приложении не остается ни одной ссылки на его (OutOfMemoryError)

3. Процедурная Java

В java переменные объявляются явно, с указанием типа. Имена типов – ключевые слова. Переменная примитивного типа – это просто ячейка памяти, в которой хранится значение переменной. Например, `Int a = 100;` – это 4 байта памяти, хранящие число, и к памяти мы обращаемся через переменную. Примитивные типы **передаются по значению!**

```
int a = 100;
a: 100
```

Присваиваем значение переменной **a** в переменную **b**, и это значит, что создается **копия значения a в другой ячейке памяти.**

```
int b = a;
b: 100
```

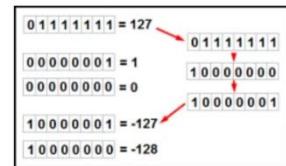
Изменение копии никак не повлияет на оригинальную ячейку памяти, т.е. на исходную переменную a.

Типы данных в Java:

примитивные (целочисленные, вещественные, логические) и ссылочные.

Почему верхняя граница int 127, а не 128?

128 не сделать, так как в двоичном коде **первая цифра – это знак**, если написать число после 0111 1111, то это уже будет -128. С Byte неизбежно будет потребляться меньше памяти (если только в массивах). Связано это с JVM, она приводит Byte и short к 32 битному int, так как системы в основном 32/64 разрядные



Двоичная система счисления:

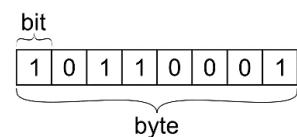
Внутри компьютера – транзисторы))) У транзистора нет знаков от 0 до 9, а есть только состояние ВКЛ (1) или ВЫКЛ (0). Мы «шифруем числа» транзисторами/лампочками, имеющими только два состояния ВКЛ/ВЫКЛ.

И получается, что в каждом разряде двоичного числа **не ЧИСЛО ДЕСЯТКОВ**, а **ЧИСЛО ДВОЕК в разной степени**. Например, десятичная система: $19547 = 1 \cdot 10000 (10^4) + 9 \cdot 1000 (10^3) + 5 \cdot 100 (10^2) + 4 \cdot 10 (10^1) + 7 \cdot 1$

И так как у нас знака только два, то мы можем обозначить либо наличие двойки, либо отсутствие. То есть в числах двойка в степени всегда обозначается единицей и количество нулей = степени ($2^2 = 4$ пишем: двойка (пишем 1) и степень = 2 (пишем два 0) и получаем запись 100)

Для того, чтобы перевести число 33 из десятичной системы счисления в двоичную, необходимо осуществить последовательное деление на 2 до тех пор, пока остаток не будет меньше, чем 2.
Полученные остатки записываем в обратном порядке, таким образом: **Ответ: $33_{10} = 100001_2$**

$$\begin{array}{r} 33 \\ -\frac{32}{1} \\ -\frac{16}{0} \\ -\frac{8}{0} \\ -\frac{4}{0} \\ -\frac{2}{0} \\ -\frac{1}{0} \\ \hline 0 \end{array}$$



Разряд – это всего лишь "структурный элемент" представления числа. То есть число 10 состоит из двух разрядов: нам надо 2 разряда, 2 места, 2 элемента, чтобы записать это число. Нам это важно понимать потому, что в двоичной системе счисления разряд – это бит (bit). Слово Bit произошло от английского "binary digit", то есть двоичное число. Оно может быть только или 0 или 1.



1. Какие примитивные типы данных есть в Java?

Тип данных определяет диапазон значений, которые может хранить переменная или константа. Целые типы различаются по размеру отведенной для них памяти.

В Java есть **4 группы, 8 примитивных типов**: легко запомнить по степени двойки (2^3)

- Целые числа - byte, short, int, long (4 типа или 2^2)
- Вещественные - float, double (числа с плавающей точкой, 2 типа или 2^1)
- Символьный – char (один тип или 2^0)
- Логический – boolean (один тип или 2^0)

Тип, название, (default value)	Размер байт / бит	Минимум знач.	Максимум знач.	Пример записи	Комментарий
Целочисленный byte (0)	1 байт = 8 бит	- 128 - 2^7	+127 + 2^7	byte a = 3;	
Целочисленный short (0)	2 байта = 16 бит	- 32768 - 2^{15}	+ 32767 + $2^{15} - 1$	short a = 3;	Используется редко. JavaDoc - you can use a short to save memory in large arrays
Символьный char (нулевой символ '\u0000')	2 байта = 16 бит один символ	хранятся в памяти как числа в диапазоне от 0 до 65 536, обозначающие Юникод символа.		char ch=102; // символ 'f' одинарные кавычки char ch='\\u0066' //	символьный тип, знаки символов UTF-16 также беззнаковое целое P.S. Символы тоже относят к целочисленным типам из-за особенностей представления в памяти и традиций.
Целочисленный int (0)	4 байта = 32 бита	- 2^{31}	+ $2^{31} - 1$	int a = 4;	Диапазон значений от -2^{31} до $2^{31} - 1$. Максимальное значение int 2147483648 – 1. При вычислении остаток отбрасывается (11 / 4 = 2)
Целочисленный long (0L)	8 байта = 64 бита	- 2^{63}	+ $2^{63} - 1$	long num = 2147483649L; (суффикс L)	Используется при работе со временем или большим расстоянием. Очень часто используется как ID при работе с БД.
Вещественный, с плавающей точкой float (0.0f)	4 байта = 32 бита	- 3.4×10^{38}	+ 3.4×10^{38}	float x = 8.5F; Чтобы указ., как float, использовать суффикс f	

Вещественный, с плавающей точкой double (0.0d)	8 байта = 64 бита	$\pm 4.9 \times 10^{-324}$	$\pm 1.8 \times 10^{308}$	double x = 8.5; целая и дробная часть разделяются точкой	Для больших чисел используют экспоненциальную форму записи (для отделения мантиссы от порядка используется символ "e" или символ "E"), например, число -3.58×10^7 записывается как $-3.58E7$ У вещественных побитовые операции НЕ поддерживаются!
Логический boolean (false)	8 (в массивах), 32 (не в массивах используется int)	одно из двух значений: «истина» или «ложь» (true или false)	boolean switch = true;		Используются в операциях отношения (сравнения) и логических операциях.

Справка: Деньги в float и double НЕ надо вычислять (javadoc - This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Numbers and Strings covers BigDecimal and other useful classes provided by the Java platform.). При делении на ноль вернут бесконечность (плюс или минус).

Переменные с плавающей точкой могут хранить не только численные значения, но и любой из особо определенных флагов (состояний): отрицательная бесконечность, отрицательный нуль, положительная бесконечность, положительный нуль и «отсутствие числа» (not-a-number, NaN). **При сложении минус бесконечности и плюс - выведет NaN (not a Number).**

Особые случаи арифметических операций.

Какой тип имеет литерал 0x0bp3? (Владыкин)
Ответ: double

Примитивные типы нужны для производительности вычислений. Работа с объектами намного медленнее.

2. Что такое char?

16-разрядное беззнаковое целое (2 байта памяти), представляющее собой символ UTF-16 (буквы и цифры). Хранятся в памяти как числа в диапазоне от 0 до 65 536, обозначающие Юникод символа (особенностей представления в памяти и традиции). Дефолтное значение '\u0000' (нулевой символ).

Справка: есть литералы, которые нельзя задать явно через консоль. Они пропечатываются в 16-битовом представлении. Чтобы справиться с задачей, нужно пользоваться **префиксом \u**. Если убрать \u, запись окажется в восьмеричном виде.

40	28	41	29	42	2A	43	2B	44	2C	45	2D	46	2E	47	2F
()	*		+	,			-		.					/
48	30	49	31	50	32	51	33	52	34	53	35	54	36	55	37
0	1	2	3	4	5	6	7								

3. Сколько памяти занимает boolean?

В стандартной реализации Sun JVM и Oracle HotSpot JVM тип boolean занимает **4 байта/32 бита, как и тип int**. Однако, в определенных версиях JVM имеются реализации, где в массиве boolean каждое значение занимает по 1-му биту.

Почему?

Современная архитектура не способна указать адрес памяти в битах, а только в байтах.

Авторский вопрос

Дан REST контроллер, программисту нужно принять с фронта какой-то флаг. Флаг может иметь ОДНО из трёх значений. Какой тип данных выберет УМНЫЙ программист и почему?

Ответ: boolean (два его значения true/false и значение null объекта класса-обёртки Boolean).

Вспоминаем:

Шесть операторов сравнения:

> >= < <= == !=

И логические операции (справа)

Приоритет логических операций

1. !	Инверсия (отрицание)
2. &	Конъюнкция (умножение)
3. ^	Дизъюнкция (ложение)
4.	Импликация (следование)
5. &&	Эквивалентность (равнозначность)
6.	

Вопрос от Владыкина: Укажите пары эквивалентных булевыхских операторов. Операторы эквивалентны, если их таблицы истинности совпадают, т.е. для любой пары аргументов оба оператора дают один и тот же результат (\mid И \parallel ... \wedge И $!=$)

4. Что такое классы-обертки?

Обертка — это специальный класс, который хранит внутри себя значение примитива.

Обёртки нужны для реализации дженериков.

Классы-обёртки неизменяемые (Immutable), поэтому при каждой автоупаковке (за исключением значений из pool) создается новый объект, что может привести к неразумному расходу памяти.

Например, завернём примитивный int в объект класса Integer. Что при этом происходит в памяти?

Примитивный int — это 4 байта памяти, в которых лежит значение. Ссылочный Integer — это ссылка на объект Integer, лежащий в куче, а внутри этого объекта лежи примитивное значение int.

5. Что такое автоупаковка и автораспаковка?

Для того, чтобы иметь **возможность оперировать с простыми числами (и boolean) как с объектами** (ссылочными типами) были придуманы классы-обёртки.

Процесс преобразования примитивных типов в эквивалентные объекты (ссылочный тип) называется автоупаковкой (autoboxing), а обратный ему — автораспаковкой (unboxing).

Автоупаковка/автораспаковка **не работает для массивов!**

Справка:

Ссылочные типы — это **объекты, включая Object, Array и Function**. Ввиду того что эти типы могут содержать очень большие объемы весьма разнородных данных, переменная, содержащая ссылочный тип, фактически его значения не содержит. Она содержит ссылку на место в памяти, где размещаются реальные данные.

Для присваивания ссылок-примитивов объектам их классов-оберток (и наоборот) не требуется ничего делать, все происходит автоматически.

Благодаря автораспаковке, мы смело можем писать выражения, **не используя методы конвертации**. Теперь за этим следит компилятор Java.

5. Что такое явное и неявное приведение типов (иначе называют кастинг)? В каких случаях в java нужно использовать явное приведение?

Явные и неявные преобразования (приведение типов)

Каждый базовый тип данных занимает определенное количество байт памяти.

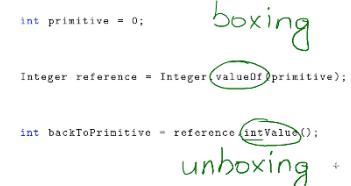
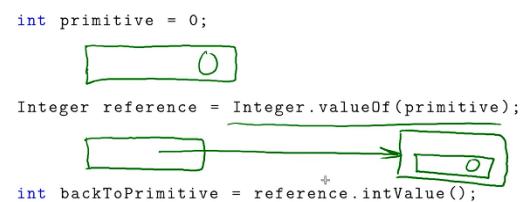
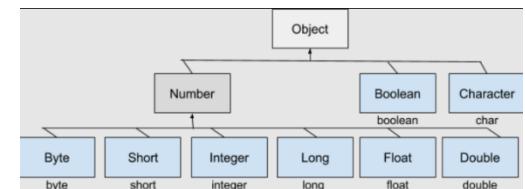
```
int a = 4;
byte b = a; // ! Ошибка
```

Т.к. мы пытаемся присвоить некоторые данные, которые занимают 4 байта, переменной, которая занимает один байт.

В этом случае необходимо использовать операцию преобразования типов (операция()):

```
int a = 4;
byte b = (byte)a; // преобразование типов: от типа int к типу byte
```

Когда в одной операции вовлечены данные разных типов, не всегда необходимо использовать операцию преобразования типов.



Неявное (безопасное) приведение – автоматическое расширение типа переменной от меньшего к большему.

Старшие разряды более широкого типа просто заполняются знаковым битом исходного значения.

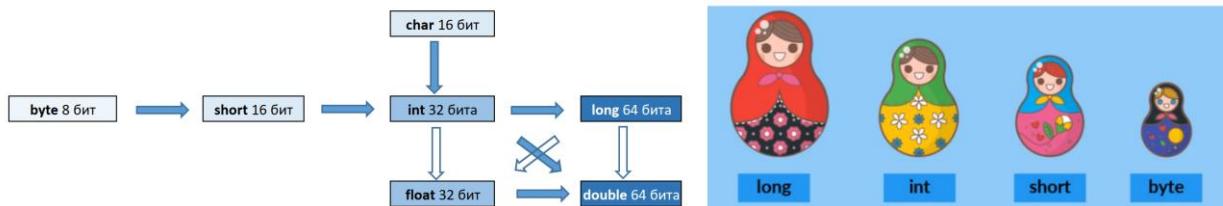
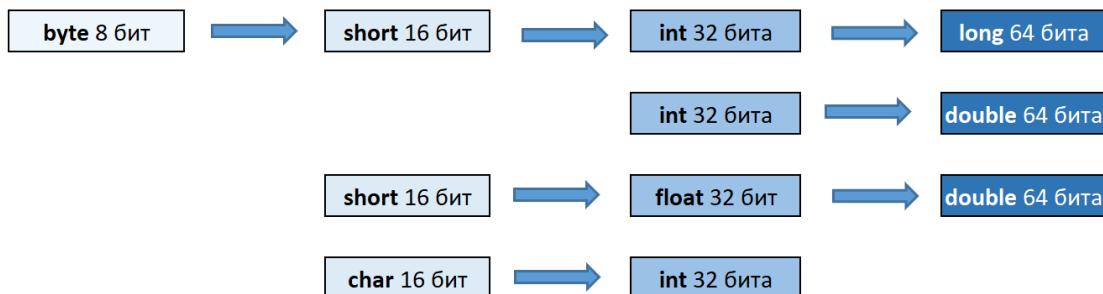
Явное приведение – это явное сужение от большего к меньшему.

Необходимо явно указать сужаемый тип.

В случае с объектами мы можем делать неявное (автоматическое) приведение от наследника к родителю, но не наоборот, иначе получим **ClassCastException**.



Расширяющие автоматические преобразования представлены следующими цепочками:



Синими стрелками на рисунке показано, какие преобразования типов могут выполняться автоматически.
Пунктирными стрелками показаны автоматические преобразования с потерей точности.

Почему приведение переменной типа long во float происходит с потерей точности? (из слайда Каты)

long это 64 бита, из которых 63 бита отводится под число, float же это 32х битное число, у которого точность примерно 24 бита, double - 64 бита, с точностью примерно 53 бита, т.е. даже приводя long в double есть риск потери данных, т.к. 53 бита не хватит чтобы точно записать 63 бита от long.
Если более подробно, то структура вещественного числа в памяти следующая, 1 бит под знак, часть под экспоненту, часть под мантиссу. float это 1-8-23, double это 1-11-52. Т.е. если мы хотим записать целое число в double, то мы его можем туда записать только используя часть памяти, выделенной для мантиссы, часть для экспоненты использовать нельзя. А у нас long 63 бита имеет под число и как записать 63 бита в 52 не потеряв данные? (ответ коллег из слайда)

При преобразовании значений с плавающей точкой к целочисленным значениям, происходит усечение дробной части:

double a = 56.9898;

int b = (int)a;

Здесь значение числа b будет равно 56, несмотря на то, что число 57 было бы ближе к 56.9898.

Кеширование или Integer pool.

— Когда мы присваиваем переменной типа Integer значение типа int, при этом вызывается метод Integer.valueOf:

Код	Что происходит на самом деле
1 Integer x = 5;	1 Integer x = Integer.valueOf(5);

В классе-обёртке Integer есть внутренний класс **IntegerCache**. Он объявлен как private static. В этом внутреннем классе кешированные объекты находятся в массиве cache[].

Кеширование выполняется при первом использовании класса-обёртки. После первого использования, вместо создания нового экземпляра (кроме использования конструктора), используются кешированные объекты.

Код метода valueOf() класса Integer выглядит так:

```
public static Integer valueOf(int i) {
    if ((i >= IntegerCache.low) &&
        (i <= IntegerCache.high))
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

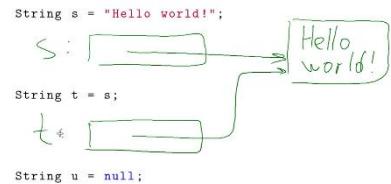
Справка: Кеширование касается не только класса-оболочки Integer. Имеются аналогичные реализации кеширования для других классов-оболочек целочисленных типов: ByteCache, ShortCache, LongCache, CharacterCache. Кешированные объекты не используются при создании объекта-обёртки с помощью конструктора.

Авторский вопрос: бывает пул boolean(-ов)? Ответ: да, только в пуле две константы true и false.

7. Какие нюансы у строк в Java?

String – это строка символов.

Переменная ссылочного типа – это ячейка памяти, содержащая ссылку на участок памяти, представляющий собой объект.



Когда мы присваиваем значение **s** в другую переменную или передаём ее в качестве параметра в метод, то выполняется копирование ссылки.

Если через вторую ссылку объект модифицируется, то те же изменения будут видны через первую ссылку (т.к. объект один). Ссылка может быть пустой – значение null.

String – неизменяемый (Immutable), финализированный (final) класс в Java, поэтому все манипуляции со строкой всегда будут создавать новую строку (ресурсоёмкость).

Как реализована неизменяемость строк? 1) final 2) private 3) нет сеттеров

Строки – объекты класса String, очень распространены, поэтому в некоторых случаях обрабатываются отлично от всех остальных объектов.

Строковые литералы записываются в **двойных кавычках**.

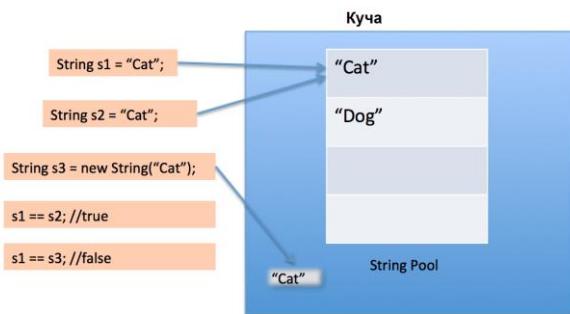
8. Что такое пул строк?

String Pool – специальный массив для хранения строк в памяти Heap (оптимизация).

Пул строк – это набор строк, который хранится в памяти Java heap.

Можно создавать объекты класса String, используя оператор **new** (предоставляя значение строки в двойных кавычках).

Диаграмма объясняет, как пул строк размещается в памяти Java heap и что происходит, когда мы используем различные способы создания строк.



Когда мы создаем строку, используя двойные кавычки, JVM ищет в пуле строк другую строку с таким же значением. Если строка найдена, то возвращается только **ссылка на существующий объект** класса String, **иначе создается новый объект** с полученным значением, и сохраняется в пуле.

Когда мы используем оператор new, виртуальная машина создает объект String, но не хранит его в пуле строк. Мы можем использовать метод **intern()** для сохранения строки в пуле строк, или получения ссылки, если такая строка уже находится в пуле.

```
1 public class StringPool {  
2     public static void main(String[] args) {  
3         String s1 = "Cat";  
4         String s2 = "Cat";  
5         String s3 = new String("Cat");  
6  
7         System.out.println("s1 == s2 :" +(s1==s2));  
8         System.out.println("s1 == s3 :" +(s1==s3));  
9     }  
10 }
```

Программа выведет следующее:

1	s1 == s2 :true
2	s1 == s3 :false

Справка:

Пул строк реализован на основе паттерна «Приспособленец». Это структурный шаблон проектирования, при котором объект, представляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым.

10. Почему не рекомендуется изменять строки в цикле? Что рекомендуется использовать?

Строка является неизменной и финализированной в Java, поэтому все наши манипуляции со строкой всегда будут создавать новую строку.

Манипуляции со строками ресурсоемкие, поэтому Java обеспечивает два полезных класса для манипуляций со строками – StringBuffer и StringBuilder.

StringBuffer и StringBuilder являются **изменяемыми** классами.

Операции с StringBuffer поток-безопасны и синхронизированы, а методы StringBuilder не поток-безопасны. Поэтому, когда несколько нитей работают с одной строкой, мы должны использовать StringBuffer, но в однопоточном окружении мы должны использовать StringBuilder. **StringBuilder более производительный**, чем StringBuffer, поскольку не обременен синхронизацией.

9. Почему строки не рекомендуется использовать для хранения паролей?

Строка неизменяемая в Java и хранится в пуле строк. С тех пор, как она была создана, она остается в пуле, пока не будет удалена сборщиком мусора, поэтому, когда мы думаем, что закончили работу с паролем, **НО объект остается доступным в памяти**.

некоторое время, и нет способа избежать этого. Это риск безопасности, поскольку кто-либо, имеющий доступ к дампу памяти сможет найти пароль в виде чистого текста.

Если мы **используем массив символов** для хранения пароля, мы можем очистить его после того, как закончим с ним работать. Таким образом, мы можем контролировать, как долго он находится в памяти, что позволяет избежать риска безопасности, свойственного строке `public char [] getPassword() {}`

10. Почему String неизменяемый и финализированный класс?

Безопасность и String pool – это основные причины неизменяемости String в Java. Безопасность объекта неизменяемого класса String обусловлена такими фактами:

- вы можете передавать строку между потоками и не беспокоиться что она будет изменена
- нет проблем с синхронизацией (не нужно синхронизировать операции со String)
- отсутствие утечек памяти
- в Java строки используются для передачи параметров для авторизации, открытия файлов и т.д. Неизменяемость позволяет избежать проблем с доступом.
- возможность кэшировать hash code

String pool позволяет экономить память и НЕ создавать новые объекты для каждой повторяющейся строки. В случае с изменяемыми строками - изменение одной приводило бы к изменению всех строк одинакового содержания.

11. Почему строка является популярным ключом в HashMap в Java?

У изменяемых объектов хэшкод изначально равен нулю и он таков до тех пор, пока мы не вызвали метод `hash()`, тогда и генерируется хэш-значение – в момент получения кода.

Поскольку **строка неизменная**, её `hashcode` кэшируется в момент создания и нет необходимости рассчитывать его снова.

Это делает строку отличным кандидатом для ключа в Map и его обработка будет быстрее, чем других ключей HashMap. Это причина, почему строка наиболее часто используемый объект в качестве ключа HashMap.

14. Что делает метод `intern()` в классе String?

Когда метод `intern()` вызван, если пул строк уже содержит строку, эквивалентную к нашему объекту (что подтверждается методом `equals(Object)`), тогда возвращается ссылка на строку из пула. В противном случае **объект строки добавляется в пул**, и ссылка на этот объект возвращается.

Этот метод всегда возвращает строку, которая имеет то же значение, что и текущая строка, но гарантирует что это будет строка из пула уникальных строк.

Сравниваем ссылки на объекты, поэтому используем знак ==

```

public class Main {
    public static void main(String[] args) {
        String str1 = "JavaRush";
        String str2 = "JavaRush";
        String str3 = (new String(original: "JavaRush")).intern();
        String str4 = (new String(original: "JavaRush")).intern();

        System.out.println("строка 1 равна строке 2 " + (str1 == str2));
        System.out.println("строка 2 равна строке 3 " + (str2 == str3));
        System.out.println("строка 3 равна строке 4 " + (str3 == str4));
    }
}

```

Run: Main
/Users/antonkupreychik/Library/Java/Java
строка 1 равна строке 2 true
строка 2 равна строке 3 true
строка 3 равна строке 4 true
Process finished with exit code 0

А если уберём метод **intern()**, то получим другой результат...

```

public class Main {
    public static void main(String[] args) {
        String str1 = "JavaRush";
        String str2 = "JavaRush";
        String str3 = (new String(original: "JavaRush"));
        String str4 = (new String(original: "JavaRush"));

        System.out.println("строка 1 равна строке 2 " + (str1 == str2));
        System.out.println("строка 2 равна строке 3 " + (str2 == str3));
        System.out.println("строка 3 равна строке 4 " + (str3 == str4));
    }
}

```

Run: Main
/Users/antonkupreychik/Library/Java/Java
строка 1 равна строке 2 true
строка 2 равна строке 3 false
строка 3 равна строке 4 false
Process finished with exit code 0

15. Можно ли использовать строки в конструкции switch?

Java 7 расширяет возможности оператора **switch** для использования строк, ранние версии Java не поддерживают этого.

Если вы реализуете условный поток для строк, вы можете использовать условия **if-else** и вы можете использовать оператор **switch**, если используете Java 7 или поздние версии.

Ключевые моменты использования **switch** для строк в Java.

- использование строк в конструкции **switch** делает **код читабельнее**, убирая множественные цепи условий **if-else**
- строки в switch чувствительны к регистру** (см. пример)
- оператор **switch** использует метод **String.equals()** для сравнения полученного значения со значениями **case**, поэтому добавьте **проверку на NULL** во избежание **NullPointerException**
- согласно документации Java 7 для строк в **switch**, компилятор Java формирует **более эффективный байткод** для строк в конструкции **switch**, чем для сцепленных условий **if-else**
- убедитесь, что это будет использоваться с Java 7 или поздней версии, иначе получите **Exception**

```

public class SwitchStringExample {
    public static void main(String[] args) {
        printColorUsingSwitch("red");
        printColorUsingSwitch("red");
        // оператор switch регистрозависимый
        printColorUsingSwitch("RED");
        printColorUsingSwitch(null);
    }
}

private static void printColorUsingIf(String color) {
    if (color.equals("blue")) {
        System.out.println("BLUE");
    } else if (color.equals("red")) {
        System.out.println("RED");
    } else {
        System.out.println("INVALID COLOR CODE");
    }
}

private static void printColorUsingSwitch(String color) {
    switch (color) {
        case "blue":
            System.out.println("BLUE");
            break;
        case "red":
            System.out.println("RED");
            break;
        default:
            System.out.println("INVALID COLOR CODE");
    }
}

```

16. Какая основная разница между **String**, **StringBuffer**, **StringBuilder**?

Строка является неизменной и финализированной в Java, поэтому все наши манипуляции со строкой всегда будут создавать новую строку.

Манипуляции со строками ресурсоемкие, поэтому Java обеспечивает два полезных класса для манипуляций со строками – `StringBuffer` и `StringBuilder`.

`StringBuffer` и `StringBuilder` являются **изменяемыми** классами.

Операции с `StringBuffer` поток-безопасны и синхронизированы, а методы `StringBuilder` не поток-безопасны. Поэтому, когда несколько нитей работают с одной строкой, мы должны использовать `StringBuffer`, но в однопоточном окружении мы должны использовать `StringBuilder`. **StringBuilder** более производительный, чем `StringBuffer`, поскольку не обременен синхронизацией.

17. Существуют ли в java многомерные массивы?

Многомерные массивы в их классическом понимании в java не существуют.

Многомерный массив всегда прямоугольный и неразрывен в памяти.

А то, что в java считается многомерным - в других языках ещё называют "зубчатым массивом" или **массивом массивов**. В интернете написано: В Java можно объявить **массив массивов**, известный как многомерный массив. К примеру: `int[][] a = new int[3][4];`

18. Какими значениями инициируются переменные по умолчанию?

Название	Тип данных	Значение по умолчанию
<code>byte</code>	целочисленный	0
<code>short</code>	целочисленный	0
<code>char</code>	символьный	"\u0000" (нулевой символ)
<code>int</code>	целочисленный	0
<code>long</code>	целочисленный	0L
<code>float</code>	вещественный,	0.0f
<code>double</code>	с плавающей точкой	0.0d
<code>boolean</code>	логический	false

19. Что такое сигнатура метода?

Сигнатурой метода — это **имя метода плюс параметры** (причем порядок параметров имеет значение). В сигнатуру метода НЕ входит возвращаемое значение, а также бросаемые им исключения.

Контракт метода — это тип возвращаемого значения, имя + параметры метода + бросаемые исключения.

20. Расскажите про метод main

Точка входа в программу (одна запускает, в классе могут быть еще методы `main`). Если в классе несколько методов `main` только один из них является точкой входа в программу (контракт `psvm` – `public static void main`).

Вопрос (Владыкин):

Предположим, вы написали программу, состоящую из двух классов, и в каждом классе объявили точку входа — метод `main`. Что из этого выйдет? **Ответ:** Программа скомпилируется и запустится. Неоднозначности не возникнет, поскольку при запуске всегда явно указывается класс, в котором JVM должна искать метод `main`.

21. Каким образом переменные передаются в методы, по значению или по ссылке?

Только по значению. Для передаваемого объекта (класс `String`, например) значением будет ссылка.

4. ООП в Java

1. Какие виды классов есть в java?

Top level class (обычный класс):

- abstract class (абстрактный);
- final class (финализированный).

Interfaces (Интерфейс)

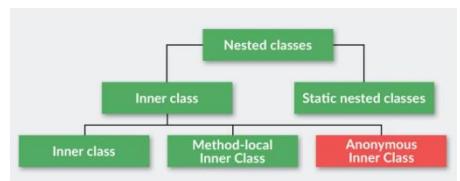
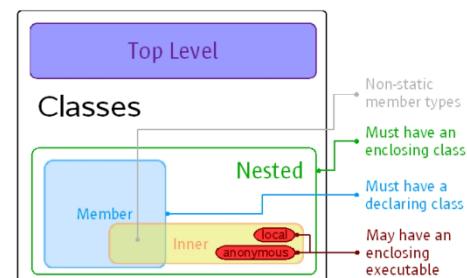
Enum (Перечисление)

Nested class (Вложенный класс):

- Static nested class (статический вложенный класс);
- Member inner class (простой внутренний класс);
- Local inner class (локальный класс);
- Anonymous inner class (анонимный класс).

<https://iavarush.ru/groups/posts/2193-anonimnihe-klassih>

Taxonomy of Classes in the Java Programming Language



Вложенный класс — это любой класс, объявление которого происходит в теле другого класса или интерфейса. Класс **верхнего уровня** — это класс, который не является вложенным классом.

Sealed types (изолированные типы или запечатанные классы) — это такие классы и интерфейсы, которые могут запрещать наследовать или реализовывать себя.

Изолированный класс или интерфейс можно наследовать или реализовывать только тем классам и интерфейсам, которым это разрешили.

Класс запечатывается применением **модификатора sealed** прямо в определении класса.

Начиная с Java 15 <https://habr.com/ru/company/jugru/blog/498494/>

```

package com.example.geometry;

sealed class Shape {...}
... class Circle extends Shape {...}
... class Rectangle extends Shape {...}
... class Square extends Shape {...}
  
```

2. Расскажите про вложенные классы. В каких случаях они применяются?

Класс называется **вложенным** (Nested class), если он определен внутри другого класса.

Вложенный класс должен создаваться только для того, чтобы **обслуживать обрамляющий его класс**. Если вложенный класс оказывается полезен в каком-либо ином контексте, он должен стать классом верхнего уровня.

Пример: класс **Iterator** помещаем внутрь класса **ArrayList** (итератор, должен помнить коллекцию, которую он обходит, чтобы получать элементы и текущую позицию обхода). Добавляемся этого, поместив определение итератора внутрь класса **ArrayList**.

Таким образом, каждый экземпляр итератора будет иметь собственное состояние (`int cursor`), хранящееся в полях, а также неявную ссылку на экземпляр внешнего класса. То есть из итератора можно напрямую обращаться к полям класса **ArrayList** (`elementData`).

```

Вложенные классы
package java.util;

public class ArrayList<E> {
    Object[] elementData;
    public Iterator<E> iterator() {
        return new Itr();
    }
    private class Itr implements Iterator<E> {
        int cursor;
        // ...
    }
}
  
```

Вложенные классы имеют доступ ко всем (в том числе приватным) полям и методам внешнего класса, но не наоборот. Из-за этого разрешения использование вложенных классов приводит к некоторому нарушению инкапсуляции.

Если **вложенный класс** имеет модификатор **static**, то теряется неявная связь с внешним классом и экземпляры вложенного класса будут жить своей собственной, независимой жизнью.

```
public class Collections {  
    public static final List<E> EMPTY_LIST = new EmptyList<E>();  
    public static final <T> List<T> emptyList() {  
        return (List<T>) EMPTY_LIST;  
    }  
    private static class EmptyList<E> {  
        // ...  
    }  
}
```

При этом размещение одного класса внутри другого мотивируется либо желанием скрыть вложенный класс (сделать его приватным), либо тесной логической связью внешнего и вложенного класса.

Существуют 4 категории вложенных классов:

- Static nested class (статический вложенный класс);
- Member inner class (простой внутренний класс);
- Local inner class (локальный класс);
- Anonymous inner class (анонимный класс).

Статический класс – это вложенный класс, объявленный с использованием ключевого слова **static**. К классам верхнего уровня модификатор **static** неприменим.

Такие категории классов, за исключением первого, также называют **внутренними (Inner class)**. Внутренние классы ассоциируются не с внешним классом, а с экземпляром внешнего.

Каждая из категорий имеет рекомендации по своему применению:

Не статический: если вложенный класс должен быть виден за пределами одного метода или он слишком длинный для того, чтобы его можно было удобно разместить в границах одного метода и, если каждому экземпляру такого класса необходима ссылка на включающий его экземпляр.

Статический: если ссылка на обрамляющий класс не требуется.

Локальный: если класс необходим только внутри какого-то метода и требуется создавать экземпляры этого класса только в этом методе.

Анонимный: если к тому же применение класса сводится к использованию лишь в одном месте и уже существует тип, характеризующий этот класс.

3. Что такое «локальный класс»? Каковы его особенности?

Local inner class (Локальный класс) – это вложенный класс, который может быть декларирован в любом блоке, в котором разрешается декларировать переменные.

Как и простые внутренние классы (Member inner class) локальные классы имеют имена и могут использоваться многократно. Как и анонимные классы, они имеют окружающий их экземпляр только тогда, когда применяются в нестатическом контексте.

Локальные классы имеют следующие особенности:

- Видны только в пределах блока, в котором объявлены;
- Не могут быть объявлены как **private/public/protected** или **static**;
- Не могут иметь внутри себя статических объявлений (полей, методов, классов);

- Имеют доступ к полям и методам обрамляющего класса;
- Могут обращаться к локальным переменным и параметрам метода, если они объявлены с модификатором final.

4. Что такое «анонимные классы»? Где они применяются?

Это **вложенный локальный класс без имени**, который разрешено декларировать в любом месте обрамляющего класса, разрешающем размещение выражений.

Создание экземпляра анонимного класса происходит одновременно с его объявлением. В зависимости от местоположения анонимный класс ведет себя как статический либо как нестатический вложенный класс – в нестатическом контексте появляется окружающий его экземпляр.

Анонимные классы имеют несколько ограничений:

- Их использование разрешено только в одном месте программы (месте его создания)
- Применение возможно только в том случае, если после порождения экземпляра нет необходимости на него ссылаться
- Реализует лишь методы своего интерфейса или суперкласса, т.е. не может объявлять каких-либо новых методов, так как для доступа к ним нет поименованного типа.

Анонимные классы обычно применяются для:

- создания объекта функции (function object), например реализация интерфейса Comparator;
- создания объекта процесса (process object), такого как экземпляры классов Thread, Runnable и подобных
- в статическом методе генерации;
- инициализации открытого статического поля final, которое соответствует сложному перечислению типов, когда для каждого экземпляра в перечислении требуется отдельный подкласс.

Справка: В документации Oracle приведена хорошая рекомендация: «Применяйте анонимные классы, если вам нужен локальный класс для одноразового использования».

Анонимный класс — это полноценный внутренний класс. Поэтому у него есть доступ к переменным внешнего класса, в том числе к статическим и приватным.

5. Каким образом из вложенного класса получить доступ к полю внешнего класса?

Статический вложенный класс имеет прямой доступ ТОЛЬКО к статическим полям обрамляющего класса.

Простой внутренний класс, может обратиться к любому полю внешнего класса напрямую. В случае, если у вложенного класса уже существует поле с таким же литералом, то обращаться к такому полю следует через ссылку на его экземпляр.

Например: Outer.this.field.

6. Что такое перечисления (enum)?

Тип Enum — специальный тип данных, который позволяет переменной быть набором **предопределенных констант**. Другими словами, он позволяет создать переменную, которая может принимать несколько значений (каждое из значений, объявленных в самом перечислении).

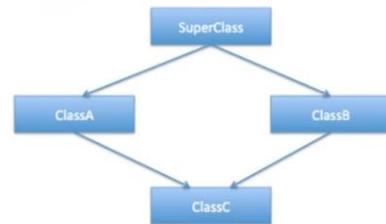
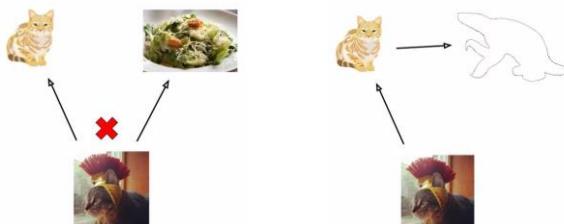
К Enum можно применять методы:

- name() - возвращает строку, имя элемента
 - ordinal() - возвращает порядковый номер
 - values() – возвращает массив возможных значений перечислений в том же порядке
 - finalize()
 - clone()
- equals(), hashCode(), toString()

```
public enum DayOfWeek {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY(args),
    SATURDAY,
    SUNDAY;
}

//fields, methods
```

7. Как проблема ромбовидного наследования решена в java?



Кот Цезарь не является наследником салата Цезарь, но он – наследник Млекопитающего.

В Java множественное наследование не поддерживается в классах, Но поддерживается в интерфейсах. И один интерфейс может расширять множество других интерфейсов.

```
package com.journaldev.inheritance;
public interface InterfaceA {
    public void doSomething();
}
package com.journaldev.inheritance;
public interface InterfaceB {
    public void doSomething();
}

package com.journaldev.inheritance;
public interface InterfaceC extends InterfaceA, InterfaceB {
    //same method is declared in InterfaceA and InterfaceB both
    public void doSomething();
}
```

Это отлично работает, потому что интерфейсы только объявляют методы, а реализация будет выполнена в классах, наследующих интерфейс. Таким образом, нет никакой возможности получить неоднозначность во множественном наследовании интерфейсов.

Каждый раз, **переопределяя любой метод** суперкласса или реализуя метод интерфейса, используем аннотацию @Override.

8. Что такое конструктор по умолчанию?

Бывают дефолтный (не принимающий аргументы конструктор) и параметризованный. Это определяется во время создания объекта. Создаётся неявно.

Если создать явно, то дефолтный сотрётся, а созданный нами уже не будет дефолтным (будет просто конструктором без параметров, куда теоретически можно будет добавить и тело, и методы).

Если в классе не объявить ни одного конструктора, то, будет генерироваться конструктор по умолчанию. То есть, конструктор по умолчанию генерируется в классе автоматически только в том случае, если класс не содержит реализаций других конструкторов.

Если класс содержит реализацию хотя бы одного конструктора с параметрами, то, чтобы объявить конструктор по умолчанию, его нужно объявлять в классе явным образом.

9. Могут ли быть приватные конструкторы? Для чего они нужны?

Приватный конструктор **запрещает создание экземпляра класса** вне методов самого класса, чтобы гарантировать существование только одного объекта определённого класса, предположим какого-то ресурса, например БД.

Например: Паттерн Singleton (Одиночка): объект в программе должен быть один и к нему есть глобальный доступ.

10. Расскажите про классы-загрузчики и про динамическую загрузку классов.

<https://www.youtube.com/watch?v=plfrmpZiuNY&t=99s>

Загрузка и создание класса или интерфейса. Наглядный пример — класс **ClassLoader**. В Java есть три **стандартных загрузчика**:

Bootstrap (базовый загрузчик)	Загружает стандартные классы JDK (Java Development Kit) из архива rt.jar
Extension ClassLoader (з-к расширений)	Загружает классы расширений, которые по умолчанию находятся в каталоге JAR файлы в директории lib/ext из JRE
System ClassLoader (системный з-к)	Загружает классы приложения, определённые в переменной среды окружения java.class.path

ИЕРАРХИЯ: Bootstrap основной, далее идёт загрузчик расширений и потом системный. Каждый загрузчик хранит указатель на родительский загрузчик, чтобы иметь возможность делегировать ему загрузку.

ТРИ принципа загрузки классов:

Принцип делегирования

Запрос на загрузку класса передаётся родительскому загрузчику. Загрузка класса выполняется автоматически только, если родительский загрузчик не смог найти и загрузить класс. Этот подход позволяет загружать классы загрузчиком, который находится максимально близко к базовому. Таким образом достигается **максимальная область видимости классов**.

Справка: Каждый загрузчик ведёт учёт загруженных классов, помещая их в свой кэш. Когда происходит запрос на загрузку класса, происходит поиск этого класса в кэше загрузчика. Множество таких классов называется «**областью видимости**».

Принцип видимости

Загрузчик видит только свои классы и классы родителя. Он не имеет понятия о классах, загруженных его потомком.

Принцип уникальности

Класс **может быть загружен** только один раз. Делегирование позволит убедиться, что загрузчик, инициирующий загрузку класса, не перегрузит загруженный ранее в JVM класс.

Исключения, которые могут быть выброшены из-за ошибки на этом этапе:
ClassNotFoundException или **NoClassDefFoundError**.

В Java есть модель динамической загрузки классов:

- классы загружаются только тогда, когда используются (но некоторые базовые классы загружаются при старте приложения)

- для загрузки классов в память и их выполнения используется загрузчик классов (как только загрузчик загрузил необходимые классы, JVM начинает выполнять код)

Любая JVM включает в себя загрузчик классов. Исходя из спецификации Java Standard Edition для того, чтобы получить работающий в JVM код необходимо выполнение ТРЕХ этапов:

1) загрузка байт-кода и создание экземпляра класса

- поиск класса среди загруженных ранее классов
- получение байт-кода класса для загрузки
- проверка корректности байт-кода
- создание экземпляра класса
- загрузка родительских классов



Если родительские классы и интерфейсы не были загружены, то нужный класс считается незагруженным.

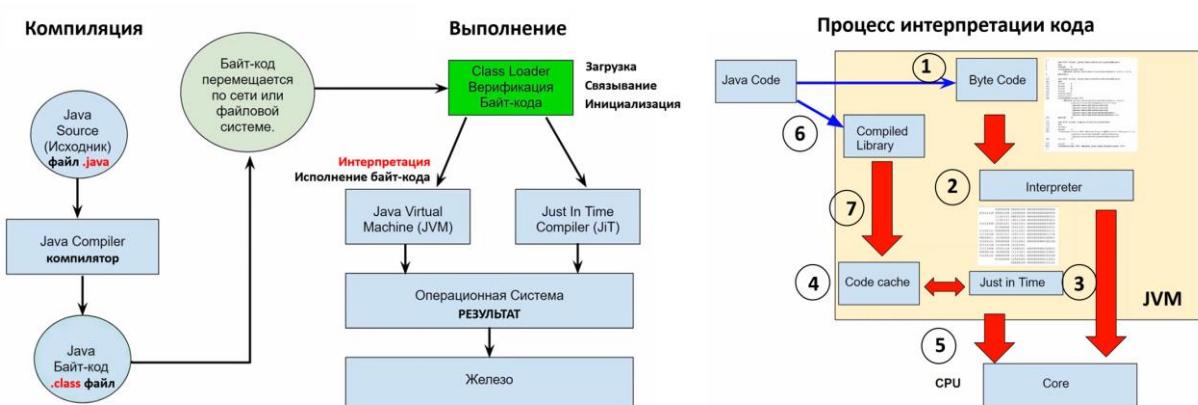
2) линковка или связывание, по спецификации этот этап разделяется еще на три:

- верификация: проверка корректности байт-кода
- подготовка: выделение памяти под статические поля и инициализация их значениями по умолчанию
- разрешение: разрешение символьных ссылок, типов полей и методов

3) инициализация полученного объекта

Все эти этапы выполняются последовательно с обязательным соблюдением следующих требований:

- класс должен быть полностью загружен перед тем, как он будет слинкован
- класс должен быть полностью проверен и подготовлен перед инициализацией
- ошибки разрешения ссылок происходят во время выполнения программы (даже если они были обнаружены на этапе линковки)



В Java реализована **ленивая загрузка классов**. Это означает, что:

- загрузка классов (ссылочных полей загружаемого класса) не выполнится, пока не будет явного обращения к полям
- то есть разрешение символьных ссылок по умолчанию не происходит
- разрешение символьных ссылок не привязано ни к какому из этапов загрузки классов

11. Чем отличаются конструкторы по умолчанию, конструктор копирования и конструктор с параметрами?

Конструктор **по умолчанию** не принимает никаких параметров.

Конструктор **с параметрами** принимает на вход параметры (обычно необходимые для инициализации полей класса).

Конструктор **копирования** принимает в качестве параметра объект текущего класса.

Конструктор – это специальный **метод**, вызываемый при создании экземпляра класса (оператор new). Задача конструктора – инициализировать состояние объекта и подготовить его к использованию.

Объявление конструктора состоит из:

- модификатора доступа
- имени класса

```
public final class Integer {
    private final int value = 0;
    public Integer(int value) {
        this.value = value;
    }
}
```

Когда в классе не объявлен ни один конструктор, то неявно создаётся конструктор по умолчанию, без параметров. Если нужно запретить создание экземпляров класса, то нужно сделать конструктор приватным: **private Math() {}**

В классе м.б. несколько перегруженных версий конструкторов с разными наборами параметров. При этом из одного конструктора можно вызывать другой.

Вопрос от Владыкина:

Какой модификатор доступа имеет конструктор без параметров, автоматически добавляемый компилятором для public-класса? (public)

```
public class BigInteger {
    public BigInteger(String val) {
        this(val, 10);
    }
    public BigInteger(String val, int radix) {
        // ...
    }
}
```

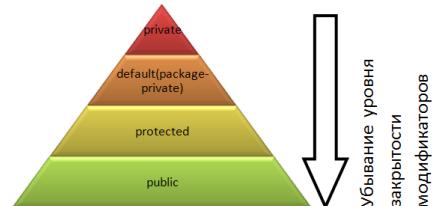
12. Какие модификаторы доступа есть в Java? Какие применимы к классам?

Модификаторы — ключевые слова, которые добавляют при инициализации для изменения значений. Чтобы использовать модификатор в Java, нужно включить его ключевое слово в определение класса, метода или переменной.

Существует два вида модификаторов в java:
4 шт. access modifiers и non-access modifiers.

Access modifier:

модификаторы доступа (private, default (по умолчанию), protected, public).



Non-access modifiers: модификаторы класса, метода, переменной и потока, используемые НЕ для доступа (static, abstract, synchronized, native, volatile, transient ...).

Модификаторы доступа позволяют задать допустимую области видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод.

ИНКАПСУЛЯЦИЯ:

Использование различных модификаторов гарантирует, что данные не будут искажены или изменены не надлежащим образом.

private (приватный) КЛАСС	члены класса доступны только внутри класса (видимый только для класса)
default (по умолчанию) ПАКЕТ	package-private, package level (доступ на уровне пакета): видимость класса/членов класса только внутри пакета, является модификатором по умолчанию - спец. обозначение не требуется
protected (защищённый) ПАКЕТ + ПОТОМКИ	члены класса доступны внутри пакета и в наследниках (видимый для пакета и всех подклассов)
public (публичный) ВЕЗДЕ И ВСЕМ	класс/члены класса доступны всем и везде

Если **модификатор доступа отсутствует**, значит доступ разрешён только в пределах пакета (пакеты служат для группировки связанных классов внутри программы).

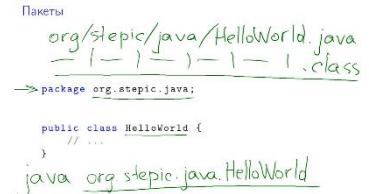
Когда мы говорим, что код из одного класса (class A) имеет доступ к коду из другого класса (class B), это означает что класс A может делать одну из трех вещей:

- создать экземпляр класса B
- наследовать класс B
- иметь доступ к определенным членам класса B.

Если сомневаетесь, открывать доступ или нет, поставьте **более строгий модификатор доступа** (Владыкин =)

Какие модификаторы доступа применимы к классу верхнего уровня (т.е. не вложенному в другой класс)?

- public
- отсутствие модификатора
(пакетный доступ package, он же default)

Пакеты


```
org/stepic/java/HelloWorld.java
package org.stepic.java;
public class HelloWorld {
    ...
}
```

Если класс объявлен public, то имя класса должно совпадать с именем файла, иначе будет ошибка компиляции.



```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello world!");
4     }
5 }
6 
```

Если есть вложенный класс (class Foo{} внутри другого), то **public** из них только ОДИН!

Ключевое слово **final** в объявлении класса означает, что от данного класса нельзя наследоваться. То есть он является **финальным в иерархии наследования**.

Модификаторы для переменных

Состояние экземпляра класса **хранится в полях** класса.

Объявление поля состоит из:

- модификатора доступа
- типа поля
- имени переменной

```
package java.lang;
public final class Integer {
    private final int value = 0;
```

Модификатор **final** означает, что **значение** полю можно присвоить только **один раз**. После чего изменять его будет нельзя.

Модификатор **final** можно применять и к параметрам методов, и к локальным переменным. Смысл один – значение присваивается однажды.

Модификаторы для методов

Поведение экземпляра класса **определяет метод** (еще говорят функция объекта). Метод исполняется в контексте конкретного экземпляра класса, поэтому он может обращаться к полям текущего объекта.

Объявление метода состоит из:

- модификатора доступа
- типа возвращаемого значения (или void)
- имени (с маленькой буквы)
- параметров (или без них)

```
public final class Integer {
    private final int value;
    public int intValue() {
        return value;
    }
}
```

Метод может иметь модификатор **final**.

Это означает что метод нельзя переопределить в классах-наследниках.

Что означает модификатор static?

Модификатор static (от англ. "статичный", "постоянный") применяют для создания методов и переменных класса. Модификатор указывает на привязку субъекта к текущему классу, **НО не к экземпляру** этого класса.

В этом случае поля и методы существуют **не зависимо от экземпляров класса** и могут вызываться просто по имени класса через точку. В комбинации с **final static** используется для объявления констант

```
public final class Integer {
    static final int MIN_VALUE = 0x80000000;
    public static int rotateRight(int i, int distance) {
        return (i >> distance) | (i << -distance);
    }
    // ...
}
```

Integer.rotateRight(5, 7)

Статические методы отличаются от обычных тем, что они привязаны к классу, выполняются в контексте класса, а не к объекту (конкретному экземпляру).

```
public class InstanceCounter {
    private static int numInstances = 0;
    protected static int getCount() {
        return numInstances;
    }
    private static void addInstance() {
        numInstances++;
    }
    InstanceCounter() {
        InstanceCounter.addInstance();
    }
    public static void main(String[] arguments) {
        System.out.println("Начинаем с " +
            InstanceCounter.getCount() + " экземпляра");
        for (int i = 0; i < 500; ++i) {
            new InstanceCounter();
        }
        System.out.println("Создано " +
            InstanceCounter.getCount() + " экземпляров");
    }
}
```

Важным свойством **статического метода** является то, что он **может обратиться только к статическим** переменным и методам.

Если указан модификатор static, значит метод можно вызывать, не создавая экземпляр содержащего его класса. По крайней мере один статический метод всегда есть в программе (psvm).

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello world!");
4     }
5 }
6
```

Статические переменные также известны как переменные класса.

В Java **локальные*** переменные **НЕ могут быть static** (т.е. объявлены статическими)

Справка: Ключевое слово static используется для создания переменных, которые будут существовать независимо от каких-либо экземпляров, созданных для класса. Только одна копия переменной static в Java существует вне зависимости от количества экземпляров класса.

Статические методы могут использовать только статические переменные и вызывать статические методы. Они не могут обращаться к переменным или методам экземпляра напрямую, без ссылки на объект. На нестатическую переменную X нельзя ссылаться из статического контекста, а на нестатический метод X нельзя ссылаться из статического контекста.

* Локальные переменные — это "рабочие лошадки" Java. Они используются для хранения промежуточных результатов вычислений. И, в отличие от полей, объявляются, инициализируются и используются в одном блоке. Для понимания кода часто более важны имя и инициализатор, чем тип локальной переменной.

Есть еще статические блоки инициализации (это см. дальше).

14. Может ли статический метод быть переопределён или перегружен?

Перегружать можно, переопределять нет. В случае со статическими методами это не переопределение (overriding), а сокрытие (hiding).

15. Могут ли нестатические методы перегрузить статические?

Да. В итоге получится два разных метода.

Статический будет принадлежать классу и будет доступен через его имя, а нестатический будет принадлежать конкретному объекту и доступен через вызов метода этого объекта.

16. Можно ли сузить уровень доступа/тип возвращаемого значения при переопределении метода?

Модификатор доступа сузить нельзя, можно расширить (или ничего не делать). Если типы возвращаемого значения совместимы, то сузить можем.

17. Что можно изменить в сигнатуре метода при переопределении? Можно ли менять модификаторы (throws и тп)?

В сигнатуре метода ничего изменять нельзя. Можно только расширить уровень доступа.

18. Могут ли классы быть статическими?

Вложенные классы могут.

Справка:

Запечатанные классы **предотвращают наследование**. Поскольку их нельзя использовать в качестве базовых классов, определенная оптимизация во время выполнения позволяет несколько ускорить вызов членов запечатанных классов.

Запечатанные типы (sealed) — это классы или интерфейсы, которые накладывают ограничения на другие классы или интерфейсы, которые могут расширять или реализовывать их. С большой долей вероятности могут появиться в Java 15.

Если подтип является абстрактным, то он неявно становится помеченный sealed модификатором, если его явно не пометить модификатором non-sealed.

19. Что означает модификатор final? К чему он может быть применен?

Модификатор final используется для завершения реализации классов, методов и переменных.

Переменная final

Переменная final может быть инициализирована только один раз.

Ссылочная переменная, объявленная как final, никогда не может быть назначена для обозначения другого объекта. Однако данные внутри объекта могут быть изменены. Таким образом, состояние объекта может быть изменено, но не ссылка.

С переменными в Java модификатор final часто используется со static, чтобы сделать константой переменную класса.

Класс final

Если **класс** объявлен как final, то от него нельзя наследоваться.

Метод final

Метод final не может быть переопределен любым подклассом (наследником). В Java модификатор final предотвращает метод от изменений в подклассе.

Главным намерением сделать метод final будет то, что содержание метода не должно быть изменено стороне. Объявление метода, использующего модификатор final в объявление класса, показано в следующем примере:

```
public class Test {  
    public final void changeName(){  
        // тело метода  
    }  
}
```

20. Что такое абстрактные классы? Чем они отличаются от обычных?

Класс может соответствовать не только конкретной сущности предметной области, но и какому-то абстрактному понятию (например, геометрическая фигура shape – абстракция). Если класс объявлен, как абстрактный, то нельзя создавать его экземпляры. Создать можно только экземпляр класса-наследника, не являющегося абстрактным.

Модификатор необходим для создания абстрактных классов и методов. Если класс объявлен как abstract, то единственная цель для него быть расширенным. Любой класс, который расширяет абстрактный класс должен реализовать все абстрактные методы суперкласса, если подкласс не является абстрактным классом.

```
public abstract class Shape {  
  
    private final Color color;  
  
    public Shape(Color color) {  
        this.color = color;  
    }  
  
    public Color getColor() {  
        return color;  
    }  
  
    public abstract double getArea();  
}  
  
public abstract class SuperClass{  
    abstract void m(); //абстрактный метод  
}  
  
class SubClass extends SuperClass{  
    // реализует абстрактный метод  
    void m(){  
        .....  
    }  
}
```

Переменные

класс abstract не может создать экземпляр

Методы

Метод abstract является методом, объявленным с любой реализацией. Тело метода (реализация) обеспечивается подклассом. Методы abstract никогда не могут быть final или strict.

Если класс в Java содержит один или несколько абстрактных методов, то класс должен быть объявлен как abstract. Абстрактный класс не обязан содержать абстрактные методы.

Абстрактные классы решают две задачи:

- определяют набор публичных методов (внешний контракт класса)
- содержат не публичные поля и методы (детали реализации)

21. Может ли быть абстрактный класс без абстрактных методов?

В абстрактном классе может не быть ни одного абстрактного метода.

22. Могут ли быть конструкторы у абстрактных классов? Для чего они нужны?

Да. Необходимы для наследников.

В абстрактном классе в Java можно объявить и определить конструкторы. Даже если вы не объявили никакого конструктора, компилятор добавит в абстрактный класс конструктор по умолчанию без аргументов.

Абстрактные конструкторы будут часто использоваться для обеспечения ограничений класса или инвариантов, таких как минимальные поля, необходимые для настройки класса.

Вопросы: могут ли в абстрактном классе быть статические методы? Да, могут.

А метод main() может быть в абстрактном классе? Да, может.

В том числе psvm? Да

23. Что такое интерфейсы? Какие модификаторы по умолчанию имеют поля и методы интерфейсов?

Цель интерфейса - определение функционала, поведения для реализации его классом. Интерфейс является контрактом, определяющим поведение объекта.

Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы.

И один класс может применять множество интерфейсов.

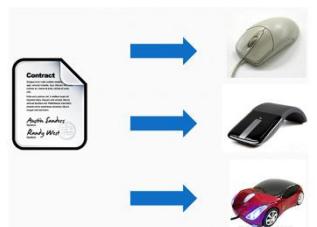
Чтобы класс применил интерфейс, надо использовать ключевое слово **implements**.

Интерфейс — набор абстрактных методов и статических констант. В интерфейсе каждый метод является открытым и абстрактным, но не содержит конструктора. Таким образом, интерфейс в основном представляет собой группу связанных методов с пустыми телами.

Интерфейс определяет как элементы будут взаимодействовать между собой.

- **методы** интерфейса являются public и abstract (публичными и абстрактными)
- **поля** — public static final (константы)

```
1 interface Stateable{  
3     int OPEN = 1;  
4     int CLOSED = 0;  
6     void printState(int n);  
7 }
```



Разные производители мыши (классы), реализующие данный контракт

```
' public interface UserService {  
    User findUserById(long id);  
    List<User> getUsers();  
    void saveUser(User user);  
    void editUser(User user);  
    void deleteById(long id);  
}
```

Объявляемые методы не содержат тел, их объявления завершаются точкой с запятой;

Кроме методов в интерфейсах могут быть определены статические константы. Хотя такие константы также не имеют модификаторов, но по умолчанию они имеют модификатор доступа public static final, и поэтому их значение доступно из любого места программы.

Вложенные интерфейсы

Как и классы, интерфейсы могут быть вложенными, то есть могут быть определены в классах или других интерфейсах.

Например:

```
1 class Printer{  
2     interface Printable {  
3         void print();  
4     }  
5 }  
6 }
```

При применении такого интерфейса нам надо указывать его полное имя вместе с именем класса:

И также как и в случае с классами, интерфейсы могут использоваться в качестве типа параметров метода или в качестве возвращаемого типа.

Интерфейсы маркеры – это интерфейсы, у которых не определены ни методы, ни переменные. Реализация этих интерфейсов придает классу определенные свойства.

Например, интерфейсы **Cloneable** и **Serializable**, отвечающие за клонирование и сохранение объекта в информационном потоке, являются интерфейсами маркерами. Если класс реализует интерфейс Cloneable, это говорит о том, что объекты этого класса могут быть клонированы.

Статические методы

Начиная с JDK 8 в интерфейсах доступны статические методы – они аналогичны методам класса:

Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод:

```
public static void main(String[] args) {  
    Printable.read();  
}
```

Чем похожи класс и интерфейс?

Интерфейс схож с классом следующим образом:

Интерфейс может содержать любое количество методов.

Интерфейс записан в файле с расширением **.java**, и имя интерфейса совпадает с именем файла.

Байт-код интерфейса находится в **.class** файле.

Интерфейсы появляются в пакетах, и их соответствующий файл байт-кода должен быть в структуре каталогов, которая совпадает с именем пакета.

Чем отличается класс от интерфейса?

Однако, интерфейс всё же отличается от класса. Отличие интерфейса от класса в Java:

Вы не можете создать экземпляр интерфейса.

В интерфейсе не содержатся конструкторы.

Все методы в интерфейсе абстрактные.

Интерфейс не может содержать поля экземпляров. Поля, которые могут появиться в интерфейсе, обязаны быть объявлены **static final**.

Интерфейс не расширяется классом, он реализуется классом.

Интерфейс может расширять множество интерфейсов.

24. Чем интерфейсы отличаются от абстрактных классов? В каких случаях следует использовать абстрактный класс, а в каких интерфейс?

1) Ключевые слова **extends** (наследование у класса), **implements** (реализация).

2) Интерфейс **описывает только поведение**. У него нет состояния.

А у абстрактного класса состояние есть: он описывает и то, и другое.

Интерфейс только описывает поведение. Соответственно, мы не сможем реализовать внутри интерфейса геттеры и сеттеры.

Такова природа интерфейса: он нужен для работы с поведением, а не состоянием.

3) Абстрактный класс связывает между собой и объединяет классы, имеющие очень близкую связь. В то же время, один и тот же **интерфейс могут реализовать классы, у которых вообще нет ничего общего**.

Главное, что с помощью абстрактного класса описать ЛЕТАЮЩИХ мы не можем. Они слишком разные. Но есть общее поведение: они могут летать. Интерфейс идеально подойдет для описания всего на свете, что умеет летать, плавать, прыгать или обладает каким-то другим поведением.

- 4) Классы могут реализовывать сколько угодно интерфейсов, но наследоваться можно только от одного класса.

Множественного наследования в Java нет, а множественная реализация есть.

Отчасти этот пункт вытекает из предыдущего: интерфейс связывает между собой множество разных классов, у которых часто нет ничего общего, а абстрактный класс создается для группы очень близких друг другу классов.

Поэтому логично, что наследоваться можно только от одного такого класса. Абстрактный класс описывает отношения «*is a*».

- 5) Абстрактный класс может реализовывать методы.

Интерфейс может реализовывать дефолтные методы начиная с 8й версии.

Синтаксические отличия интерфейса от абстрактного класса (АК)? 5 отличий

Синтаксические отличия:

- 1) При создании АК (абстрактного класса) используется слово **abstract**, а при определении интерфейса **interface**
- 2) При наследовании от АК используется ключевое слово **extends** (eng - расширяет), а при реализации интерфейса **implements** (eng - реализует)
- 3) У АК есть конструктор (если он не описан, то по умолчанию). У интерфейса конструктора нет.
- 4) Все переменные у интерфейсов неявно являются **public static final** (то есть константами). final подразумевает, что переменной обязательно должно быть присвоено значение во время инициализации. А в АК переменные могут быть любыми - абстрактность класса не накладывает ограничений.
- 5) Интерфейс не может реализовать интерфейс, не может наследовать АК, но может наследовать (используя ключевое слово **extends**) множество других интерфейсов. В то же время АК также может реализовать до 65 535 интерфейсов.

25. Может ли один интерфейс наследоваться от другого? От двух других?

Наследование (или расширение) интерфейсов

Интерфейсы, как и классы, могут наследоваться (через ключевое слово **extends**):

Один интерфейс, в отличие от классов, может расширять несколько интерфейсов.

```
1 interface BookPrintable extends Printable{  
3     void paint();  
4 }
```

При применении этого интерфейса класс Book должен будет реализовать как методы интерфейса BookPrintable, так и методы базового интерфейса Printable.

Множественная реализация интерфейсов

Если нам надо применить в классе несколько интерфейсов, то они все перечисляются через запятую после слова **implements**:

```
1 interface Printable {  
3     // методы интерфейса без параметров!  
4 }  
6 interface Searchable {  
8     // методы интерфейса  
9 }  
10  
11 class Book implements Printable, Searchable{  
13     // реализация класса  
14 }
```

26. Что такое дефолтные методы интерфейсов? Для чего они нужны?

В JDK 8 была добавлена такая функциональность как методы по умолчанию с модификатором default.

И теперь интерфейсы могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Это нужно для обратной совместимости.

Если один или несколько методов добавляются к интерфейсу, все реализации также будут вынуждены их реализовывать. Методы интерфейса по умолчанию являются эффективным способом решения этой проблемы.

Например, программист написал библиотеку, в частности там есть интерфейс. Этот интерфейс используется у сотни разработчиков. Запуская новую версию, программист добавляет новый метод final и после обновления всех библиотек перестанет работать.

А если есть дефолтный метод интерфейса все обновятся, напишут собственную реализацию и все будет работать.

27. Как решается проблема ромбовидного наследования при наследовании интерфейсов при наличии default методов?

Класс, наследующий конфликтующие интерфейсы, должен явно через super определить, какой именно метод вызвать: **InterfaceB.super.method();**

28. Каков порядок вызова конструкторов и блоков инициализации с учётом иерархии классов?

- **Статические** блоки от первого до последнего предка (**от предка до наследника**)
- **Попарно** динамический блок инициализации (не статический) и **конструктор** от первого до последнего предка

Parent static block(s) → **Child static** block(s) → **Grandchild static** block(s)

→ **Parent** non-static block(s) → **Parent** constructor →

→ **Child** non-static block(s) → **Child** constructor →

→ **Grandchild** non-static block(s) → **Grandchild** constructor

29. Зачем нужны и какие бывают блоки инициализации?

Существуют статические и нестатические блоки инициализации.

Инициализация означает, что переменная запущена в работу, ей присвоено начальное значение, она инициализирована.

Без присвоения начального значения переменная просто объявлена, а с начальным значением она еще и инициализирована.

Блоки инициализации представляют собой код, заключенный в фигурные скобки {} и размещаемый **внутри класса вне объявления методов или конструкторов.**

Блок инициализации выполняется перед инициализацией класса загрузчиком классов или созданием объекта класса с помощью конструктора.

Несколько блоков инициализации выполняются в порядке следования в коде класса.

Блок инициализации способен генерировать исключения, если их объявления перечислены в **throws** всех конструкторов класса.

Блок инициализации возможно создать и в анонимном классе.

30. Для чего в Java используются статические блоки инициализации?

Статические блоки инициализации используются для выполнения кода, который должен выполняться один раз при инициализации класса загрузчиком классов, в момент, предшествующий созданию объектов этого класса при помощи конструктора.

Такой блок (в отличие от нестатических, принадлежащих конкретному объекту класса) принадлежит только самому классу (объекту метакласса Class).

31. Что произойдет, если в блоке инициализации возникнет исключительная ситуация?

Для нестатических блоков инициализации, если выбрасывание исключения прописано явным образом требуется, чтобы объявления этих исключений были **перечислены в throws всех конструкторах класса.** Иначе будет ошибка компиляции.

Для статического блока выбрасывание исключения в явном виде приводит к ошибке компиляции.

В остальных случаях, взаимодействие с исключениями будет проходить так же, как и в любом другом месте. Класс не будет инициализирован, если ошибка происходит в статическом блоке и объект класса не будет создан, если ошибка возникает в нестатическом блоке.

32. Какое исключение выбрасывается при возникновении ошибки в блоке инициализации класса?

Если возникшее исключение - наследник RuntimeException:

- для статических блоков инициализации будет выброшено ExceptionInInitializerError
- для нестатических будет выброшено исключение-источник

Если возникшее исключение - наследник Error, то в обоих случаях будет выброшено java.lang.Error.

33. Что такое класс Object?

В Java определен один специальный класс, называемый Object.

```
public class Classes010 {  
    // примеры инициализационных блоков  
  
    String str1, str2, str5;  
    String str3 = "Привет мир!";  
  
    static String str4, str6;  
  
    // инициализационный блок  
    {  
        println("В инициализационном блоке");  
        str1 = "Hello World!";  
    }  
  
    // статический инициализационный блок  
    static {  
        println("В статическом инициализационном блоке 1");  
        str4 = "STATIC";  
    }  
  
    static {  
        println("В статическом инициализационном блоке 2");  
        str6 = "static";  
    }  
  
    Classes010(){  
        println("В конструкторе по умолчанию");  
    }  
}
```

Класс `Object` является корнем иерархии классов (т.к. ООП). У каждого класса есть `Object` как суперкласс. Все объекты, включая массивы, реализуют методы этого класса.

Все остальные классы являются подклассами, производными от этого класса, даже если в объявлении это явно не указано. Поэтому ссылочная переменная класса `Object` может ссылаться на объект любого другого класса.

Так как массивы являются тоже классами, то переменная класса `Object` может ссылаться и на любой массив.

Чтобы с объектом что-то сделать, нужно выполнить приведение типов.

```
Cat cat = (Cat) obj;
```

33.1. Что такое класс `Class`?

В Java все объекты являются экземплярами какого-либо класса. И сами классы являются также объектами. Так вот объекты-классы являются экземплярами класса `Class`.

34. Какие методы есть у класса `Object` (перечислить все)? Что они делают?

В классе `Object` определен ряд методов, которые доступны всем классам языка Java.
6 методов, не считая «многопоточки»:

Методы класса `Object` в Java:

1. `protected Object clone()` - создает новый объект, не отличающийся от клонируемого.
2. `public boolean equals(Object obj)` - определяет, равен ли один объект другому.
3. `protected void finalize()` - вызывается перед удалением неиспользуемого объекта.
4. `public final Class<?> getClass()` - получает класс объекта во время выполнения.
5. `public int hashCode()` - возвращает хэш-код, связанный с вызывающим объектом.
6. `public final void notify()` - возобновляет исполнение потока, ожидающего вызывающего объекта.
7. `public final void notifyAll()` - возобновляет исполнение всех потоков, ожидающих вызывающего объекта.
8. `public String toString()` - возвращает символьную строку, описывающую объект.
9. `public final void wait()` - ожидает другого потока исполнения.
10. `public final void wait(long timeout)` - ожидает другого потока исполнения.
11. `public final void wait(long timeout, int nanos)` - ожидает другого потока исполнения.

35. Расскажите про `equals` и `hashcode`

Эти методы предназначены для получения характеристики логического равенства двух объектов.

Хеш-код — это «цифровой отпечаток», целочисленный результат работы метода, которому в качестве входного параметра передан объект.

Если более точно, то это битовая строка фиксированной длины, полученная из массива произвольной длины.

`Equals` – это метод, определенный в `Object`, который служит для сравнения СОСТОЯНИЯ объектов.

- При сравнении объектов при помощи `==` сравниваются ссылки
- При сравнении по `equals()` сравниваются состояния объектов

При переопределении `equals()` обязательно нужно переопределить метод `hashCode()`.

Равные объекты должны возвращать одинаковые хэш коды.

36. Каким образом реализованы методы `hashCode()` и `equals()` в классе `Object`?

Реализация метода `Object.equals()` сводится к проверке на равенство двух ссылок:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

HashCode реализован так, что для одного и того же входного объекта, хеш-код всегда будет одинаковым.

Реализация метода **Object.hashCode()** описана как **native**, т.е. написана НЕ на ЯП Java.

Конкретная реализация зависит от версии Java

<https://srvaroa.github.io/vm/java/openjdk/biased-locking/2017/01/30/hashCode.html>

Не переопределенный hashCode возвращает идентификационный хеш, основанный на **состоянии потока**, объединённого с **xorshift** (в OpenJDK8).

37. Зачем нужен equals(). Чем он отличается от операции ==?

Обычное сравнение двух объектов через оператор “==” — это плохая идея, потому что знак “==” сравнивает ссылки.

Пример: ссылки car1 и car2 указывают на разные адреса в памяти, поэтому они не равны. Но мы хотим сравнить два объекта, а не две ссылки и лучшее решение для сравнения объектов — это метод **equals()**.

Контракт **equals** (переопределяя метод equals(), обязательно соблюдай контракт). Это не просто набор каких-то «полезных рекомендаций», а именно **жесткий контракт методов**, прописанный в документации Oracle.

Рефлексивность	Любой объект должен быть равен по equals() самому себе. В методе указано: if (this == o) return true;
Симметричность	Если a.equals(b) == true , то и b.equals(a) должно возвращать true.
Транзитивность	Если два объекта равны какому-то третьему объекту, то, они должны быть равны друг и другу. Если a.equals(b) == true и a.equals(c) == true , значит проверка b.equals(c) тоже должна возвращать true.
Постоянство	Результаты работы equals() должны меняться только при изменении входящих в него полей. Если данные двух объектов не менялись, результаты проверки на equals() должны быть всегда одинаковыми.
Неравенство с null	Для любого объекта проверка a.equals(null) должна возвращать false.

```
class MyClass {  
    double a;  
    Object b;  
    private int internalField; // Служебное свойство, не участвует в сравнении  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true; // Рефлексивность  
        if (o == null) return false; // Неравенство null  
        if (getClass() != o.getClass()) return false; // Симметричность  
        MyClass other = (MyClass) o;  
        return other.a == this.a && (this.b == null && other.b == null || b.equals(other.b));  
    }  
}
```

38. Правила переопределения equals()

Общий алгоритм переопределения equals

- 1) Проверить на равенство ссылки объектов this и параметра метода o.
`if (this == o) return true;`
- 2) Проверить, определена ли ссылка o, т. е. является ли она null.
Если в дальнейшем при сравнении типов объектов будет использоваться оператор instanceof, этот пункт можно пропустить, т. к. этот параметр возвращает false в данном случае null instanceof Object.
- 3) Сравнить типы объектов this и o с помощью оператора instanceof или метода getClass(), руководствуясь описанием выше и собственным чутьем.
- 4) Если метод equals переопределяется в подклассе, не забудьте сделать вызов super.equals(o)
- 5) Выполнить преобразование типа параметра o к требуемому классу.
- 6) Выполнить сравнение всех значимых полей объектов:
для примитивных типов (кроме float и double), используя оператор ==
для ссылочных полей необходимо вызывать их метод equals
для массивов можно воспользоваться перебором по циклу, либо методом Arrays.equals()
для типов float и double необходимо использовать методы сравнения соответствующих оберточных классов Float.compare() и Double.compare()
- 7) И, наконец, ответить на три вопроса: является ли реализованный метод симметричным? Транзитивным? Согласованным?
Два других принципа (рефлексивность и определенность), как правило, выполняются автоматически.

39. Что будет, если переопределить equals() не переопределяя hashCode()? Какие могут возникнуть проблемы?

Нарушится контракт. Классы и методы, которые использовали правила этого контракта могут некорректно работать. Так для объекта HashMap это может привести к тому, что пара, которая была помещена в Map, возможно не будет найдена в ней при обращении к Map, если используется новый экземпляр ключа.

Методы equals и hashCode необходимо переопределять вместе

Исходя из описанных выше контрактов следует, что переопределяя в своем коде метод equals, необходимо всегда переопределять и метод hashCode.
Так как фактически два экземпляра класса отличаются, потому что находятся в разных областях памяти, сравнивать их приходится по некоторым логическим признакам. Соответственно, два логически эквивалентных объекта, должны возвращать одинаковое значение хэш-функции.

Что произойдет, если будет переопределен только один из этих методов?

- 1) **equals есть, hashCode нет.**
Допустим мы правильно определили метод equals в нашем классе, а метод hashCode решили оставить как он есть в классе Object.
Тогда с точки зрения метода equals два объекта будут логически равны, в то время как с точки зрения метода hashCode они не будут иметь ничего общего. И, таким образом, помещая некий объект в хэш-таблицу, мы рискуем не получить его обратно по ключу.
- 2) **hashCode есть, equals нет.**
Что будет если мы переопределим метод hashCode, а реализацию метода equals унаследуем из класса Object.
Как известно метод equals по умолчанию просто сравнивает указатели на объекты, определяя, являются ли они на один и тот же объект.
Для успешного поиска объекта в хэш-таблице помимо сравнения хэш-значений ключа используется также определение логического равенства ключа с искомым объектом. Т. е. без переопределения метода equals никак не получится обойтись.

40. Какой контракт между hashCode() и equals()?

СОГЛАШЕНИЕ ИЛИ КОНТРАКТ

между equals и hashCode в Java

- 1) Если объекты равны по результатам выполнения метода equals, тогда их hashCode должны быть одинаковыми.
- 2) Если объекты не равны по результатам выполнения метода equals, тогда их hashCode могут быть как одинаковыми, так и разными.
Однако для повышения производительности, лучше, чтобы разные объекты возвращали разные коды.

если a.equals(b), то a.hashCode() == b.hashCode()

41. Для чего нужен метод hashCode()?

Метод hashCode() – это ПРИКЛЮЧЕНИЕ: объекты возможно равны? Если ок, то вызываем equals() для ТОЧНОГО сравнения. Разные коды, следовательно содержание разное.

Хэш — это числовой отпечаток объекта, мета-инфо о состоянии объекта), некоторое число, генерируемое на основе объекта и описывающее его состояние в какой-то момент времени.

Это число используется в Java преимущественно в хэш-таблицах, таких как HashMap.

При этом хэш-функция получения числа на основе объекта должна быть реализована таким образом, чтобы обеспечить относительно равномерное распределение элементов по хэш-таблице.

А также минимизировать вероятность появления **коллизий**, когда по разным ключам функция вернет одинаковое значение.

Для понимания нюансов <https://www.youtube.com/watch?v=ZcZQEgxmjOQ>

42. - Правила переопределения метода hashCode().

Контракт hashCode

Для реализации хэш-функции в спецификации языка определены следующие правила:

- 1) Вызов метода hashCode один и более раз над одним и тем же объектом должен возвращать **одно и то же хэш-значение**, при условии что поля объекта, участвующие в вычислении значения, не изменились.
- 2) Вызов метода hashCode над двумя объектами должен всегда возвращать **одно и то же число**, если эти объекты равны (вызов метода equals для этих объектов возвращает true).
- 3) Вызов метода hashCode над двумя неравными между собой объектами должен возвращать **разные хэш-значения**. Хотя это требование и не является обязательным, следует учитывать, что его выполнение положительно повлияет на производительность работы хэш-таблиц.

43. Есть ли какие-либо рекомендации о том, какие поля следует использовать при подсчете hashCode()?

Выбирать поля, которые с большой долей вероятности будут различаться. Для этого необходимо использовать **уникальные, лучше всего примитивные поля**, например такие как id, uuid. При этом нужно следовать правилу, если поля задействованы при вычислении hashCode(), то они должны быть задействованы и при выполнении equals().

44. Могут ли у разных объектов быть одинаковые hashCode()?

Да. Ситуация, когда у разных объектов одинаковые хэшкоды, называется **коллизией**. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хэш-кода.

45. Почему нельзя реализовать hashCode() который будет гарантированно уникальным для каждого объекта?

В Java множество возможных хэш кодов **ограничено типом int**, а множество объектов ничем не ограничено. Из-за этого, вполне возможна ситуация, что хэш коды разных объектов могут совпасть. Например:

```
fun main() {  
    //generates 310265957 as hashCode  
    print("I have a common prefixDB".hashCode())  
  
    //generates 310265957 as hashCode  
    print("I have a common prefixCa".hashCode())  
}
```

Вообще если длина строк одинаковая и выполняется:

```
31*(b[n-2] - a[n-2]) == (a[n-1] - b[n-1])
```

то, хэши всегда будут одинаковыми, не буду вдаваться в подробности почему так :D

Хэшкод может быть и отрицательным!

46. Есть класс Point {int x, y;}.

Почему хэш-код в виде $31 * x + y$ предпочтительнее чем $x + y$?

Множитель создает зависимость значения хэш-кода от очередности обработки полей, а это дает гораздо лучшую хэш-функцию.

Алгоритм доступно и близко к практике (работа с данными) <https://youtu.be/lWnzRILIEZ0>

```
override fun hashCode(): Int {
    var result = id.hashCode()
    result = 31 * result + name.hashCode()
    result = 31 * result + (mum?.hashCode() ?: 0) // Если объект null, он не влияет
    return result
}
```

У String, кстати свой алгоритм: $h(s)=\sum(s[index] * 31^{(n-1-index)})$ Смысл тот же, только проходим по всем char

Справка: почему 31?

Как вы заметили везде используется это непонятное число 31. А по каким критериям оно отбирается?

- Число должно быть простым и нечетным, чтобы уменьшить вероятность коллизий.
- Число должно занимать мало места в памяти
- Умножение должно выполняться быстро

Оказывается, 31 идеальный кандидат ведь:

- Оно простое и нечетное
- Занимает всего 1 байт в памяти
- Умножение можно заменить на быстрый побитовой сдвиг. $31 * i == (i << 5) - i$

PS: А че за $(i << x) - i$?

Побитовой сдвиг влево на x позиций. Работает точно также как умножить какое-то число на 2 x раз.

```
println(n.shl(1)) // 6
println(n.shl(2)) // 12
println(n.shl(3)) // 24
```

47. Чем `a.getClass().equals(A.class)` отличается от `a instanceof A.class`

`getClass()` получает только класс, а оператор `instanceof` проверяет является ли объект экземпляром или потомком класса.

`Instanceof` проверяет всю цепочку наследования, а дерево может состоять из множества объектов (это **один из самых медленных операторов**).

5. Исключения

1. Что такое исключения?

Исключение в Java — это **объект**, который описывает исключительное состояние, возникшее в каком-либо участке программного кода (событие, ошибка).

Когда возникает исключительное состояние, создается объект класса `Exception`. Этот объект пересыпается в метод, обрабатывающий данный тип исключительной ситуации. Исключения могут возбуждаться и для того, чтобы сообщить о некоторых нештатных ситуациях.

```
public final class String {
    @Override
    public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
        if (anObject instanceof String) {
            String other = (String)anObject;
            // ...
        }
        return false;
    }
}
```

null → false
→ false

2. Опишите иерархию исключений.

Исключения делятся на несколько классов, но все они имеют общего предка — класс **Throwable** (в свою очередь он – наследник `Object`), его потомки – это подклассы `Exception` (исключения) и `Error` (ошибки).

Класс Exception используется для описания исключительных ситуаций, которые должны перехватываться программным кодом пользователя.

Класс Error предназначен для описания исключительных ситуаций, которые при обычных условиях не должны перехватываться в пользовательской программе.

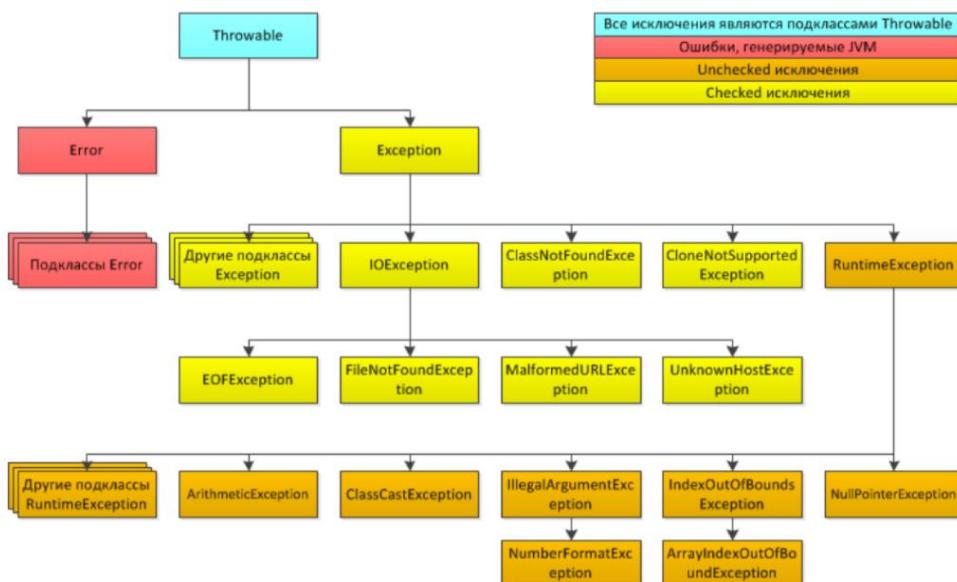
Все исключительные ситуации делятся на «проверяемые» (checked) и «непроверяемые» (unchecked).

Checked исключения отличаются от Unchecked исключения в Java, тем что:

Наличие\обработка Checked исключения проверяются **на этапе компиляции**.
Наличие\обработка Unchecked исключения происходит **на этапе выполнения**.

Checked исключения, это те, которые **должны обрабатываться блоком catch или описываться в сигнатуре метода**.

Unchecked исключения в Java **могут не обрабатываться и не быть описаными**.



Вопрос со звёздочкой:

Есть исключение `OutOfMemoryError`, что это? (Исключение выбрасывается, когда в куче заканчивается память). Все исключения – это объекты.

Внимание, вопрос: Каким образом выбрасывается это исключение, если память закончилась?

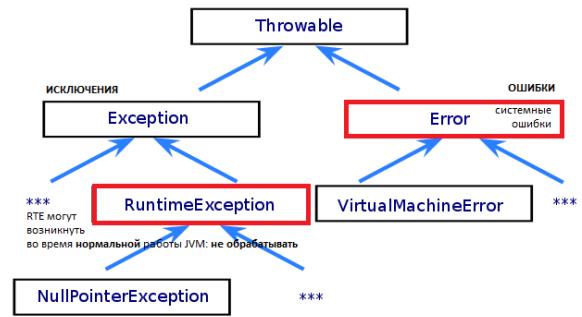
Ответ: этот объект создаётся при запуске этого приложения и ждёт своего часа.

Сейчас даже можно отловить такое исключение (Эпсилон GC и фантомные ссылки)

3. Расскажите про обрабатываемые и необрабатываемые исключения

Checked исключения наследуются от `Exception` (кроме ветки `Runtime`). Это те, которые должны обрабатываться блоком `catch` или описываться в сигнатуре метода.

Unchecked исключения (наследованные от `RuntimeException`) могут не обрабатываться и не быть описанными.



РАЗЛИЧИЯ:

- Наличие обработки **Checked исключения** проверяются на этапе компиляции (проверяемое компилятором)
- Наличие обработки **Unchecked исключения** происходит на этапе выполнения (необрабатываемые)

СИНТАКСИС И ИДЕОЛОГИЯ:

- Можно ловить и checked, и unchecked, но ловить unchecked не нужно. И обрабатывать, собственно, нужно только checked исключения.
- Unchecked исключения не `Error`, но `RunTimeException` это больше ошибка кода (программиста), которую нужно исправить.

4. Можно ли обработать необрабатываемые исключения?

Можно, чтобы в некоторых случаях программа не прекратила работу?

Обрабатывать можно, но делать этого не стоит. Разработчику не предоставлены инструменты для обработки ошибок системы и виртуальной машины.

5. Какой оператор позволяет принудительно выбросить исключение?

Оператор **throw** `new Exception();`

Авторский вопрос: можно так написать и что будет выведено в консоль?

Ответ: будет выброшено исключение `ClassCastException` (т.к. объект находится выше в иерархии наследования).

```

public class Main {
    public static void main(String[] args) throws Throwable {
        throw (Throwable) new Object();
    }
}
    
```

6. О чём говорит ключевое слово `throws`?

Метод потенциально может выбросить исключение с указанным типом (элемент контракта метода). Передаёт обработку исключения вышестоящему методу.

7. Как создать собственное («пользовательское») исключение?

Необходимо унаследоваться от базового класса требуемого типа исключений (например, от `Exception` или `RuntimeException`) и переопределить методы.

8. Расскажите про механизм обработки исключений в java (Try-catch-finally)

```
1 try {  
2 //здесь код, который потенциально может привести к ошибке  
3 }  
4 catch (SomeException e) { //в скобках указывается класс конкретной ожидаемой ошибки  
5 //здесь описываются действия, направленные на обработку исключений  
6 }  
7 finally {  
8 //выполняется в любом случае ( блок finally не обязателен)  
9 }
```

- Try – блок, в котором может появиться исключение
- Catch – блок, в котором мы указываем исключение и логику его обработки
- Finally – блок, который обязательно отработает (кроме исключительных случаев)

Блок finally всегда выполняется после кода в предыдущем блоке try. Не имеет значения, выдает ли блок try исключение, перехвачено ли оно или нет, выполняет ли он оператор return.

Когда генерируется исключение, выполнение метода направляется по нелинейному пути. Если файл открывается в начале метода и закрывается в конце, то вряд ли кто-нибудь устроит, что код, закрывающий файл, будет обойден механизмом обработки исключений.

Для таких непредвиденных обстоятельств и служит оператор finally.

Оператор finally образует блок кода, который будет выполнен по завершении блока операторов try-catch, но перед следующим за ним кодом.

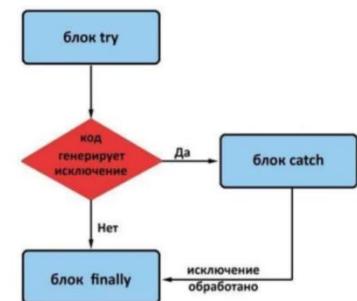
Блок оператора finally выполняется независимо от того, выброшено ли исключение или нет. Если исключение выброшено, блок оператора finally выполняется, даже при условии, что ни один из операторов catch не совпадает с этим исключением.

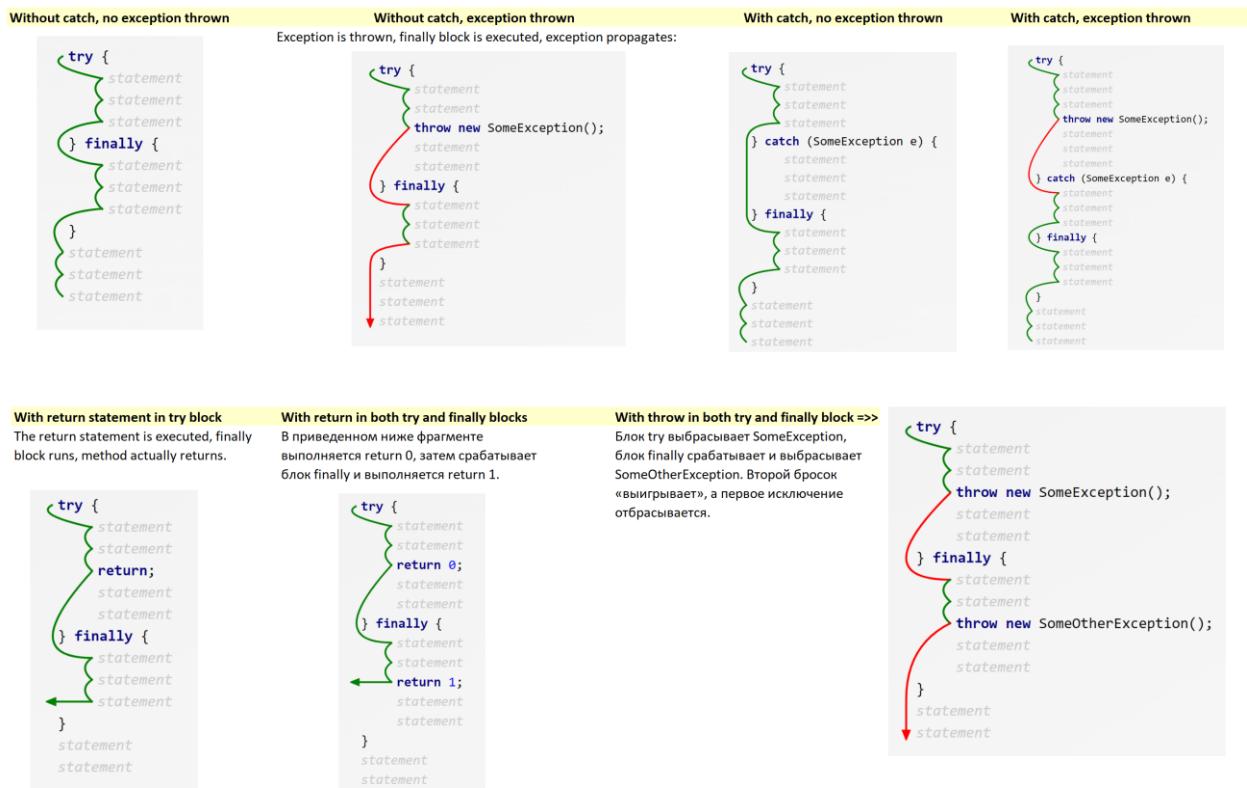
В любой момент, когда метод собирается возвратить управление вызывающему коду из блока оператора try-catch (через необработанное исключение или явным образом через оператор return), блок оператора finally выполняется перед возвратом управления из метода.

Finally часто используется для восстановления некоторого состояния после операции, например освобождения ресурсов.

9. Возможно ли использование блока try-finally (без catch)?

<https://programming-guide.java/try-finally.html>





10. Может ли один блок catch отлавливать сразу несколько исключений?

Да, в порядке **от частного случая к общему**, например:

Объявление нескольких исключений в одном блоке catch (multi-catch блок)

```
try {
    ...
} catch(IllegalStateException | SQLException | ContextException e) {
    System.out.println(e.getMessage());
}
```

// Логическое OR (ИЛИ), знак |

11. Всегда ли выполняется блок finally? Существуют ли ситуации, когда блок finally не будет выполнен?

Когда исключение передано, выполнение метода направляется по нелинейному пути.

Это может стать источником проблем. Например, при входе метод открывает файл и закрывает при выходе.

Чтобы закрытие файла не было пропущено из-за обработки исключения, был предложен механизм **finally**.

Ключевое слово **finally** создаёт блок кода, который будет выполнен после завершения блока **try/catch**, но перед кодом, следующим за ним. Блок будет выполнен, независимо от того, передано исключение или нет. Оператор **finally** не обязателен, однако каждый оператор **try** требует наличия либо **catch**, либо **finally**.

Код в блоке **finally** будет выполнен **всегда**.

Когда еще НЕ будет выполнен?

1. Если вы вызываете `System.exit()`
2. Если JVM падает сначала
3. Если JVM достигает бесконечного цикла (или некоторого другого непрерываемого, не завершающего оператора) в блоке `try` или `catch`
4. Если ОС принудительно завершает процесс JVM; например, `kill -9 <pid>` в UNIX
5. Если хост-система умирает; например, сбой питания, аппаратная ошибка, паника ОС и т.д.
6. Если блок `finally` будет выполняться потоком демона и все остальные потоки, не являющиеся демонами, завершатся до вызова `finally`

Единственный «верный» способ предотвратить запуск блока finally — завершить работу JVM с помощью System.exit или JVM убить ее вручную.

```
1 try {  
2     System.exit(0);  
3 } catch(Exception e) {  
4     e.printStackTrace();  
5 } finally {}
```

Здесь finally недостижим, так как происходит системный выход из программы.

Общими словами: когда jvm умирает, ей не до finally (отсюда можете придумать другие примеры как убить jvm и ответить на вопрос в заголовке).

12. Может ли метод main() выбросить исключение во вне и если да, то где будет происходить обработка данного исключения?

Может и оно будет передано в виртуальную машину Java (JVM).

Для случая с методом main произойдет две вещи:

- будет завершен главный поток приложения
- будет вызван **ThreadGroup.uncaughtException**

13. В каком порядке следует обрабатывать исключения в catch блоках?

От наследника к предку. От частного к общему (родителю).

Общая форма блока обработки исключений try-catch-finally

```
public class ExceptionDemo1 {  
    public static void main(String[] args) {  
        try {  
            // блок кода, в котором отслеживаются ошибки  
        } catch (ExceptionType1 e) {  
            // обработчик исключений типа ExceptionType1  
        } catch (ExceptionType2 e) {  
            // обработчик исключений типа ExceptionType2  
        } finally {  
            // блок кода, который должен быть выполнен после завершения блока try  
        }  
    }  
}
```

catch — полиморфная конструкция, т.е. catch по типу Parent перехватывает летящие экземпляры любого типа, который является Parent-ом (т.е. экземпляры непосредственно Parent-а или любого потомка Parent-а)

catch по потомку не может поймать предка!

catch по одному «брать» не может поймать другого «брата» (Error и Exception не находятся в отношении предок-потомок, они из параллельных веток наследования от Throwable)

Есть такое правило — нельзя ставить потомка после предка! (RuntimeException после Exception). Ставить брата после брата — можно (RuntimeException после Error).

В некоторых случаях один фрагмент кода может инициировать более одного исключения. Используется два или более операторов catch, каждый для перехвата своего типа исключений.

Когда возбуждается исключение, каждый оператор catch проверяется по порядку, и первый из них, чей тип соответствует исключению, выполняется. После того как выполнится ОДИН из операторов catch, все остальные пропускаются, и выполнение программы продолжается с места, следующего за блоком try-catch.

Когда используются множественные операторы catch, важно помнить, что подклассы исключений должны следовать перед любыми их суперклассами (родителями). Это потому, что оператор catch, который использует суперкласс, будет перехватывать все исключения этого суперкласса плюс всех его подклассов.

То есть подкласс исключения никогда не будет обработан, если вы попытаетесь его перехватить после его суперкласса.

В Java недостижимый код является ошибкой:

```
public class SuperSubCatch {  
    public static void main(String[] args) {  
        try {  
            int a = 0;  
            int b = 42 / a;  
        } catch (Exception e) {  
            System.out.println("Перехват исключений общего класса Exception.");  
        } /*catch (ArithmaticException e) {  
            //ОШИБКА: недостижимый код !  
            System.out.println("Этот код вообще недостижим.");  
        }*/  
    }  
}
```

14. Что такое механизм try-with-resources?

Механизм дает возможность объявлять один или несколько ресурсов в блоке try, которые будут закрыты автоматически без использования finally блока.

В качестве ресурса можно использовать любой объект, класс которого реализует интерфейс `java.lang.AutoCloseable` или `java.io.Closeable`.

try-catch с ресурсами (спец. конструкция) – это улучшенное управление ресурсами, которые должны быть закрыты после окончания работы с ними. Избавляет разработчиков от обязанности освобождать ресурсы, используемые в блоке try.

Кроме того, код, использующий try-with-resources, часто является более чистым и читаемым, что облегчает управление кодом.

Для того чтобы это автоматическое закрытие работало создан специальный интерфейс `AutoCloseable`.

Ресурсы **создаются** только в блоке try.

В какой момент **закрываются** ресурсы? При выходе из блока try.

Справка:

Интерфейс AutoCloseable.

Для того, чтобы класс можно было использовать в операторе try с ресурсами, он должен реализовывать интерфейс `AutoCloseable`. Хорошая новость, в том, что сделать это не так уж и сложно – необходимо реализовать всего лишь один метод – `public void close() throws Exception`.

Открытый интерфейс Closeable (расширяет AutoCloseable)

`A Closeable` — это источник или место назначения данных, которые можно закрыть. Метод `close()` вызывается для освобождения ресурсов, удерживаемых объектом (например, открытых файлов).

Синтаксис

Синтаксис try-with-resources почти идентичен обычному синтаксису try-catch-finally. Единственное различие – это скобки после try, в которых мы объявляем, какие ресурсы мы будем использовать:

```
BufferedWriter writer = null;  
try {  
    writer = new BufferedWriter(new FileWriter(fileName));  
    writer.write(str); // do something with the file we've opened // сделать что-нибудь с файлом, который мы открыли  
} catch (IOException e) {  
    // handle the exception // обрабатывать исключение  
} finally {  
    try {  
        if (writer != null)  
            writer.close();  
    } catch (IOException e) {  
        // handle the exception // обрабатывать исключение  
    }  
}
```

Тот же код, написанный с использованием **try-with-resources**, будет выглядеть следующим образом:

```
try (BufferedWriter writer = new BufferedWriter(new FileWriter(fileName))) {  
    writer.write(str); // do something with the file we've opened // сделать что-нибудь с файлом, который мы открыли  
}  
catch(IOException e){  
    // handle the exception // обрабатывать исключение  
}
```

Как Java понимает этот код:

Ресурсы, открытые в скобках после оператора try, понадобятся только здесь и сейчас.

Я вызову их методы .close(), как только закончу работу в блоке try.

Если в блоке try возникнет исключение, я все равно закрою эти ресурсы.

Множество Ресурсов

Еще одним хорошим аспектом try-catch-with-resource является простота добавления/удаления ресурсов, которые мы используем, с гарантией того, что они будут закрыты после того, как мы закончим.

Если бы мы хотели работать с несколькими файлами, мы бы открыли файлы в инструкции try() и разделили их точкой с запятой:

```
try (BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));  
     Scanner scanner = new Scanner(System.in)) {  
if (scanner.hasNextLine())  
    writer.write(scanner.nextLine());  
}  
catch(IOException e) {  
    // handle the exception  
}
```

Затем Java позаботится о том, чтобы **вызвать .close() на всех ресурсах, которые мы открыли в try()**.

Важно: ресурсы закрываются в обратном порядке объявления, что означает, что в этом примере сканер будет закрыт перед автором.

15. Что произойдет если исключение будет выброшено из блока catch после чего другое исключение будет выброшено из блока finally?

finally-секция может «перебить» throw/return при помощи другого throw/return

16. Что произойдет если исключение будет выброшено из блока catch после чего другое исключение будет выброшено из метода close() при использовании try-with-resources?

В try-with-resources добавлена возможность хранения "подавленных" исключений, и брошенное try-блоком исключение имеет больший приоритет, чем исключения, получившиеся во время закрытия.

Подавленные исключения (suppressed exception) образуются, когда в блоке try генерируется исключение и в методе close().

При закрытии ресурса генерируется исключение, в этом случае ПЕРВОЕ исключение считается ГЛАВНЫМ остальные ПОДАВЛЕННЫЕ.

Если исключение создается из блока Java try-with-resources,
любой ресурс, открытый в круглых скобках блока try, все равно будет автоматически закрыт.
 Как упоминалось ранее, try-with-resources работает так же, как try-catch-finally, за исключением одного небольшого дополнения.
Добавление называется подавленные исключения.

ПРИМЕР:

По какой-то причине в блоке try-with-resources возникает исключение
 Java останавливает выполнение в блоке try-with-resources и вызывает .close() для всех ресурсов, объявленных в try()
 Один из методов .close() создает исключение
 Какое исключение будет “ловить” блок catch?

Подавленное исключение - это исключение, которое выбрасывается, но каким-то образом **игнорируется**.
 Эти исключения являются исключениями, которые **встречаются в методах .close()**, и доступ к ним осуществляется иначе, чем к “обычным” исключениям.

Распространенный сценарий для этого в Java-это когда блок finally создает исключение.
Любое исключение, первоначально возникшее в блоке try, затем подавляется.
 Начиная с Java 7, теперь мы можем использовать два метода в классе Throwable для обработки подавленных исключений: addSuppressed и getSuppressed

6. Сериализация и копирование

1. Что такое сериализация и как она реализована в Java?

Сериализация – это процесс сохранения состояния объекта в последовательность байт (сохранение в поток для дальнейшего использования или хранения этого объекта);

Реализована через интерфейс-маркер (без методов) **Serializable**.

2. Для чего нужна сериализация?

Для компактного сохранения состояния объекта и считывание этого состояния.

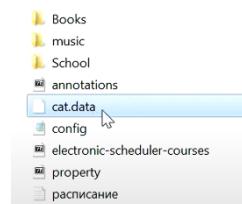
Можно взять объект из памяти JVM, превратить его в набор байтов вместе со всеми значениями его полей и дальше работать с этим набором байт. Например, записать в файл или отправить по сети. А потом этот набор байт можно прочитать, декодировать и получить точно такой же объект!

3. Опишите процесс сериализации/десериализации с использованием Serializable.

Запомнить 4 этапа:

- 1) записываются метаданные
- 2) рекурсивно записывается информация (описание классов) от родителя до наследника
- 3) примитивы записываются
- 4) пишутся в поток

- 1) Класс объекта должен реализовывать интерфейс Serializable
- 2) Создать поток ObjectOutputStream (oos), который записывает объект в переданный OutputStream.
- 3) Записать в поток: oos.writeObject(Object);
- 4) Сделать oos.flush() и oos.close() (очистить буфер и закрыть)



```
import java.io.Serializable;

public class Cat implements Serializable {

    private String name; []

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```
Main.java x Cat.java x Serializer.java x
Serializer serialization()

try {
    FileOutputStream fos = new FileOutputStream(file);
    if(fos!=null) {
        oos = new ObjectOutputStream(fos);
        oos.writeObject(cat);
        flag = true;
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if(oos != null){
        try {
            oos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

return flag;
```

4. Как изменить стандартное поведение сериализации/десериализации?

Использовать интерфейс Externalizable. Переопределить методы
writeExternal(ObjectOutput out) throws IOException
readExternal(ObjectInput in) throws IOException, ClassNotFoundException

Не путать:

	Поток ВВОДА (как сканер)	Поток ВЫВОДА (как принтер)
Работает с байтами	InputStream	OutputStream
Работает с символами (Unicode)	Reader (чтение)	Writer (запись)

Потоки ввода/вывода

<https://docs.oracle.com/javase/tutorial/essential/io/>

Потоки байтов (Byte Streams) обрабатывают ввод-вывод необработанных двоичных данных.

Потоки символов (Character Streams) обрабатывают ввод-вывод символьных данных, автоматически выполняя преобразование в локальный набор символов и обратно.

Буферизированные потоки (Buffered Streams) оптимизируют ввод и вывод, сокращая количество вызовов собственного API.

Сканирование и форматирование (Scanning and Formatting) позволяет программе читать и писать форматированный текст.

Ввод/вывод из командной строки (I/O from the Command Line) описывает стандартные потоки и объект консоли.

Потоки данных (Data Streams) обрабатывают двоичный ввод-вывод примитивных типов данных и Stringзначений.

Потоки объектов (Object Streams) обрабатывают двоичный ввод-вывод объектов.

5. Какие поля не будут сериализованы при сериализации? Будет ли сериализовано final поле?

- 1) Добавить к полю модификатор **transient**.

В таком случае после восстановления его значение будет null.

Иногда бывает так, что в объекте есть поля, которые не нужно сохранять.

Это может быть какой-то кэш или другие данные, которые легко вычисляются по тому, что есть в объекте.

Их потеря ничем не грозит, зато позволит сократить размер памяти, занимаемой объектом при сериализации.

Отметить поля ключевым словом **Transient** и JVM будет их игнорировать при записи.

Свойства класса, помеченные модификатором **transient, не сериализуются.**

- 2) Сделать поле **static**. Значения статических полей автоматически не сохраняются.
- 3) Поля с **модификатором final** сериализуются как и обычные. За одним исключением – их невозможно десериализовать при использовании Externalizable, поскольку final-поля должны быть инициализированы в конструкторе, а после этого в readExternal изменить значение этого поля будет невозможно. Соответственно, если необходимо сериализовать объект с final-полем, то необходимо использовать только стандартную сериализацию.
- 4) Аннотация **@Transient**

6. Как создать собственный протокол сериализации?

Для создания собственного протокола нужно просто переопределить writeExternal() и readExternal().

В отличие от двух других вариантов сериализации, здесь ничего не делается автоматически. Протокол полностью в ваших руках.

7. Какая роль поля `serialVersionUID` в сериализации?

Переменная `private static final long serialVersionUID`

Это поле содержит уникальный идентификатор версии сериализованного класса.

Идентификатор версии есть у любого класса, который implements интерфейс `Serializable`.

Он вычисляется по содержимому класса — полям, порядку объявления, методам.

И если мы поменяем в нашем классе тип поля и/или количество полей, идентификатор версии моментально изменится.

`serialVersionUID` тоже записывается при сериализации класса.

ВАЖНО!

Если мы не объявляем это поле явно, то при любом изменении класса (даже если пробел добавим куда-нибудь), версия изменится (`Java` сделает это за нас).

Руками нужно менять, чтобы программист сам понимал, **являются ли изменения обратно-совместимыми** или нет.

Если что-то изменил и это обратно-совместимо, версию `serialVersionUID` менять НЕ нужно, а если изменил и это обратно-НЕ совместимо, то нужно поменять версию.

8. Когда стоит изменять значение поля `serialVersionUID`?

Следует изменять `serialVersionUID` только тогда, когда вы **сознательно хотите нарушить совместимость** со всеми существующими сериализациями, например, когда изменения в вашем классе сделают его настолько семантически отличным, что у вас не будет выбора.

В этом случае действительно нужно несколько раз подумать о том, зачем вы это делаете.

9. В чем проблема сериализации `Singleton`?

Проблема: после десериализации мы **получим другой объект**.

Так сериализация дает возможность создать `Singleton` еще раз, что не совсем нам нужно.

Решение: в классе определяется метод с сигнатурой "Object `readResolve()` throws `ObjectStreamException`"

Назначение этого метода: возвращать замещающий объект вместо объекта, на котором он вызван.

10. Расскажите про клонирование объектов.

в `Java`, есть 3 (4) способа клонирования объекта:

- 1) С использованием **интерфейса `Cloneable`**;

Первый способ подразумевает, что вы будете использовать механизм так называемого **«поверхностного клонирования»** и сами позаботитесь о клонировании полей-объектов. Метод `clone()` в родительском классе `Object` является **protected**, поэтому требуется переопределение его с объявлением как **public**. Он возвращает экземпляр объекта с копированными полями-примитивами и ссылками. И получается, что у оригинала и его клона **поля-ссылки указывают на одни и те же объекты**.

Ниже рассмотрен пример **неполного клонирования (Shallow cloning)**.

Этот вид клонирования применяется в том случае, если объект не содержит сложных объектов. Тут добавили метод clone() типа User, который через super вызывает метод clone у класса Object.

```
public class User implements Cloneable {
    private int id;
    private String name;

    public User(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

Соответственно изменился и клиентский код:

```
1 User u1 = new User(1, "Name 1");
2
3 User u2 = null;
4 try {
5     u2 = u1.clone();
6 } catch (CloneNotSupportedException e) {
7     e.printStackTrace();
8 }
9 u2.setId(2);
10 System.out.println(u1);
11 System.out.println(u2);
```

Вывод на консоль:

```
User{id=1, name='Name 1'}
User{id=2, name='Name 1'}
```

Теперь верно: у объекта u1 id = 1, а у объекта u2 id = 2.

Пример глубокого клонирования (Deep cloning).

Создадим класс Адрес:

Класс User теперь выглядит вот так:

```
1 public class User implements Cloneable {
2
3     private int id;
4     private String name;
5     private Address address;
6
7     public User(int id, String name, Address address) {
8         this.id = id;
9         this.name = name;
10        this.address = address;
11    }
12
13    public Address getAddress() {
14        return address;
15    }
16}
```

Вот так выглядит класс Address:

```
1 public class Address implements Cloneable{
2     private String street;
3     private int house;
4
5     public Address(String street, int house) {
6         this.street = street;
7         this.house = house;
8     }
9
10    public Address(int house) {
11        this.house = house;
12    }
13
14}
```

```
1 User u1 = new User(1, "Name 1", new Address("street 1", 10));
2
3 User u2 = null;
4 try {
5     u2 = u1.clone();
6 } catch (CloneNotSupportedException e) {
7     e.printStackTrace();
8 }
9 u2.setId(2);
10 u2.getAddress().setHouse(2);
11 System.out.println(u1);
12 System.out.println(u2);
```

Вывод на консоль:

```
User{id=1, name='Name 1', address=Address{street='street 1', house=10}}
User{id=2, name='Name 1', address=Address{street='street 1', house=2}}
```

Обратите внимание на метод clone в классе User:

а) сначала мы клонируем поля класса User (без поля address)

б) потом мы клонируем address

в) возвращаем объект

Как видим здесь только у объекта u2 номер дома 2, а объекта u1 номер дома 10 не изменился

2) С использованием конструктора копирования (клонирования) объекта;

В классе описывается конструктор, который **принимает объект этого же класса** в качестве параметров и инициализирует значениями его полей поля нового объекта.

Поверхностное клонирование:

Здесь стоит обратить внимание на конструктор:

```
1 | public User (User u) {
2 |     this(u.getId(), u.getName());
3 | }
```

Здесь просто в конструктор передается объект User и вызывается конструктор User(int id, String name) через this.

```
1 | User u1 = new User(1, "Name 1");
2 |
3 | User u2 = new User(u1);
4 |
5 | u2.setId(2);
6 | System.out.println(u1);
7 | System.out.println(u2);
```

Вывод на консоль:

```
User{id=1, name='Name 1'}
User{id=2, name='Name 1'}
```

Глубокое (полное) клонирование с использованием конструктора копирования.

```
1 | public class User {
2 |
3 |     private int id;
4 |     private String name;
5 |     private Address address;
6 |
7 |
8 |     public User(int id, String name, Address address) {
9 |         this.id = id;
10 |        this.name = name;
11 |        this.address = address;
12 |    }
13 |
14 |
15 |
16 |
17 |     public User (User u) {
18 |         this(u.getId(), u.getName(), new Address(u.address));
19 |     }
20 | }
```

```
1 | public class Address {
2 |     private String street;
3 |     private int house;
4 |
5 |
6 |     public Address(String street, int house) {
7 |         this.street = street;
8 |         this.house = house;
9 |     }
10 |
11 |
12 |
13 |
14 |
15 |     public Address(Address a) {
16 |         this(a.getStreet(), a.getHouse());
17 |     }
18 |
19 |     public Address(int house) {
20 |         this.house = house;
21 |     }
22 | }
```

Вызов метода

```
1 | User u1 = new User(1, "Name 1", new Address("street 1", 1));
2 |
3 | User u2 = new User(u1);
4 |
5 | u2.setId(2);
6 | u2.getAddress().setHouse(32);
7 | System.out.println(u1);
8 | System.out.println(u2);
```

Вывод на консоль:

```
User{id=1, name='Name 1', address=Address@1}
User{id=2, name='Name 1', address=Address@2}
```

Как видим, при изменении объекта u2 для поля «Номер дома», данные в объекте u1 не изменились.

3) С использованием **серIALIZАЦИИ**.

Сохраняем объект в поток байт с последующей его «экстремацией» (восстановление).

4) С использованием паттерна **Прототип** (в этом блоке не обязательно знать)

11. В чем отличие между поверхностью и глубоким клонированием?

Поверхностное копирование копирует настолько малую часть информации, насколько это возможно. По умолчанию, клонирование в Java является поверхностью, т.е. Object class не знает о структуре класса, которого он копирует.

При поверхностном клонируются ссылки, а не объекты.

Глубокое копирование дублирует все. **Глубокое копирование — это две коллекции (как пример),** в одну из которых дублируются все элементы оригинальной коллекции.

12. Какой способ клонирования предпочтительней?

Наиболее **безопасным** и, следовательно, предпочтительным способом клонирования является использование **специализированного конструктора копирования**:

- Отсутствие ошибок наследования (не нужно беспокоиться, что у наследников появятся новые поля, которые не будут клонированы через метод `clone()`);
- Поля для клонирования указываются явно;
- Возможность клонировать даже `final` поля.

12. Почему метод `clone()` объявлен в классе `Object`, а не в интерфейсе `Cloneable`?

Метод `clone()` объявлен в классе `Object` с сигнатурой `native`, чтобы обеспечить доступ к стандартному механизму "поверхностного копирования" объектов (копируются значения всех полей, включая ссылки на сторонние объекты).

Метод `clone()` объявлен, **как protected**, чтобы нельзя было вызвать этот метод у не переопределивших его объектов.

13. Как создать глубокую копию объекта? (2 способа)

Глубокое клонирование требует выполнения следующих правил:

Нет необходимости копировать отдельно примитивные данные;

- Все классы-члены в оригинальном классе должны поддерживать клонирование.
- Для каждого члена класса должен вызываться `super.clone()` при переопределении метода `clone()`;

Если какой-либо член класса не поддерживает клонирование, то в методе клонирования необходимо создать новый экземпляр этого класса и скопировать каждый его член со всеми атрибутами в новый объект класса, по одному.

1) Сериализация – это еще один способ глубокого копирования.

Мы просто сериализуем нужный объект и десериализуем его. Очевидно, объект должен поддерживать интерфейс `Serializable`. Мы сохраняем объект в массив байт и потом прочитаем из него.

2) При помощи библиотеки DeepCloneable

Глубокое клонирование с этой библиотекой сводится с двум строкам кода:

- `Cloner cloner = new Cloner();`
- `DeepCloneable clone = cloner.deepClone(this);`